«a2» 2013.11.11

Volume III

Programmazione

«

```
$ cc -Wall -o prog prog.c
$ ./prog
```

ISBN 978-88-905012-2-7

Via Morganella Est, 21 -- I-31050 Ponzano Veneto (TV) -- appunti2@gmail.com

L'opera, nei file sorgenti e nella composizione finale, include degli esempi in forma di sequenze animate, contenenti eventualmente anche delle spiegazioni vocali. Si tratta di video brevi e di qualità molto bassa. Tuttavia, a seconda di come viene diffusa o fruita l'opera, può darsi che sia necessario assolvere a degli obblighi di legge.

Il numero ISBN 978-88-905012-2-7 si riferisce all'opera originale in formato elettronico, pubblicata a titolo gratuito.

Se quest'opera viene consultata in-linea, attraverso uno spazio web, le informazioni generali sull'accesso ai file dell'opera (indirizzo IP di origine, nome del provider della connessione di origine, nome e versione del navigatore usato per accedere, geolocalizzazione dell'origine, data e orario di accesso), assieme al dettaglio delle pagine visitate o dei file scaricati, potrebbero essere annotate in un registro (log) di tale spazio web. Ciò per fini di controllo e statistica, a qualunque titolo. La conservazione di tale registro, se presente, dipende dalla politica del gestore e potrebbe avvenire per una durata di tempo indeterminata.

Quest'opera non contiene tecnologie atte a raccogliere e annotare i dati persona-

li degli utenti che la consultano. Tuttavia, lo spazio web che la ospita potrebbe richiedere una forma di iscrizione o di autenticazione. Se tale iscrizione o autenticazione fosse richiesta, è necessario valutare l'informativa sul trattamento dei dati personali dello spazio web in questione, per sapere come e a che scopo tali dati vengono trattati.

Quest'opera, così come realizzata dal suo autore, non contiene inserzioni pubblicitarie. Tuttavia, lo spazio web che la ospita potrebbe iniettare il codice necessario a somministrare della pubblicità durante la sua consultazione o prima dello scarico dei file. Tali inserzioni pubblicitarie, se ci sono, non hanno nessuna relazione con l'autore di quest'opera e nemmeno vi portano alcun beneficio economico, in quanto servono esclusivamente al mantenimento dello spazio web ospitante.
Le inserzioni pubblicitarie, se presenti, possono utilizzare una tecnologia atta a riconoscere gli accessi che provengono dallo stesso computer o dallo stesso terminale, assieme a tutte le informazioni che possono essere estrapolate dall'origine dell'accesso e sulle funzionalità del computer o del terminale usato per accedere (incluso il fatto che sia disponibile o meno del software che possa essere utile a recepire la pubblicità stessa). Per conoscere il modo in cui le informazioni vengono raccolte dalla pubblicità (se c'è) e il loro utilizzo effettivo, è necessario valutare l'informativa sul trattamento dei dati personali dello spazio web che ospita l'opera.

Per tutti i riferimenti dell'opera si veda *http://informaticalibera.net* . Al momento della pubblicazione di questa edizione, i punti di distribuzione in-linea più importanti, sono presso Internet Archive (*http://www.archive.org/details/AppuntiDiInformaticaLibera*), il GARR (*http://appuntilinux.mirror.garr.it/mirrors/appuntilinux/*), ILS (*http://appunti.linux.it*) e il Pluto (*http://a2.pluto.it*).

Parte iii Dal linguaggio macchina al C 9
62 Algoritmi e notazioni 15
63 Linguaggio macchina 97
64 Microprocessori x86-32 179
65 Compilazione e formato binario eseguibile 407
66 Introduzione al linguaggio C 557
67 Gestione dei flussi di file in C 821
Esempi di programmazione in C 873
68 Introduzione alle estensioni POSIX 909
69 Libreria C, con qualche estensione POSIX 1055
70 Libreria POSIX 1307
Tabelle riepilogative della libreria C e POSIX 1353
71 Gettext 1449
Parte iv COBOL 1465
72 Manuale COBOL 1467
73 Programmare in COBOL 1729
Parte v Basi di dati 1859
74 DBMS e SQL 1863
75 PostgreSQL 1949
76 MySQL 2067
77 SQLite 2103
78 ODBC 2121

79 SQL: lezioni pratiche e verifiche 2131

Parte vi Corso basilare di programmazione 2265

Introduzione 2267

80 Dai sistemi di numerazione all'organizzazione della memoria 2279

81 Nozioni minime sul linguaggio C 2373

82 Puntatori, array e stringhe in C 2475

Indice analitico del volume 2541

Parte iii

Dal linguaggio macchina al C

«

62 Algoritmi e notazioni 15

62.1 Notazione BNF 17
62.2 Pseudocodifica 20
62.3 Problemi elementari di programmazione 22
62.4 Scansione di array 34
62.5 Problemi classici di programmazione 37
62.6 Gestione dei file 53
62.7 Trasformazione in lettere 66
62.8 Algoritmi elementari con la shell POSIX 79
62.9 Riferimenti 94

63 Linguaggio macchina 97

63.1 Organizzazione della memoria 99
63.2 Architettura, linguaggio, contesto virtuale, terminologia 124
63.3 Rappresentazione di valori numerici 133
63.4 Calcoli con i valori binari rappresentati nella forma usata negli elaboratori 147
63.5 Scorrimenti, rotazioni, operazioni logiche 163
63.6 Confronti attraverso la sottrazione 174
63.7 Riferimenti 177

64 Microprocessori x86-32 179

64.1 Terminologia impropria 183
64.2 Registri principali fino ai 32 bit 183
64.3 Sintesi delle istruzioni principali 186
64.4 Primo approccio al linguaggio assemblatore per x86 213
64.5 Esempi con le «quattro operazioni» 244
64.6 Esempi con gli «spostamenti» 275
64.7 Esempi con i confronti 290
64.8 Le istruzioni di salto 298
64.9 Esempi di programmi con strutture di controllo 305
64.10 Funzioni 322
64.11 Esempi di funzioni ricorsive 350
64.12 Indirizzamento dei dati 358
64.13 Rappresentazione dei dati in memoria attraverso un esempio 368
64.14 Esempi con gli array 375
64.15 Calcoli con gli indirizzi in fase di compilazione . . . 391
64.16 Interazione con il sistema operativo 395
64.17 Riferimenti 405

65 Compilazione e formato binario eseguibile 407

65.1 Compilazione di programmi composti da più file sorgenti 410
65.2 Librerie dinamiche e librerie statiche 419
65.3 Dal sorgente all'immagine in memoria 432

65.4 Formato ELF 453
65.5 Programmi completamente autonomi 470
65.6 Compilazione C dal basso in alto 495
65.7 Compilazione C dall'alto in basso 509
65.8 Compilazione guidata con Make 523
65.9 Riferimenti 552

66 Introduzione al linguaggio C 557

66.1 Nozioni minime 563
66.2 Istruzioni del precompilatore 619
66.3 Dal campo di azione alla compilazione 650
66.4 Annotazioni sulla terminologia 670
66.5 Puntatori, array, stringhe e allocazione dinamica della memoria 677
66.6 Le funzioni 743
66.7 Struttura, unione, campo, enumerazione, costante composta 755
66.8 Tipi di dati speciali, di uso comune 777
66.9 Configurazione locale 789
66.10 Organizzazione dei file sorgenti 800
66.11 K&R 808
66.12 Riferimenti 816

67 Gestione dei flussi di file in C 821

67.1 Concetti generali 822
67.2 Utilizzo comune dei file 832

67.3 Conversione di input e output 850
67.4 Riferimenti 871

Esempi di programmazione in C 873

Problemi elementari 873
Scansione di array 890
Algoritmi tradizionali 896

68 Introduzione alle estensioni POSIX 909

68.1 Dal C a POSIX 912
68.2 Espressioni regolari POSIX 916
68.3 Avvio e conclusione dei processi 940
68.4 Nozioni sui thread POSIX 953
68.5 I file secondo i sistemi POSIX 979
68.6 Il file system Unix e la sua gestione tipica 996
68.7 Il file system Minix 1 1008
68.8 Creazione ed eliminazione di file di qualunque tipo 1022
68.9 Condotti 1034
68.10 Lettura delle directory 1047
68.11 Riferimenti 1051

69 Libreria C, con qualche estensione POSIX 1055

69.1 Funzionalità di libreria non dichiarate 1069
69.2 File «assert.h» 1069
69.3 File «limits.h» 1071
69.4 File «stdint.h» 1077
69.5 File «errno.h» 1087

69.6 File «locale.h» 1092
69.7 File «ctype.h» 1101
69.8 File «stdarg.h» 1123
69.9 File «stdlib.h» 1127
69.10 File «inttypes.h» 1156
69.11 File «iso646.h» 1167
69.12 File «stdbool.h» 1168
69.13 File «stddef.h» 1169
69.14 File «string.h» 1170
69.15 File «signal.h» 1223
69.16 File «time.h» 1239
69.17 File «stdio.h» 1254
69.18 Riferimenti 1302

70 Libreria POSIX 1307

70.1 File «sys/types.h» 1312
70.2 File «sys/stat.h» 1315
70.3 File «strings.h» 1321
70.4 File «fcntl.h» 1322
70.5 File «unistd.h» 1329
70.6 File «dirent.h» 1342
70.7 File «termios.h» 1345
70.8 Riferimenti 1350

Tabelle riepilogative della libreria C e POSIX 1353

File «stdarg.h» 1359

File «limits.h» 1360
File «stdint.h» 1361
File «inttypes.h» 1365
File «ctype.h» 1375
File «stdlib.h» 1379
File «string.h» 1390
File «time.h» 1399
File «stdio.h» per la gestione dei file e degli errori 1406
File «stdio.h» per la composizione dell'output 1421
File «stdio.h» per l'interpretazione dell'input . . . 1428
File «assert.h» 1435
File «stddef.h» 1435
File «locale.h» 1436
File «regex.h» 1437
File «sys/stat.h» 1442

71 Gettext 1449

71.1 Principio di funzionamento 1449
71.2 Fasi di preparazione 1450
71.3 Abbinamento a un «pacchetto» 1454
71.4 Creazione e mantenimento dei file PO 1455
71.5 Gettext con i programmi Perl 1457

Capitolo 62

Algoritmi e notazioni

62.1 Notazione BNF 17
62.1.1 BNF essenziale 17
62.1.2 Estensioni usuali 18
62.2 Pseudocodifica 20
62.3 Problemi elementari di programmazione 22
62.3.1 Somma tra due numeri positivi 22
62.3.2 Moltiplicazione di due numeri positivi attraverso la somma 24
62.3.3 Divisione intera tra due numeri positivi 25
62.3.4 Elevamento a potenza 26
62.3.5 Radice quadrata 28
62.3.6 Fattoriale 29
62.3.7 Massimo comune divisore 30
62.3.8 Numero primo 31
62.3.9 Successione di Fibonacci 32
62.4 Scansione di array 34
62.4.1 Ricerca sequenziale 34
62.4.2 Ricerca binaria 36
62.5 Problemi classici di programmazione 37
62.5.1 Bubblesort 37
62.5.2 Fusione tra due array ordinati 39
62.5.3 Torre di Hanoi 41

62.5.4 Quicksort (ordinamento non decrescente) 44
62.5.5 Permutazioni 51
62.6 Gestione dei file 53
62.6.1 Fusione tra due file ordinati 53
62.6.2 Riordino attraverso la fusione 56
62.7 Trasformazione in lettere 66
62.7.1 Da numero a sequenza alfabetica pura 66
62.7.2 Da numero a numero romano 69
62.7.3 Da numero a lettere, nel senso verbale 75
62.8 Algoritmi elementari con la shell POSIX 79
62.8.1 ARCS0: ricerca del valore più grande tra tre numeri interi 80
62.8.2 ARCS1: moltiplicazione di due numeri interi 81
62.8.3 ARCS2: valore assoluto della differenza tra due valori 82
62.8.4 ARCS3: somma tra due numeri 83
62.8.5 ARCS4: prodotto tra due numeri 83
62.8.6 ARCS5: quoziente 85
62.8.7 ARCS6: verifica della parità di un numero 86
62.8.8 ARCS7: fattoriale 87
62.8.9 ARCS8: coefficiente binomiale 87
62.8.10 ARCS10: massimo comune divisore 89
62.8.11 ARCS11: massimo comune divisore 90
62.8.12 ARCS12: radice quadrata intera 91

62.8.13 ARCS13: numero primo 92
62.8.14 ARCS14: numero primo 93
62.9 Riferimenti 94

62.1 Notazione BNF

In molti documenti si usa la «notazione BNF» per mostrare la sintassi di qualcosa, particolarmente quando si tratta della descrizione formale dei linguaggi di programmazione. La sigla BNF sta per *Backus Naur form*, a ricordare che si tratta di una notazione introdotta da John Backus e Peter Naur, tra il 1959 e il 1960.

62.1.1 BNF essenziale

La notazione BNF utilizza pochi simboli per attribuire un significato a ciò che descrive:

Simbolo	Significato
::=	Si legge come: «è definito da». Sta a indicare che l'oggetto alla sinistra di tale simbolo viene definito come ciò che si trova alla destra di questo.
|	Si legge come: «oppure». Sta a indicare che può essere usato l'oggetto che sta a sinistra del simbolo, oppure quello a destra, indifferentemente.
<***nome***>	Indica il nome di una categoria. Il nome può essere scritto senza vincoli particolari.
x	Qualunque cosa sia scritta al di fuori delle parentesi angolari ('**<**', '**>**'), escludendo altri simboli di cui sia stato dichiarato il significato, va interpretata letteralmente (come parola chiave).

A titolo di esempio, viene mostrata la definizione di una lettera dell'alfabeto latino, suddividendo il problema, definendo cosa sono le lettere latine minuscole e cosa sono le lettere latine maiuscole:

```
<lettera_alfabeto_latino> ::=
        <lettera_alfabeto_latino_maiuscola>
      | <lettera_alfabeto_latino_minuscola>
```

```
<lettera_alfabeto_latino_maiuscola> ::=
        A | B | C | D | E | F | G | H | I | J | K | L
      | M | N | O | P | Q | R | S | T | U | V | W | X
      | Y | Z
```

```
<lettera_alfabeto_latino_minuscola> ::=
        a | b | c | d | e | f | g | h | i | j | k | l
      | m | n | o | p | q | r | s | t | u | v | w | x
      | y | z
```

62.1.2 Estensioni usuali

«

Di fatto, la notazione BNF viene usata estendendo la simbologia, per semplificarne la lettura ed evitare ambiguità, ma a volte la simbologia viene anche cambiata leggermente.

Simbolo	Significato
`[...]`	Le parentesi quadre vengono usate normalmente per delimitare una porzione facoltativa della dichiarazione.

Simbolo	Significato
{...}	Le parentesi graffe vengono usate normalmente per delimitare una porzione necessaria della dichiarazione, che però, per qualche ragione, va intesa come un blocco unito. A volte, l'inclusione tra parentesi graffe va intesa come la possibilità di ripetere indefinitamente il suo contenuto, ma per questo, di solito si aggiungono tre puntini di sospensione in coda.
. . .	Tre puntini di sospensione vengono usati per indicare una porzione della dichiarazione che può essere ripetuta.
"..." '...'	Una coppia di apici doppi o singoli, può essere usata per delimitare qualcosa che va inteso letteralmente e non va confuso con la simbologia usata per la notazione BNF.
nome	I nomi di qualcosa potrebbero essere annotati senza le parentesi angolari, se si usa una forma tipografica particolare per evidenziarli (per esempio in corsivo o in nero, rispetto a un testo normale per ciò che va inteso letteralmente).

Segue un esempio molto semplice, dove si vede l'uso delle parentesi quadre, graffe e dei puntini di sospensione, per descrivere un'istruzione condizionale di un certo linguaggio, senza però entrare troppo nel dettaglio:

```
<istruzione_condizionale> ::=
    IF <espressione_logica>
        THEN
            <sequenza_di_istruzioni>
      [ ELSE
            <sequenza_di_istruzioni> ]
    END FI

<sequenza_di_istruzioni> ::=
    { <istruzione> | <commento> | <riga_bianca> }…
```

Segue lo stesso esempio, modificato in modo da evidenziare i nomi di categoria, evitando così l'uso delle parentesi angolari:

istruzione_condizionale `::= IF` ***espressione_logica***
`THEN`
sequenza_di_istruzioni
`[ ELSE`
sequenza_di_istruzioni `]`
`END FI`

sequenza_di_istruzioni `::= {` ***istruzione*** `|` ***commento*** `|` ***riga_bianca*** `}…`

62.2 Pseudocodifica

«

Un tempo la programmazione avveniva attraverso lunghe fasi di studio a tavolino. Prima di iniziare il lavoro di scrittura del programma (su moduli cartacei che venivano trasferiti successivamente nella

macchina) si passava per la realizzazione di un diagramma di flusso, o *flow chart*.

Il diagramma di flusso andava bene fino a quando si utilizzavano linguaggi di programmazione procedurali, come il COBOL. Quando si sono introdotti concetti nuovi che rendevano tale sistema di rappresentazione più complicato del linguaggio stesso, si è preferito schematizzare gli algoritmi attraverso righe di codice vero e proprio o attraverso una pseudocodifica più o meno adatta al concetto che si vuole rappresentare di volta in volta.

Nelle sezioni successive appaiono esempi realizzati attraverso una pseudocodifica. Tali esempi non sono ottimizzati perché si intende puntare sulla chiarezza piuttosto che sull'eventuale velocità di esecuzione. La pseudocodifica si rifà a termini e concetti comuni a molti linguaggi di programmazione recenti. Vale la pena di chiarire solo alcuni dettagli:

- le variabili di scambio di una subroutine (una procedura o una funzione) vengono semplicemente nominate a fianco del nome della procedura, tra parentesi, cosa che corrisponde a una dichiarazione implicita di quelle variabili con un campo di azione locale e con caratteristiche identiche a quelle usate nelle chiamate relative;
- il trasferimento dei parametri di una chiamata alla subroutine avviene per valore, impedendo l'alterazione delle variabili originali;
- per trasferire una variabile per riferimento, in modo che il suo valore venga aggiornato al termine dell'esecuzione di una sub-

routine, occorre aggiungere il simbolo '`@`' di fronte al nome della variabile utilizzata nella chiamata;

- il simbolo '`#`' rappresenta l'inizio di un commento;
- il simbolo '`:=`' rappresenta l'assegnamento;
- il simbolo '`:==:`' rappresenta lo scambio tra due operandi.

La pseudocodifica in questione non gestisce i puntatori e l'uso dell'operatore «@» è solo un modo per affermare che le modifiche apportate alla variabile devono essere mantenute alla conclusione della funzione.

62.3 Problemi elementari di programmazione

«

Nelle sezioni seguenti sono descritti alcuni problemi elementari attraverso cui si insegnano le tecniche di programmazione ai principianti. Assieme ai problemi vengono proposte le soluzioni in forma di pseudocodifica.

62.3.1 Somma tra due numeri positivi

«

La somma di due numeri positivi può essere espressa attraverso il concetto dell'incremento unitario: $n+m$ equivale a incrementare m, di un'unità, per n volte, oppure incrementare n per m volte. L'algoritmo risolutivo è banale, ma utile per apprendere il funzionamento dei cicli:

```
SOMMA (X, Y)

    LOCAL Z INTEGER
    LOCAL I INTEGER

    Z := X
    FOR I := 1; I <= Y; I++
        Z++
    END FOR

    RETURN Z

END SOMMA
```

In questo caso viene mostrata una soluzione per mezzo di un ciclo enumerativo, '**FOR**'. Il ciclo viene ripetuto '**Y**' volte, incrementando la variabile '**Z**' di un'unità. Alla fine, '**Z**' contiene il risultato della somma di '**X**' per '**Y**'. La pseudocodifica seguente mostra invece la traduzione del ciclo '**FOR**' in un ciclo '**WHILE**':

```
SOMMA (X, Y)

    LOCAL Z INTEGER
    LOCAL I INTEGER

    Z := X
    I := 1
    WHILE I <= Y
        Z++
        I++
    END WHILE

    RETURN Z

END SOMMA
```

62.3.2 Moltiplicazione di due numeri positivi attraverso la somma

«

La moltiplicazione di due numeri positivi, può essere espressa attraverso il concetto della somma: ***n**m*** equivale a sommare ***m*** volte ***n***, oppure ***n*** volte ***m***. L'algoritmo risolutivo è banale, ma utile per apprendere il funzionamento dei cicli:

```
MOLTIPLICA (X, Y)

    LOCAL Z INTEGER
    LOCAL I INTEGER

    Z := 0
    FOR I := 1; I <= Y; I++
        Z := Z + X
    END FOR

    RETURN Z

END MOLTIPLICA
```

In questo caso viene mostrata una soluzione per mezzo di un ciclo '**FOR**'. Il ciclo viene ripetuto '**Y**' volte, incrementando la variabile '**Z**' del valore di '**X**'. Alla fine, '**Z**' contiene il risultato del prodotto di '**X**' per '**Y**'. La pseudocodifica seguente mostra invece la traduzione del ciclo '**FOR**' in un ciclo '**WHILE**':

```
MOLTIPLICA (X, Y)

    LOCAL Z INTEGER
    LOCAL I INTEGER

    Z := 0
    I := 1
    WHILE I <= Y
        Z := Z + X
        I++
    END WHILE

    RETURN Z

END MOLTIPLICA
```

62.3.3 Divisione intera tra due numeri positivi

La divisione di due numeri positivi, può essere espressa attraverso la sottrazione: $n:m$ equivale a sottrarre m da n fino a quando n diventa inferiore di m. Il numero di volte in cui tale sottrazione ha luogo, è il risultato della divisione.

```
DIVIDI (X, Y)

    LOCAL Z INTEGER
    LOCAL I INTEGER

    Z := 0
    I := X
    WHILE I >= Y
        I := I - Y
        Z++
    END WHILE

    RETURN Z

END DIVIDI
```

62.3.4 Elevamento a potenza

«

L'elevamento a potenza, utilizzando numeri positivi, può essere espresso attraverso il concetto della moltiplicazione: ***n*****m*** equivale a moltiplicare ***m*** volte ***n*** per se stesso.

```
EXP (X, Y)

    LOCAL Z INTEGER
    LOCAL I INTEGER

    Z := 1
    FOR I := 1; I <= Y; I++
        Z := Z * X
    END FOR

    RETURN Z

END EXP
```

In questo caso viene mostrata una soluzione per mezzo di un ciclo '**FOR**'. Il ciclo viene ripetuto '**Y**' volte; ogni volta la variabile '**Z**' viene moltiplicata per il valore di '**X**', a partire da uno. Alla fine, '**Z**' contiene il risultato dell'elevamento di '**X**' a '**Y**'. La pseudocodifica seguente mostra invece la traduzione del ciclo '**FOR**' in un ciclo '**WHILE**':

```
EXP (X, Y)

    LOCAL Z INTEGER
    LOCAL I INTEGER

    Z := 1
    I := 1
    WHILE I <= Y
        Z := Z * X
        I++
    END WHILE

    RETURN Z

END EXP
```

La pseudocodifica seguente mostra una soluzione ricorsiva:

```
EXP (X, Y)

    IF X = 0
        THEN
            RETURN 0
        ELSE
            IF Y = 0
                THEN
                    RETURN 1
                ELSE
                    RETURN X * EXP (X, Y-1)
            END IF
    END IF

END EXP
```

62.3.5 Radice quadrata

«

Il calcolo della parte intera della radice quadrata di un numero si può fare per tentativi, partendo da 1, eseguendo il quadrato fino a quando il risultato è minore o uguale al valore di partenza di cui si calcola la radice.

```
RADICE (X)

    LOCAL Z INTEGER
    LOCAL T INTEGER

    Z := 0
    T := 0

    WHILE TRUE

        T := Z * Z
```

```
        IF T > X
            THEN
                # È stato superato il valore massimo.
                Z--
                RETURN Z
        END IF

        Z++

    END WHILE

END RADICE
```

62.3.6 Fattoriale

Il fattoriale è un valore che si calcola a partire da un numero positivo. Può essere espresso come il prodotto di n per il fattoriale di $n-1$, quando n è maggiore di 1, mentre equivale a 1 quando n è uguale a 1. In pratica, $n! = n * (n-1) * (n-2)... * 1$.

```
FATTORIALE (X)

    LOCAL I INTEGER

    I := X - 1

    WHILE I > 0
        X := X * I
        I--
    END WHILE

    RETURN X

END FATTORIALE
```

La soluzione appena mostrata fa uso di un ciclo '**WHILE**' in cui l'indice '**I**', che inizialmente contiene il valore di '**X-1**', viene usato per essere moltiplicato al valore di '**X**', riducendolo ogni volta di un'unità. Quando '**I**' raggiunge lo zero, il ciclo termina e '**X**' contiene il valore del fattoriale. L'esempio seguente mostra invece una soluzione ricorsiva che dovrebbe risultare più intuitiva:

```
FATTORIALE (X)

    IF X == 1
        THEN
            RETURN 1
        ELSE
            RETURN X * FATTORIALE (X - 1)
    END IF

END FATTORIALE
```

62.3.7 Massimo comune divisore

«

Il massimo comune divisore tra due numeri può essere ottenuto sottraendo a quello maggiore il valore di quello minore, fino a quando i due valori sono uguali. Quel valore è il massimo comune divisore.

```
MCD (X, Y)

    WHILE X != Y

        IF X > Y
            THEN
                X := X - Y
            ELSE
                Y := Y - X
        END IF

    END WHILE

    RETURN X

END MCD
```

62.3.8 Numero primo

Un numero intero è numero primo quando non può essere diviso per un altro intero diverso dal numero stesso e da 1, generando un risultato intero.

```
PRIMO (X)

    LOCAL PRIMO BOOLEAN
    LOCAL I INTEGER
    LOCAL J INTEGER

    PRIMO := TRUE
    I := 2

    WHILE (I < X) AND PRIMO

        J := X / I
        J := X - (J * I)
```

```
        IF J == 0
            THEN
                PRIMO := FALSE
            ELSE
                I++
        END IF

    END WHILE

    RETURN PRIMO

END PRIMO
```

62.3.9 Successione di Fibonacci

«

La successione di Fibonacci è una sequenza di numeri interi positivi che hanno la proprietà di essere costituiti dalla somma dei due numeri precedenti nella sequenza stessa. Pertanto, l'n-esimo elemento di questa successione, indicato solitamente come F_n, è dato dalla somma di F_{n-1} e F_{n-2}.

La successione di Fibonacci parte storicamente dal presupposto che F_1 e F_2 siano entrambi pari a uno, ma attualmente si indica anche F_0 pari a zero, cosa che consente di calcolare correttamente F_2.

Per il calcolo della successione di Fibonacci, dall'elemento zero, fino all'elemento n-esimo, vengono proposte due modalità di calcolo, la prima in forma ricorsiva, la seconda in forma iterativa.

```
FIBONACCI (N)
    IF N == 0
    THEN
        RETURN 0
    ELSE
        IF N == 1
        THEN
            RETURN 1
        ELSE
            RETURN (FIBONACCI (N - 1) + FIBONACCI (N - 2))
        END IF
    END IF
END FIBONACCI
```

```
FIBONACCI (N)
    LOCAL F1 := 1
    LOCAL F0 := 0
    LOCAL FN := N
    LOCAL I

    FOR I := 2; I <= N; I++
        FN := F1 + F0
        F0 := F1
        F1 := FN
    END FOR

    RETURN FN

END FIBONACCI
```

La successione di Fibonacci, per cui F_0 è pari a zero e F_1 è pari a uno, è: 0, 1, 1, 2, 3, 5, 8, 13,...

62.4 Scansione di array

«

Nelle sezioni seguenti sono descritti alcuni problemi legati alla scansione di array. Assieme ai problemi vengono proposte le soluzioni in forma di pseudocodifica.

62.4.1 Ricerca sequenziale

«

La ricerca di un elemento all'interno di un array disordinato può avvenire solo in modo sequenziale, cioè controllando uno per uno tutti gli elementi, fino a quando si trova la corrispondenza cercata. Segue la descrizione delle variabili più importanti che appaiono nella pseudocodifica successiva:

Variabile	Descrizione
`LISTA`	È l'array su cui effettuare la ricerca.
`X`	È il valore cercato all'interno dell'array.
`A`	È l'indice inferiore dell'intervallo di array su cui si vuole effettuare la ricerca.
`Z`	È l'indice superiore dell'intervallo di array su cui si vuole effettuare la ricerca.

Ecco un esempio di pseudocodifica che risolve il problema in modo iterativo:

```
RICERCASEQ (LISTA, X, A, Z)

    LOCAL I INTEGER

    FOR I := A; I <= Z; I++
        IF X == LISTA[I]
            THEN
                RETURN I
        END IF
    END FOR

    # La corrispondenza non è stata trovata.
    RETURN -1

END RICERCASEQ
```

Solo a scopo didattico, viene proposta una soluzione ricorsiva:

```
RICERCASEQ (LISTA, X, A, Z)

    IF A > Z
        THEN
            RETURN -1
        ELSE
            IF X == LISTA[A]
                THEN
                    RETURN A
                ELSE
                    RETURN RICERCASEQ (@LISTA, X, A+1, Z)
            END IF
    END IF

END RICERCASEQ
```

62.4.2 Ricerca binaria

«

La ricerca di un elemento all'interno di un array ordinato può avvenire individuando un elemento centrale: se questo corrisponde all'elemento cercato, la ricerca è terminata, altrimenti si ripete nella parte di array precedente o successiva all'elemento, a seconda del suo valore e del tipo di ordinamento esistente.

Il problema posto in questi termini è ricorsivo. La pseudocodifica mostrata utilizza le stesse variabili già descritte per la ricerca sequenziale.

```
RICERCABIN (LISTA, X, A, Z)

  LOCAL M INTEGER

  # Determina l'elemento centrale dell'array.
  M := (A + Z) / 2

  IF M < A
    THEN
      # Non restano elementi da controllare:
      # l'elemento cercato non c'è.
      RETURN -1
    ELSE
      IF X < LISTA[M]
        THEN
          # Si ripete la ricerca nella parte
          # inferiore.
          RETURN RICERCABIN (@LISTA, X, A, M-1)
        ELSE
          IF X > LISTA[M]
            THEN
              # Si ripete la ricerca nella
              # parte superiore.
```

```
                RETURN RICERCABIN (@LISTA, X, M+1, Z)
              ELSE
                # M rappresenta l'indice
                # dell'elemento cercato.
                RETURN M
            END IF
        END IF
    END IF

END RICERCABIN
```

62.5 Problemi classici di programmazione

Nelle sezioni seguenti sono descritti alcuni problemi classici attraverso cui si insegnano le tecniche di programmazione. Assieme ai problemi vengono proposte le soluzioni in forma di pseudocodifica.

62.5.1 Bubblesort

Il Bubblesort è un algoritmo relativamente semplice per l'ordinamento di un array, in cui ogni scansione trova il valore giusto per l'elemento iniziale dell'array stesso. Una volta trovata la collocazione di un elemento, si ripete la scansione per il segmento rimanente di array, in modo da collocare un altro valore. La pseudocodifica dovrebbe chiarire il meccanismo.

Variabile	Descrizione
`LISTA`	È l'array da ordinare.
`A`	È l'indice inferiore del segmento di array da ordinare.
`Z`	È l'indice superiore del segmento di array da ordinare.

Viene mostrata una soluzione iterativa:

```
BSORT (LISTA, A, Z)

    LOCAL J INTEGER
    LOCAL K INTEGER

    # Scandisce l'array attraverso l'indice J in modo da
    # collocare ogni volta il valore corretto all'inizio
    # dell'array stesso.
    FOR J := A; J < Z; J++

        # Scandisce l'array attraverso l'indice K scambiando
        # i valori quando sono inferiori a quello di
        # riferimento.
        FOR K := J+1; K <= Z; K++

            IF LISTA[K] < LISTA[J]
                THEN
                    # I valori vengono scambiati.
                    LISTA[K] :==: LISTA[J]
            END IF

        END FOR

    END FOR

END BSORT
```

Segue una soluzione ricorsiva:

```
BSORT (LISTA, A, Z)

    LOCAL K INTEGER

    # L'elaborazione termina quando l'indice inferiore è
```

```
    # maggiore o uguale a quello superiore.
    IF A < Z
        THEN

            # Scandisce l'array attraverso l'indice K
            # scambiando i valori quando sono inferiori a
            # quello iniziale.
            FOR K := A+1; K <= Z; K++

                IF LISTA[K] < LISTA[A]
                    THEN
                        # I valori vengono scambiati.
                        LISTA[K] :==: LISTA[A]
                END IF

            END FOR

            # L'elemento LISTA[A] è collocato correttamente,
            # adesso si ripete la chiamata della funzione in
            # modo da riordinare la parte restante
            # dell'array.
            BSORT (@LISTA, A+1, Z)
    END IF

END BSORT
```

62.5.2 Fusione tra due array ordinati

Due array a una dimensione, con la stessa struttura, ordinati secondo qualche criterio, possono essere fusi in un array singolo, che mantiene l'ordinamento.

Variabile	Descrizione
`A`	È il primo array.
`B`	È il secondo array.
`C`	È l'array da generare con la fusione di '**A**' e '**B**'.
`I`	È l'indice usato per scandire '**A**'.
`J`	È l'indice usato per scandire '**B**'.
`K`	È l'indice usato per scandire '**C**'.
`N`	È la dimensione di '**A**' (l'indice dell'ultimo elemento dell'array).
`M`	È la dimensione di '**B**' (l'indice dell'ultimo elemento dell'array).

Viene mostrata una soluzione iterativa, dove si presume che gli array siano ordinati in modo non decrescente:

```
MERGE (A, N, B, M, C)

    LOCAL I INTEGER
    LOCAL J INTEGER
    LOCAL K INTEGER

    # Si presume che l'indice del primo elemento degli
    # array sia pari a uno.

    I := 1
    J := 1
    K := 1
```

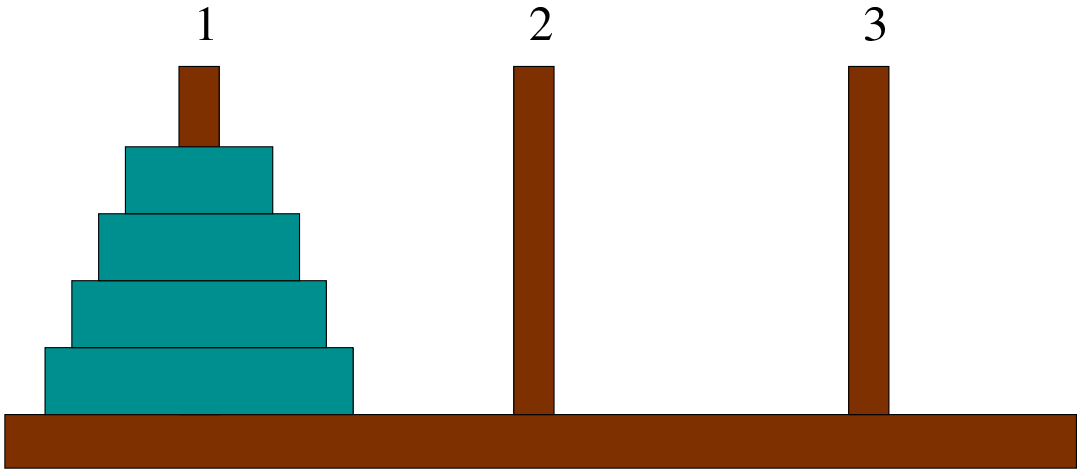
```
    UNTIL I >= N AND J >= M

        IF A(I) <= B(J)
            THEN
                C(K) := A(I)
                I++
            ELSE
                C(K) := B(J)
                J++
        END IF
        K++

    END UNTIL

END MERGE
```

62.5.3 Torre di Hanoi

La torre di Hanoi è un gioco antico: si compone di tre pioli identici conficcati verticalmente su una tavola e di una serie di anelli di larghezze differenti. Gli anelli sono più precisamente dei dischi con un foro centrale che permette loro di essere infilati nei pioli.

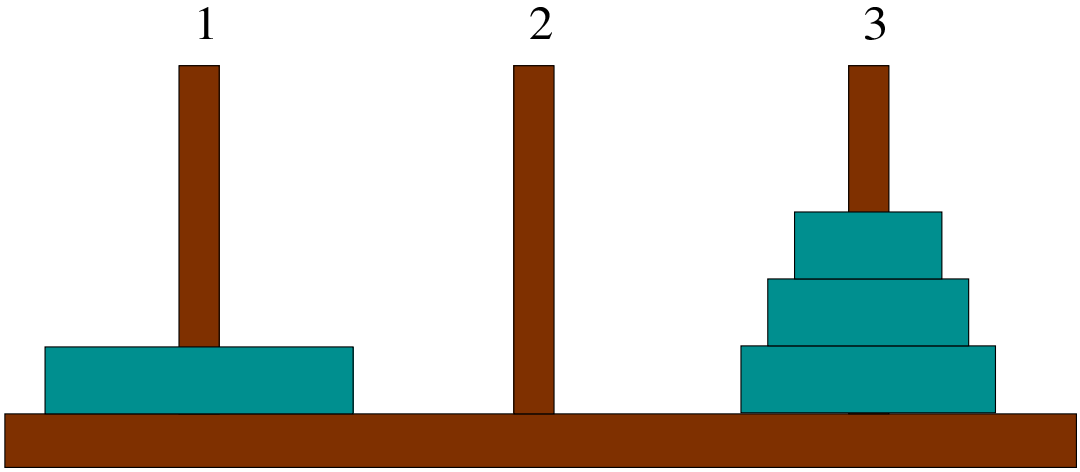
Il gioco inizia con tutti gli anelli collocati in un solo piolo, in ordine, in modo che in basso ci sia l'anello più largo e in alto quello più stretto. Si deve riuscire a spostare tutta la pila di anelli in un dato piolo muovendo un anello alla volta e senza mai collocare un anello più grande sopra uno più piccolo.

Figura 62.27. Situazione iniziale della torre di Hanoi all'inizio del gioco.



Nella figura 62.27 gli anelli appaiono inseriti sul piolo 1; si supponga che questi debbano essere spostati sul piolo 2. Si può immaginare che tutti gli anelli, meno l'ultimo, possano essere spostati in qualche modo corretto, dal piolo 1 al piolo 3, come nella situazione della figura 62.28.

Figura 62.28. Situazione dopo avere spostato $n-1$ anelli.



A questo punto si può spostare l'ultimo anello rimasto (l'n-esimo), dal piolo 1 al piolo 2; quindi, come prima, si può spostare in qualche modo il gruppo di anelli posizionati attualmente nel piolo 3, in modo che finiscano nel piolo 2 sopra l'anello più grande.

Pensando in questo modo, l'algoritmo risolutivo del problema deve essere ricorsivo e potrebbe essere gestito da un'unica subroutine che può essere chiamata opportunamente '**HANOI**'.

Variabile	Descrizione
`N`	È la dimensione della torre espressa in numero di anelli: gli anelli sono numerati da 1 a '**N**'.
`P1`	È il numero del piolo su cui si trova inizialmente la pila di '**N**' anelli.
`P2`	È il numero del piolo su cui deve essere spostata la pila di anelli.
`6-P1-P2`	È il numero dell'altro piolo. Funziona così se i pioli sono numerati da 1 a 3.

Segue la pseudocodifica ricorsiva per la soluzione del problema:

```
HANOI (N, P1, P2)

  IF N > 0
    THEN
      HANOI (N-1, P1, 6-P1-P2)
      scrivi: "Muovi l'anello" N "dal piolo" P1 "al piolo" P2
      HANOI (N-1, 6-P1-P2, P2)
  END IF
END HANOI
```

Se '**N**', il numero degli anelli da spostare, è minore di 1, non si deve compiere alcuna azione. Se '**N**' è uguale a 1, le istruzioni che dipendono dalla struttura IF-END IF vengono eseguite, ma nessuna delle chiamate ricorsive fa alcunché, dato che '**N-1**' è pari a zero. In questo caso, supponendo che '**N**' sia uguale a 1, che '**P1**' sia pari a 1 e '**P2**' pari a 2, il risultato è semplicemente:

```
Muovi l'anello 1 dal piolo 1 al piolo 2
```

Il risultato è quindi corretto per una pila iniziale consistente di un solo anello.

Se '**N**' è uguale a 2, la prima chiamata ricorsiva sposta un anello ('**N-1**' = 1) dal piolo 1 al piolo 3 (ancora assumendo che i due anelli debbano essere spostati dal primo al terzo piolo) e si sa che questa è la mossa corretta. Quindi viene stampato il messaggio che dichiara lo spostamento del secondo piolo (l'**N**'-esimo) dalla posizione 1 alla posizione 2. Infine, la seconda chiamata ricorsiva si occupa di spostare l'anello collocato precedentemente nel terzo piolo, nel secondo, sopra a quello che si trova già nella posizione finale corretta.

In pratica, nel caso di due anelli che devono essere spostati dal primo al secondo piolo, appaiono i tre messaggi seguenti.

```
Muovi l'anello 1 dal piolo 1 al piolo 3
Muovi l'anello 2 dal piolo 1 al piolo 2
Muovi l'anello 1 dal piolo 3 al piolo 2
```

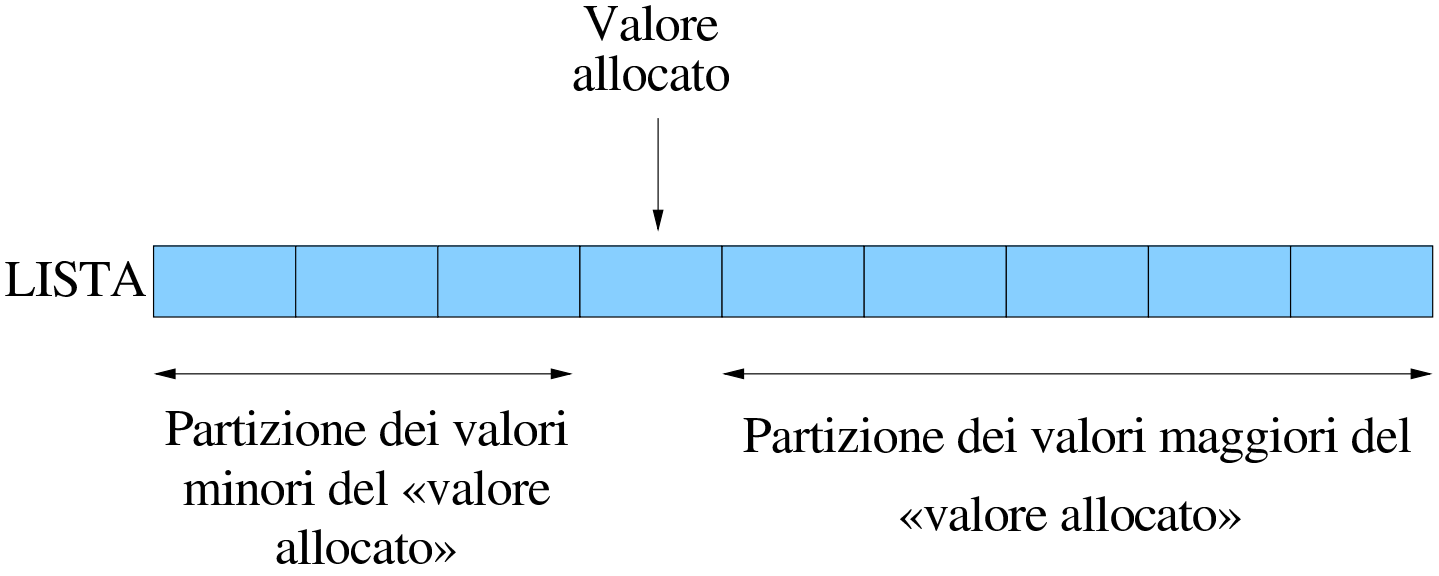
Nello stesso modo si potrebbe dimostrare il funzionamento per un numero maggiore di anelli.

62.5.4 Quicksort (ordinamento non decrescente)

«

L'ordinamento degli elementi di un array è un problema tipico che si può risolvere in tanti modi. Il Quicksort è un algoritmo sofisticato, ottimo per lo studio della gestione degli array, oltre che per quello della ricorsione. Il concetto fondamentale di questo tipo di algoritmo è rappresentato dalla figura 62.33.

Figura 62.33. Il concetto base dell'algoritmo del Quicksort: suddivisione dell'array in due gruppi disordinati, separati da un valore piazzato correttamente nel suo posto rispetto all'ordinamento.



Una sola scansione dell'array è sufficiente per collocare definitivamente un elemento (per esempio il primo) nella sua destinazione finale e allo stesso tempo per lasciare tutti gli elementi con un valore inferiore a quello da una parte, anche se disordinati, e tutti quelli con un valore maggiore, dall'altra.

In questo modo, attraverso delle chiamate ricorsive, è possibile elaborare i due segmenti dell'array rimasti da riordinare.

L'algoritmo può essere descritto grossolanamente come:

1. localizzazione della collocazione finale del primo valore, separando in questo modo i valori;
2. ordinamento del segmento precedente all'elemento collocato definitivamente;
3. ordinamento del segmento successivo all'elemento collocato

definitivamente.

Viene qui indicato con '**PART**' la subroutine che esegue la scansione dell'array, o di un suo segmento, per determinare la collocazione finale (indice '**CF**') del primo elemento (dell'array o del segmento in questione).

Sia '**LISTA**' l'array da ordinare. Il primo elemento da collocare corrisponde inizialmente a '**LISTA[A]**' e il segmento di array su cui intervenire corrisponde a '**LISTA[A:Z]**' (cioè a tutti gli elementi che vanno dall'indice '**A**' all'indice '**Z**').

Alla fine della prima scansione, l'indice '**CF**' rappresenta la posizione in cui occorre spostare il primo elemento, cioè '**LISTA[A]**'. In pratica, '**LISTA[A]**' e '**LISTA[CF]**' vengono scambiati.

Durante la scansione che serve a determinare la collocazione finale del primo elemento, '**PART**' deve occuparsi di spostare gli elementi prima o dopo quella posizione, in funzione del loro valore, in modo che alla fine quelli inferiori o uguali a quello dell'elemento da collocare si trovino nella parte inferiore e gli altri dall'altra. In pratica, alla fine della prima scansione, gli elementi contenuti in '**LISTA[A:(CF-1)]**' devono contenere valori inferiori o uguali a '**LISTA[CF]**', mentre quelli contenuti in '**LISTA[(CF+1):Z]**' devono contenere valori superiori.

Indichiamo con '**QSORT**' la subroutine che esegue il compito complessivo di ordinare l'array. Il suo lavoro consisterebbe nel chiamare '**PART**' per collocare il primo elemento, continuando poi con la chiamata ricorsiva di se stessa per la parte di array precedente all'elemento collocato e infine alla chiamata ricorsiva per la parte restante di array.

Assumendo che '**PART**' e le chiamate ricorsive di '**QSORT**' svolgano il loro compito correttamente, si potrebbe fare un'analisi informale dicendo che se l'indice '**Z**' **non** è maggiore di '**A**', allora c'è un elemento (o nessuno) all'interno di '**LISTA[A:Z]**' e inoltre, '**LISTA[A:Z]**' è già nel suo stato finale. Se '**Z**' è maggiore di '**A**', allora (per assunzione) '**PART**' ripartisce correttamente '**LISTA[A:Z]**'. L'ordinamento separato dei due segmenti (per assunzione eseguito correttamente dalle chiamate ricorsive) completa l'ordinamento di '**LISTA[A:Z]**'.

Le figure 62.34 e 62.35 mostrano due fasi della scansione effettuata da '**PART**' all'interno dell'array o del segmento che gli viene fornito.

Figura 62.34. La scansione dell'array da parte di '**PART**' avviene portando in avanti l'indice '**I**' e portando indietro l'indice '**CF**'. Quando l'indice '**I**' localizza un elemento che contiene un valore maggiore di '**LISTA[A]**' e l'indice '**CF**' localizza un elemento che contiene un valore inferiore o uguale a '**LISTA[A]**', gli elementi cui questi indici fanno riferimento vengono scambiati, quindi il processo di avvicinamento tra '**I**' e '**CF**' continua.

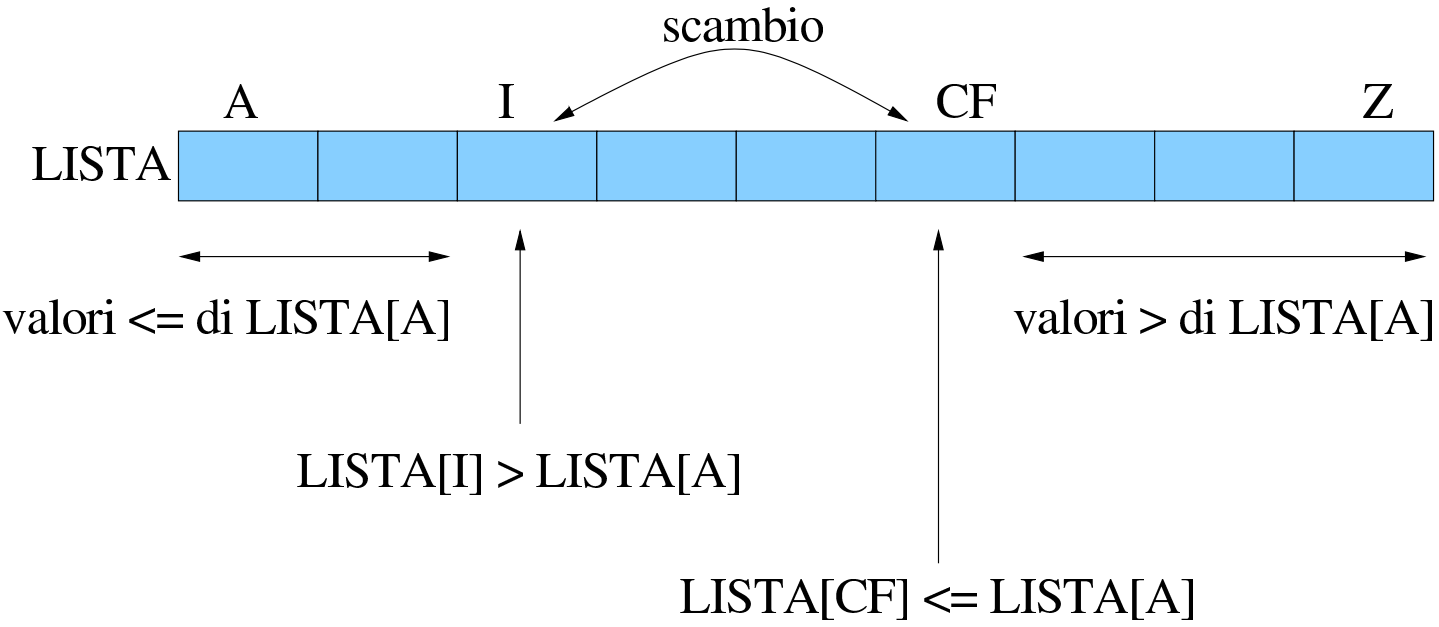
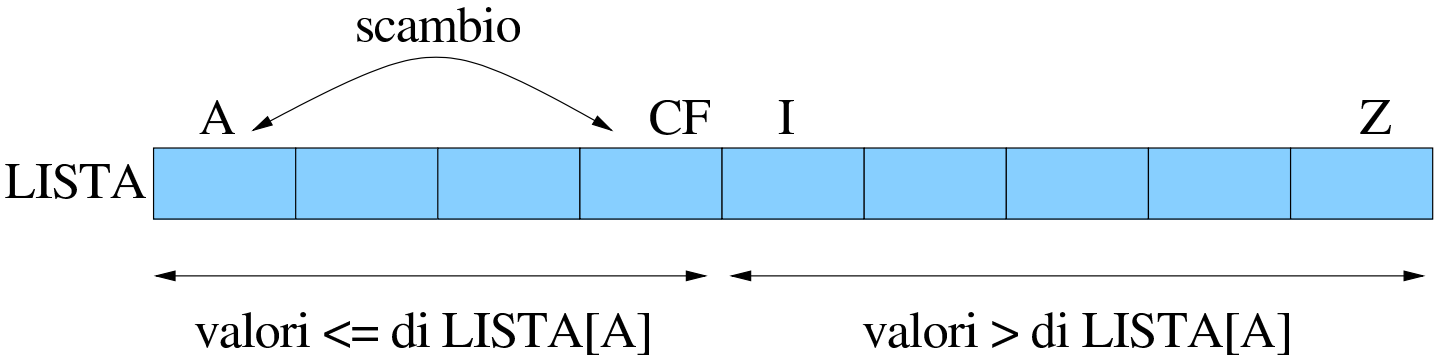


Figura 62.35. Quando la scansione è giunta al termine, quello che resta da fare è scambiare l'elemento '**LISTA[A]**' con '**LISTA[CF]**'.



In pratica, l'indice '**I**', iniziando dal valore '**A+1**', viene spostato verso destra fino a che viene trovato un elemento maggiore di '**LISTA[A]**', quindi è l'indice '**CF**' a essere spostato verso sinistra, iniziando dalla stessa posizione di '**Z**', fino a che viene incontrato un elemento minore o uguale a '**LISTA[A]**'. Questi elementi vengono scambiati e lo spostamento di '**I**' e '**CF**' riprende. Ciò prosegue fino a che '**I**' e '**CF**' si incontrano, momento in cui '**LISTA[A:Z]**' è stata ripartita e '**CF**' rappresenta l'indice di un elemento che si trova nella sua collocazione finale.

Variabile	Descrizione
`LISTA`	L'array da ordinare in modo crescente.
`A`	L'indice inferiore del segmento di array da ordinare.
`Z`	L'indice superiore del segmento di array da ordinare.
`CF`	Sta per «collocazione finale» ed è l'indice che cerca e trova la posizione giusta di un elemento nell'array.
`I`	È l'indice che insieme a '**CF**' serve a ripartire l'array.

Segue la pseudocodifica delle due subroutine:

```
PART (LISTA, A, Z)

    LOCAL I INTEGER
    LOCAL CF INTEGER

    # si assume che A < Z

    I := A + 1
    CF := Z

    WHILE TRUE # ciclo senza fine.

        WHILE TRUE

            # sposta I a destra

            IF ((LISTA[I] > LISTA[A]) OR (I >= CF))
                THEN
                    BREAK
                ELSE
                    I := I + 1
            END IF

        END WHILE

        WHILE TRUE

            # sposta CF a sinistra

            IF (LISTA[CF] <= LISTA[A])
                THEN
                    BREAK
                ELSE
                    CF := CF - 1
```

```
            END IF

        END WHILE

        IF CF <= I
            THEN
                # è avvenuto l'incontro tra I e CF
                BREAK
            ELSE
                # vengono scambiati i valori
                LISTA[CF] :==: LISTA[I]
                I := I + 1
                CF := CF - 1
        END IF

    END WHILE

    # a questo punto LISTA[A:Z] è stata ripartita e CF è la
    # collocazione di LISTA[A]

    LISTA[CF] :==: LISTA[A]

    # a questo punto, LISTA[CF] è un elemento (un valore)
    # nella giusta posizione

    RETURN CF

END PART
```

```
QSORT (LISTA, A, Z)

    LOCAL CF INTEGER

    IF Z > A
        THEN
            CF := PART (@LISTA, A, Z)
            QSORT (@LISTA, A, CF-1)
            QSORT (@LISTA, CF+1, Z)
    END IF
END QSORT
```

Vale la pena di osservare che l'array viene indicato nelle chiamate in modo che alla subroutine sia inviato un riferimento a quello originale, perché le variazioni fatte all'interno delle subroutine devono riflettersi sull'array originale.

62.5.5 Permutazioni

La permutazione è lo scambio di un gruppo di elementi posti in sequenza. Il problema che si vuole analizzare è la ricerca di tutte le permutazioni possibili di un dato gruppo di elementi.

Se ci sono n elementi in un array, allora alcune delle permutazioni si possono ottenere bloccando l'n-esimo elemento e generando tutte le permutazioni dei primi $n-1$ elementi. Quindi l'n-esimo elemento può essere scambiato con uno dei primi $n-1$, ripetendo poi la fase precedente. Questa operazione deve essere ripetuta finché ognuno degli n elementi originali è stato usato nell'n-esima posizione.

Variabile	Descrizione
`LISTA`	L'array da permutare.
`A`	L'indice inferiore del segmento di array da permutare.
`Z`	L'indice superiore del segmento di array da permutare.
`K`	È l'indice che serve a scambiare gli elementi.

Segue la pseudocodifica:

```
PERMUTA (LISTA, A, Z)

    LOCAL K INTEGER
    LOCAL N INTEGER

    IF (Z - A) >= 1
        # Ci sono almeno due elementi nel segmento di array.
        THEN
            FOR K := Z; K >= A; K--

                LISTA[K] :==: LISTA[Z]

                PERMUTA (LISTA, A, Z-1)

                LISTA[K] :==: LISTA[Z]

            END FOR
        ELSE
            scrivi LISTA
    END IF

END PERMUTA
```

62.6 Gestione dei file

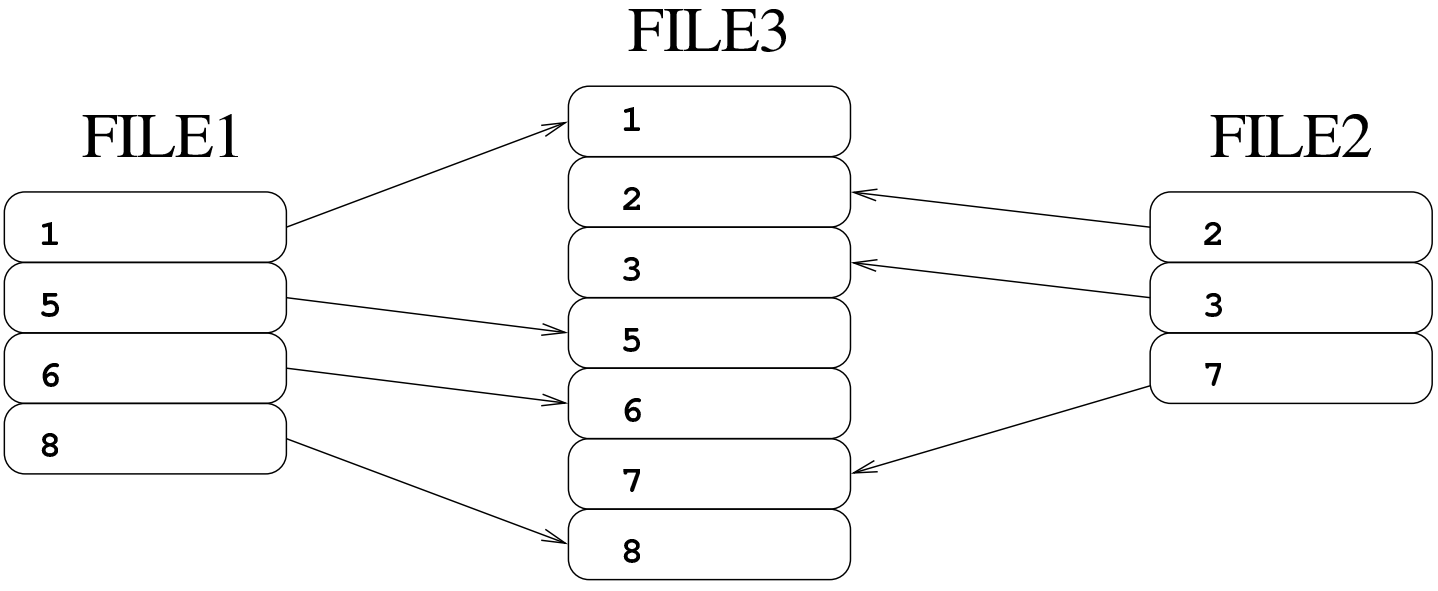
La gestione dei file è sempre la questione più complessa nello studio degli algoritmi, quanto si esclude la possibilità di gestire semplicemente tutto nella memoria centrale. In questo tipo di situazione, il file deve essere inteso come un insieme di record, che spesso sono di dimensione uniforme.

Nelle sezioni seguenti sono descritti alcuni problemi legati alla gestione dei file, con la soluzione in forma di pseudocodifica.

62.6.1 Fusione tra due file ordinati

La fusione di due file ordinati, aventi la stessa struttura, avviene leggendo un record da entrambi i file, confrontando la chiave di ordinamento e scrivendo nel file da ottenere il record con chiave più bassa. Successivamente, viene letto solo record successivo del file che conteneva la chiave più bassa e si ripete il confronto.

Figura 62.41. Il file '`FILE1`' e '`FILE2`' vengono fusi per generare il file '`FILE3`'.



Segue un esempio di pseudocodifica per la soluzione del problema della fusione tra due file. La funzione riceve il riferimento ai file da elaborare e si può osservare l'utilizzo di variabili strutturate per accogliere i record dei file da elaborare. Come si può intuire, le variabili booleane il cui nome inizia per '**EOF**', rappresentano l'avverarsi della condizione di «fine del file»; in pratica, quando contengono il valore *Vero*, indicano che la lettura del file a cui si riferiscono è andata oltre la conclusione del file.

```
FUSIONE_DUE_FILE (FILE_IN_1, FILE_IN_2, FILE_OUT)

  LOCAL RECORD_1:
        CHIAVE_1       CHARACTER (8)
        DATI_1         CHARACTER (72)
  LOCAL EOF_1 := FALSE

  LOCAL RECORD_2:
        CHIAVE_2       CHARACTER (8)
        DATI_2         CHARACTER (72)
  LOCAL EOF_2 := FALSE

  LOCAL RECORD_3       CHARACTER (80)

  OPEN INPUT  FILE_IN_1
  OPEN INPUT  FILE_IN_2
  OPEN OUTPUT FILE_OUT

  READ FILE_IN_1 NEXT RECORD INTO RECORD_1
  IF END OF FILE
    THEN
      EOF_1 := TRUE
  END IF

  READ FILE_IN_2 NEXT RECORD INTO RECORD_2
```

```
IF END OF FILE
  THEN
    EOF_2 := TRUE
END IF

UNTIL EOF_1 AND EOF_2

  IF EOF_1
    THEN
      RECORD_3 := RECORD_2
      READ FILE_IN_2 NEXT RECORD INTO RECORD_2
      IF END OF FILE
        THEN
          EOF_2 := TRUE
      END IF
    ELSE
      IF EOF_2
        THEN
          RECORD_3 := RECORD_1
          READ FILE_IN_1 NEXT RECORD INTO RECORD_1
          IF END OF FILE
            THEN
              EOF_1 := TRUE
          END IF
        ELSE
          IF CHIAVE_1 < CHIAVE_2
            THEN
              RECORD_3 := RECORD_1
              READ FILE_IN_1 NEXT RECORD INTO RECORD_1
              IF END OF FILE
                THEN
                  EOF_1 := TRUE
              END IF
            ELSE
```

```
                    RECORD_3 := RECORD_2
                    READ FILE_IN_2 NEXT RECORD INTO RECORD_2
                    IF END OF FILE
                      THEN
                        EOF_2 := TRUE
                    END IF
                  END IF
            END IF
      END IF

      WRITE FILE_OUT RECORD FROM RECORD_3

   END UNTIL

   CLOSE FILE_IN_1
   CLOSE FILE_IN_2
   CLOSE FILE_OUT

END FUSIONE_DUE_FILE
```

62.6.2 Riordino attraverso la fusione

«

Un file non ordinato può essere ordinato, attraverso una serie di passaggi, che prevedono la divisione in due parti del file (ovvero ***biforcazione***), contenenti le raccolte dei blocchi di record che risultano essere nella sequenza corretta, per poi fondere queste due parti e ripetere il procedimento. Le figure successive mostrano le due fasi: la separazione in due file, la fusione dei due file. Se il file risultante non è ordinato completamente, occorre procedere con una nuova fase di separazione e fusione. Si osservi, in particolare, che nelle figure, il file iniziale contiene solo tre blocchi di record in sequenza, pertanto,

l'ultimo di questi blocchi viene collocato nel file finale senza una fusione con un blocco corrispondente.

Figura 62.43. Biforcazione del file.

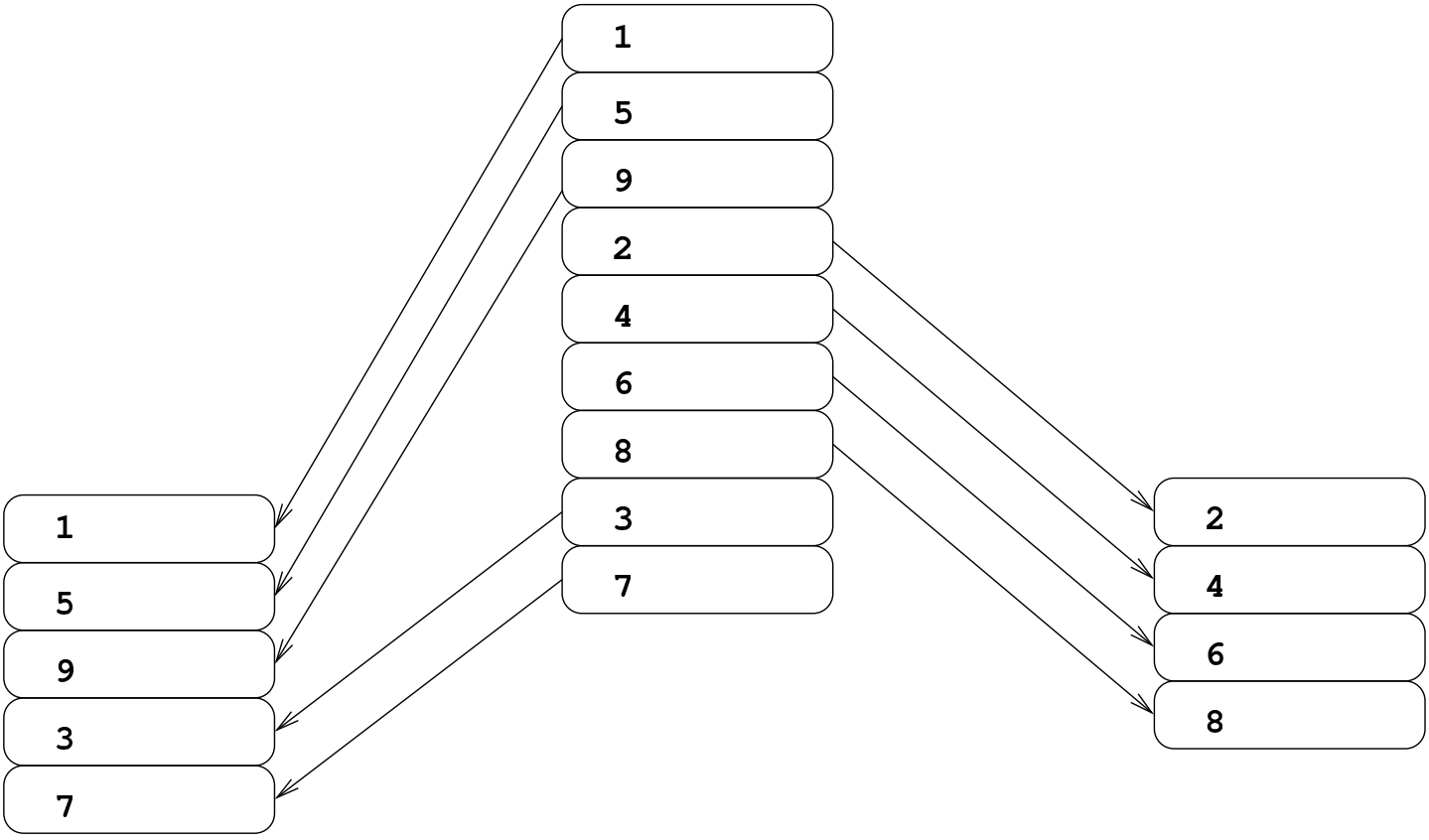
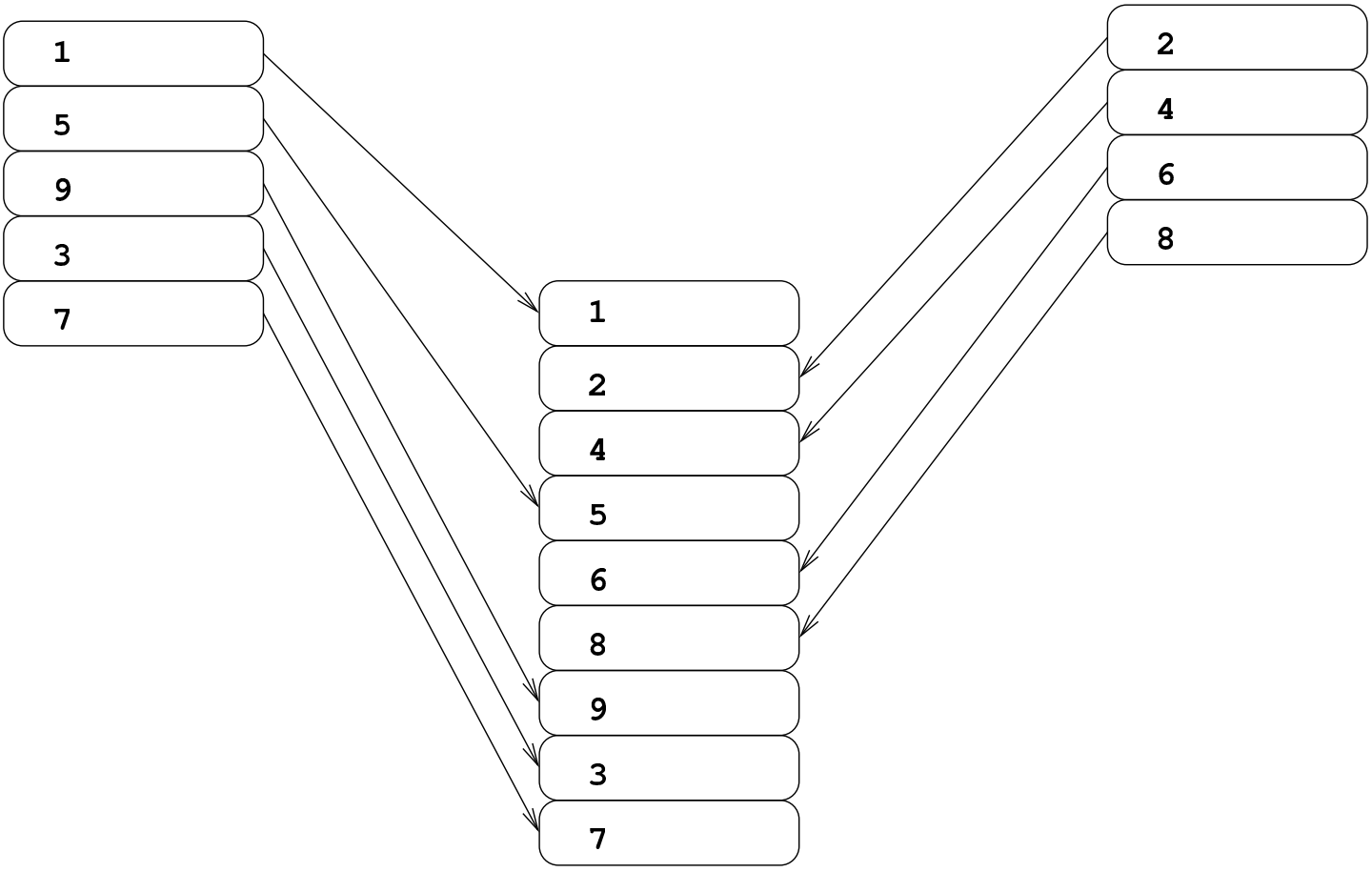


Figura 62.44. Fusione.



Per costruire un programma che utilizza questa tecnica di ordinamento, si può considerare che sia avvenuta l'ultima fase di biforcazione quando si contano un massimo di due blocchi; pertanto, la fase successiva di fusione produce sicuramente il file ordinato finale.

Segue un esempio di pseudocodifica per eseguire la biforcazione. Si osservi che i nomi dei file vengono passati in qualche modo, così che dopo la chiamata di questa procedura sia possibile riaprire tali file per la fusione; inoltre, l'informazione contenuta nella variabile ***BIFORCAZIONI***, viene restituita come valore della chiamata della funzione, in modo da poter conoscere, dopo la chiamata, quante separazioni sono state eseguite nel file di partenza.

```
BIFORCA (FILE_IN_1, FILE_OUT_1, FILE_OUT_2)
```

```
LOCAL RECORD_IN_1:
      CHIAVE         CHARACTER (8)
      DATI           CHARACTER (72)
LOCAL CHIAVE_ORIG    CHARACTER (8)
LOCAL EOF_1 := FALSE

LOCAL RECORD_OUT_1   CHARACTER (80)
LOCAL RECORD_OUT_2   CHARACTER (80)

LOCAL SCAMBIO := 1
LOCAL BIFORCAZIONI := 0

OPEN INPUT  FILE_IN_1
OPEN OUTPUT FILE_OUT_1
OPEN OUTPUT FILE_OUT_2

READ FILE_IN_1 NEXT RECORD INTO RECORD_IN_1
IF END OF FILE
  THEN
    EOF_1 := TRUE
  ELSE
    BIFORCAZIONI++
    WRITE FILE_OUT_1 RECORD FROM RECORD_IN_1
    CHIAVE_ORIG := CHIAVE
    READ FILE_IN_1 NEXT RECORD INTO RECORD_IN_1
    IF END OF FILE
      THEN
        EOF_1 := TRUE
    END IF
END IF

UNTIL EOF_1
```

```
    IF CHIAVE >= CHIAVE_ORIG
      THEN
        IF SCAMBIO == 1
          THEN
            WRITE FILE_OUT_1 RECORD FROM RECORD_IN_1
            CHIAVE_ORIG := CHIAVE
            READ FILE_IN_1 NEXT RECORD INTO RECORD_IN_1
            IF END OF FILE
              THEN
                EOF_1 := TRUE
            END IF
          ELSE
            WRITE FILE_OUT_2 RECORD FROM RECORD_IN_1
            CHIAVE_ORIG := CHIAVE
            READ FILE_IN_1 NEXT RECORD INTO RECORD_IN_1
            IF END OF FILE
              THEN
                EOF_1 := TRUE
            END IF
        END IF
      ELSE
        BIFORCAZIONI++
        CHIAVE_ORIG := CHIAVE
        IF SCAMBIO == 1
          THEN
            SCAMBIO := 2
          ELSE
            SCAMBIO := 1
        END IF
    END IF
  END UNTIL

  CLOSE FILE_IN_1
  CLOSE FILE_OUT_1
```

```
  CLOSE FILE_OUT_2

  RETURN BIFORCAZIONI

END BIFORCA
```

Segue un esempio di pseudocodifica per eseguire la fusione a blocchi. Si osservi che i nomi dei file vengono passati in qualche modo, così che dopo la chiamata di questa procedura sia possibile riaprire tali file per la biforcazione e di nuovo per la fusione. Come si può intuire, le variabili booleane il cui nome inizia per '**EOB**', rappresentano l'avverarsi della condizione di «fine del blocco» non decrescente; in pratica, quando contengono il valore *Vero*, indicano che la lettura del file a cui si riferiscono ha prodotto un record che ha una chiave inferiore rispetto a quello letto precedentemente, oppure che non sono disponibili altri record.

```
FUSIONE (FILE_IN_1, FILE_IN_2, FILE_OUT)

  LOCAL RECORD_1:
        CHIAVE_1       CHARACTER (8)
        DATI_1         CHARACTER (72)
  LOCAL CHIAVE_1_ORIG CHARACTER (8)
  LOCAL EOF_1 := FALSE
  LOCAL EOB_1 := FALSE

  LOCAL RECORD_2:
        CHIAVE_2       CHARACTER (8)
        DATI_2         CHARACTER (72)
  LOCAL CHIAVE_2_ORIG CHARACTER (8)
  LOCAL EOF_2 := FALSE
  LOCAL EOB_2 := FALSE
```

```
LOCAL RECORD_3      CHARACTER (80)

OPEN INPUT  FILE_IN_1
OPEN INPUT  FILE_IN_2
OPEN OUTPUT FILE_OUT

READ FILE_IN_1 NEXT RECORD INTO RECORD_1
IF END OF FILE
  THEN
    EOF_1 := TRUE
    EOB_1 := TRUE
  ELSE
    CHIAVE_1_ORIG := CHIAVE_1
END IF

READ FILE_IN_2 NEXT RECORD INTO RECORD_2
IF END OF FILE
  THEN
    EOF_2 := TRUE
    EOB_2 := TRUE
  ELSE
    CHIAVE_2_ORIG := CHIAVE_2
END IF

UNTIL EOF_1 AND EOF_2
  UNTIL EOB_1 AND EOB_2

    IF EOB_1
      THEN
        RECORD_3 := RECORD_2
        READ FILE_IN_2 NEXT RECORD INTO RECORD_2
        IF END OF FILE
          THEN
            EOF_2 := TRUE
```

```
            EOB_2 := TRUE
          END IF
        ELSE
          IF EOB_2
            THEN
              RECORD_3 := RECORD_1
              READ FILE_IN_1 NEXT RECORD INTO RECORD_1
              IF END OF FILE
                THEN
                  EOF_1 := TRUE
                  EOB_1 := TRUE
              END IF
            ELSE
              IF CHIAVE_1 < CHIAVE_2
                THEN
                  RECORD_3 := RECORD_1
                  READ FILE_IN_1 NEXT RECORD INTO RECORD_1
                  IF END OF FILE
                    THEN
                      EOF_1 := TRUE
                      EOB_1 := TRUE
                    ELSE
                      IF CHIAVE_1 >= CHIAVE_1_ORIG
                        THEN
                          CHIAVE_1_ORIG := CHIAVE_1
                        ELSE
                          EOB_1 := TRUE
                    END IF
                  END IF
                ELSE
                  RECORD_3 := RECORD_2
                  READ FILE_IN_2 NEXT RECORD INTO RECORD_2
                  IF END OF FILE
                    THEN
```

```
                        EOF_2 := TRUE
                        EOB_2 := TRUE
                      ELSE
                        IF CHIAVE_2 >= CHIAVE_2_ORIG
                          THEN
                            CHIAVE_2_ORIG := CHIAVE_2
                          ELSE
                            EOB_2 := TRUE
                        END IF
                    END IF
                END IF
            END IF
        END IF

        WRITE FILE_OUT RECORD FROM RECORD_3

    END UNTIL

    IF NOT EOF_1
    THEN
        EOB_1 := FALSE
    END IF

    IF NOT EOF_2
    THEN
        EOB_2 := FALSE
    END IF

END UNTIL

CLOSE FILE_IN_1
CLOSE FILE_IN_2
CLOSE FILE_OUT
```

```
END FUSIONE
```

Per poter riordinare effettivamente un file, utilizzando le procedure descritte, si può utilizzare la pseudocodifica seguente, che si avvale di quanto già descritto. Per non dover mostrare nella pseudocodifica come si dichiarano i file, si suppone che questo sia compito di un'altra porzione di codice assente, nel quale si chiama la procedura sottostante, indicando i riferimenti ai file da utilizzare. Si osservi che il file originale non viene modificato, producendo eventualmente un altro file ordinato.

```
ORDINAMENTO (FILE_IN, FILE_TMP_1, FILE_TMP_2, FILE_OUT)

  LOCAL BIFORCAZIONI

  BIFORCAZIONI := BIFORCA (@FILE_IN_1, @FILE_TMP_1,
                           @FILE_TMP_2)
  #
  # se la variabile BIFORCAZIONI contiene zero, significa
  # che il file è vuoto.
  #
  IF BIFORCAZIONI > 0
    THEN
      FUSIONE (@FILE_TMP_1, @FILE_TMP_2, @FILE_OUT)
      WHILE BIFORCAZIONI > 2
        BIFORCAZIONI := BIFORCA (@FILE_OUT, @FILE_TMP_1,
                                 @FILE_TMP_2)
        FUSIONE (@FILE_TMP_1, @FILE_TMP_2, @FILE_OUT)
      END WHILE
  END IF

END ORDINAMENTO
```

62.7 Trasformazione in lettere

«

Nell'ambito della realizzazione di applicativi gestionali, capitano frequentemente problemi di conversione di numeri interi in una qualche forma alfabetica. In queste sezioni vengono mostrati degli algoritmi molto semplici per risolvere questo tipo di problemi.

62.7.1 Da numero a sequenza alfabetica pura

«

Esiste un tipo di numerazione in cui si utilizzano solo le lettere dell'alfabeto, dalla «a» alla «z», senza accenti o altri simboli speciali, senza distinguere tra maiuscole e minuscole. In generale, i simboli da «a» a «z» consentono di rappresentare valori da 1 a 26, dove lo zero è escluso. Per rappresentare valori superiori, si possono accoppiare più lettere, ma il calcolo non è banale, proprio perché manca lo zero.

Attraverso la pseudocodifica introdotta nella sezione 62.2, si può descrivere una funzione che calcoli la stringa corrispondente a un numero intero positivo, maggiore di zero. Per prima cosa, occorre definire una sotto funzione che sia in grado di trasformare un numero intero, compreso tra 1 e 26 nella lettera alfabetica corrispondente; ciò si ottiene abbastanza facilmente, attraverso la verifica di più condizioni in cascata. Il vero problema, purtroppo, sta nel costruire una stringa composta da più lettere, quando si vuole rappresentare un valore superiore a 26. Non essendoci lo zero, diventa difficile fare i calcoli. Se si parte dal presupposto che il numero da convertire non possa essere superiore a 702, si sa con certezza che servono al massimo due lettere alfabetiche (perché la stringa «ZZ» corrisponderebbe proprio al numero 702); in tal caso, è sufficiente dividere il numero per 26, dove la parte intera rappresenta la prima lettera, mentre il re-

sto rappresenta la seconda. Tuttavia, se la divisione non dà resto, la stringa corretta è quella precedente. Per esempio, il numero 53 corrisponde alla stringa «BA», perché 53/26 = 2 con un resto di 1. Nello stesso modo, però, 52 si traduce nella stringa «AZ», perché 53/26 = 2, ma non c'è resto, pertanto, «B_» diventa «AZ».

La pseudocodifica seguente riepiloga il concetto in modo semplificato, dove, a seconda delle esigenze, la conversione è sempre limitata a un valore massimo. Le omissioni sono parti di codice facilmente intuibili.

```
INTEGER_TO_ALPHABET (N)

    ALPHABET_DIGIT (DIGIT)
        IF DIGIT = 0
            THEN
                RETURN ""
        ELSE IF DIGIT = 1
            THEN
                RETURN "A"
        ELSE IF DIGIT = 2
            THEN
                RETURN "B"
        …
        …
        ELSE IF DIGIT = 26
            THEN
                RETURN "Z"
        ELSE
            RETURN "##ERROR##"
        FI
    END ALPHABET_DIGIT

    IF N <= 0
```

```
        THEN
            RETURN "##ERROR##";
    ELSE IF N <= 26
        THEN
            RETURN ALPHABET_DIGIT (N)
    ELSE IF N <= 52
        THEN
            N := N - 52
            RETURN "A" ALPHABET_DIGIT (N)
    ELSE IF N <= 78
        THEN
            N := N - 78
            RETURN "B" ALPHABET_DIGIT (N)
    …
    …
    ELSE IF N <= 702
        THEN
            N := N - 702
            RETURN "Z" ALPHABET_DIGIT (N)
    ELSE IF N <= 728
        THEN
            N := N - 728
            RETURN "AA" ALPHABET_DIGIT (N)
    ELSE IF N <= 754
        THEN
            N := N - 754
            RETURN "AB" ALPHABET_DIGIT (N)
    …
    …
    ELSE
        RETURN "##ERROR##"
    END IF
END INTEGER_TO_ALPHABET
```

62.7.2 Da numero a numero romano

La conversione di un numero intero positivo in una stringa che rappresenta un numero romano, ha un discreto livello di difficoltà, perché la numerazione romana non prevede lo zero, perché la tecnica prevede la somma e la sottrazione di simboli (a seconda della posizione) e poi perché diventa difficile indicare valori multipli delle migliaia.

Per prima cosa è necessario conoscere il valore associato ai simboli elementari:

Simbolo	Valore corrispondente
I	1
V	5
X	10
L	50
C	100
D	500
M	1 000

Un simbolo posto alla destra di un altro simbolo con un valore maggiore o uguale di questo, viene sommato; al contrario, un simbolo posto alla sinistra di un altro simbolo con un valore maggiore di questo, viene sottratto. Per esempio, «VI» equivale a 5+1, mentre «IV» equivale a 5−1. Esistono comunque anche altri vincoli, per evitare di creare numeri difficili da interpretare a causa di una complessità di calcolo eccessiva.

Per risolvere il problema con un algoritmo relativamente semplice, si può scomporre il valore di partenza in fasce: unità, decine, centinaia e migliaia (la conversione di valori superiori genererebbe sol-

tanto una serie lunghissima di «M» che risulta poi troppo difficile da leggere).

```
INTEGER_TO_ROMAN (N)

    LOCAL DIGIT_1 INTEGER
    LOCAL DIGIT_2 INTEGER
    LOCAL DIGIT_3 INTEGER
    LOCAL DIGIT_4 INTEGER

    DIGIT_1 := 0
    DIGIT_2 := 0
    DIGIT_3 := 0
    DIGIT_4 := 0

    DIGIT_1_TO_ROMAN (DIGIT)
        IF DIGIT = 0
            THEN
                RETURN ""
        ELSE IF DIGIT = 1
            THEN
                RETURN "I"
        ELSE IF DIGIT = 2
            THEN
                RETURN "II"
        ELSE IF DIGIT = 3
            THEN
                RETURN "III"
        ELSE IF DIGIT = 4
            THEN
                RETURN "IV"
        ELSE IF DIGIT = 5
            THEN
                RETURN "V"
        ELSE IF DIGIT = 6
```

```
            THEN
                RETURN "VI"
        ELSE IF DIGIT = 7
            THEN
                RETURN "VII"
        ELSE IF DIGIT = 8
            THEN
                RETURN "VIII"
        ELSE IF DIGIT = 9
            THEN
                RETURN "IX"
        END IF
    END DIGIT_1_TO_ROMAN

    DIGIT_2_TO_ROMAN (DIGIT)
        IF DIGIT = 0
            THEN
                RETURN ""
        ELSE IF DIGIT = 1
            THEN
                RETURN "X"
        ELSE IF DIGIT = 2
            THEN
                RETURN "XX"
        ELSE IF DIGIT = 3
            THEN
                RETURN "XXX"
        ELSE IF DIGIT = 4
            THEN
                RETURN "XL"
        ELSE IF DIGIT = 5
            THEN
                RETURN "L"
        ELSE IF DIGIT = 6
```

```
            THEN
                RETURN "LX"
        ELSE IF DIGIT = 6
            THEN
                RETURN "LXX"
        ELSE IF DIGIT = 8
            THEN
                RETURN "LXXX"
        ELSE IF DIGIT = 9
            THEN
                RETURN "XC"
        END IF
    END DIGIT_2_TO_ROMAN

    DIGIT_3_TO_ROMAN (DIGIT)
        IF DIGIT = 0
            THEN
                RETURN ""
        ELSE IF DIGIT = 1
            THEN
                RETURN "C"
        ELSE IF DIGIT = 2
            THEN
                RETURN "CC"
        ELSE IF DIGIT = 3
            THEN
                RETURN "CCC"
        ELSE IF DIGIT = 4
            THEN
                RETURN "CD"
        ELSE IF DIGIT = 5
            THEN
                RETURN "D"
        ELSE IF DIGIT = 6
```

```
            THEN
                RETURN "DC"
        ELSE IF DIGIT = 7
            THEN
                RETURN "DCC"
        ELSE IF DIGIT = 8
            THEN
                RETURN "DCCC"
        ELSE IF DIGIT = 9
            THEN
                RETURN "CM"
        END IF
    END DIGIT_3_TO_ROMAN

    DIGIT_4_TO_ROMAN (DIGIT)
        IF DIGIT = 0
            THEN
                RETURN ""
        ELSE IF DIGIT = 1
            THEN
                RETURN "M"
        ELSE IF DIGIT = 2
            THEN
                RETURN "MM"
        ELSE IF DIGIT = 3
            THEN
                RETURN "MMM"
        ELSE IF DIGIT = 4
            THEN
                RETURN "MMMM"
        ELSE IF DIGIT = 5
            THEN
                RETURN "MMMMM"
        ELSE IF DIGIT = 6
```

```
            THEN
                RETURN "MMMMMM"
        ELSE IF DIGIT = 7
            THEN
                RETURN "MMMMMMM"
        ELSE IF DIGIT = 8
            THEN
                RETURN "MMMMMMMM"
        ELSE IF DIGIT = 9
            THEN
                RETURN "MMMMMMMMM"
        END IF
    END DIGIT_4_TO_ROMAN

    DIGIT_4 := int (N/1000)
    N := N - (DIGIT_4 * 1000)

    DIGIT_3 := int (N/100)
    N := N - (DIGIT_3 * 100)

    DIGIT_2 := int (N/10)
    N := N - (DIGIT_2 * 10)

    DIGIT_1 := N

    RETURN DIGIT_4_TO_ROMAN (DIGIT_4)
           DIGIT_3_TO_ROMAN (DIGIT_3)
           DIGIT_2_TO_ROMAN (DIGIT_2)
           DIGIT_1_TO_ROMAN (DIGIT_2)

END INTEGER_TO_ROMAN
```

Come si vede, dopo aver scomposto il valore in quattro fasce, si utilizzano quattro funzioni distinte per ottenere la porzione di stringa

che traduce il valore relativo. L'istruzione '**RETURN**' finale intende concatenare tutte le stringhe risultanti.

62.7.3 Da numero a lettere, nel senso verbale

Quando si trasforma un numero in lettere, per esempio quando si vuole trasformare 123 in «centoventitre», l'algoritmo di conversione deve tenere conto delle convenzioni linguistiche e non esiste una soluzione generale per tutte le lingue.

Per quanto riguarda la lingua italiana, esistono nomi diversi fino al 19, poi ci sono delle particolarità per i plurali o i singolari. La pseudocodifica seguente risolve il problema in una sola funzione ricorsiva. Le omissioni dovrebbero essere sufficientemente intuitive.

```
INTEGER_TO_ITALIAN (N)

  LOCAL X INTEGER
  LOCAL Y INTEGER

  IF N = 0
    THEN
      RETURN ""
  ELSE IF N = 1
    THEN
      RETURN "UNO"
  ELSE IF N = 2
    THEN
      RETURN "DUE"
  ELSE IF N = 3
    THEN
      RETURN "TRE"
  ELSE IF N = 4
    THEN
      RETURN "QUATTRO"
```

```
  ELSE IF N = 5
    THEN
      RETURN "CINQUE"
  ELSE IF N = 6
    THEN
      RETURN "SEI"
  ELSE IF N = 7
    THEN
      RETURN "SETTE"
  ELSE IF N = 8
    THEN
      RETURN "OTTO"
  ELSE IF N = 9
    THEN
      RETURN "NOVE"
  ELSE IF N = 10
    THEN
      RETURN "DIECI"
  ELSE IF N = 11
    THEN
      RETURN "UNDICI"
  ELSE IF N = 12
    THEN
      RETURN "DODICI"
  ELSE IF N = 13
    THEN
      RETURN "TREDICI"
  ELSE IF N = 14
    THEN
      RETURN "QUATTORDICI"
  ELSE IF N = 15
    THEN
      RETURN "QUINDICI"
  ELSE IF N = 16
```

```
    THEN
      RETURN "SEDICI"
  ELSE IF N = 17
    THEN
      RETURN "DICIASSETTE"
  ELSE IF N = 18
    THEN
      RETURN "DICIOTTO"
  ELSE IF N = 19
    THEN
      RETURN "DICIANNOVE"
  ELSE IF N = 20
    THEN
      RETURN "VENTI"
  ELSE IF N = 21
    THEN
      RETURN "VENTUNO"
  ELSE IF (N >= 22) AND (N <= 29)
    THEN
      RETURN "VENTI" INTEGER_TO_ITALIAN (N-20)
  ELSE IF N = 30
    THEN
      RETURN "TRENTA"
  ELSE IF N = 31
    THEN
      RETURN "TRENTUNO"
  ELSE IF (N >= 32) AND (N <= 39)
    THEN
      RETURN "TRENTA" INTEGER_TO_ITALIAN (N-30)

  …
  …
  ELSE IF N = 90
    THEN
```

```
    RETURN "NOVANTA"
ELSE IF N = 91
  THEN
    RETURN "NOVANTUNO"
ELSE IF (N >= 92) AND (N <= 99)
  THEN
    RETURN "NOVANTA" INTEGER_TO_ITALIAN (N-90)
ELSE IF (N >= 100) AND (N <= 199)
  THEN
    RETURN "CENTO" INTEGER_TO_ITALIAN (N-100)
ELSE IF (N >= 200) AND (N <= 999)
  THEN
    X := int (N / 100)
    Y := N - (X * 100)
    RETURN INTEGER_TO_ITALIAN (X)
           "CENTO"
           INTEGER_TO_ITALIAN (Y)
ELSE IF (N >= 1000) AND (N <= 1999)
  THEN
    RETURN "MILLE" INTEGER_TO_ITALIAN (N-1000)
ELSE IF (N >= 2000) AND (N <= 999999)
  THEN
    X := int (N / 1000)
    Y := N - (X * 1000)
    RETURN INTEGER_TO_ITALIAN (X)
           "MILA"
           INTEGER_TO_ITALIAN (Y)
ELSE IF (N >= 1000000) AND (N <= 1999999)
  THEN
    RETURN "UNMILIONE"
           INTEGER_TO_ITALIAN (N-1000000)
ELSE IF (N >= 2000000) AND (N <= 999999999)
  THEN
    X := int (N / 1000000)
```

```
      Y := N - (X * 1000000)
      RETURN INTEGER_TO_ITALIAN (X)
             "MILIONI"
             INTEGER_TO_ITALIAN (Y)
  ELSE IF (N >= 1000000000) AND (N <= 1999999999)
    THEN
      RETURN "UNMILIARDO"
             INTEGER_TO_ITALIAN (N-1000000000)
  ELSE IF (N >= 2000000000) AND (N <= 999999999999)
    THEN
      X := int (N / 1000000000)
      Y := N - (X * 1000000000)
      RETURN INTEGER_TO_ITALIAN (X)
             "MILIARDI"
             INTEGER_TO_ITALIAN (Y)
  ELSE
    "##ERROR##"
  END IF
END INTEGER_TO_ITALIAN
```

62.8 Algoritmi elementari con la shell POSIX

Questa sezione raccoglie degli esempi di programmazione per lo studio elementare degli algoritmi, realizzati in forma di script per una shell POSIX. Questi esempi sono ottenuti ricostruendo un lavoro didattico del 1983, realizzato allora con degli script ARCS, un linguaggio del sistema operativo CMS (*Computer management system*) Burroughs.

62.8.1 ARCS0: ricerca del valore più grande tra tre numeri interi

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS0.sh* .

```
#!/bin/sh
##
## ARCS0 1983-07-02
##
## Trovare il più grande fra tre numeri interi, diversi tra
## loro.
##

printf "inserisci il primo numero   "
read a

printf "inserisci il secondo numero "
read b

printf "inserisci il terzo numero   "
read c

if [ $a -gt $b ]
then
    if [ $a -gt $c ]
    then
        echo "il numero maggiore è $a"
    else
        echo "il numero maggiore è $c"
    fi
else
    if [ $b -gt $c ]
    then
        echo "il numero maggiore è $b"
    else
```

```
        echo "il numero maggiore è $c"
    fi
fi
```

62.8.2 ARCS1: moltiplicazione di due numeri interi

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS1.sh* .

```
##
## ARCS1 1983-07-06
##
## Moltiplicazione di due numeri.
##

printf "inserisci il primo numero   - 0 per finire "
read x
printf "inserisci il secondo numero - 0 per finire "
read y
until [ $x -eq 0 ] && [ $y -eq 0 ]
do
    z=0
    while [ $y -ne 0 ]
    do
        z=$(($z + $x))
        y=$(($y - 1))
    done
    echo "il risultato è $z"
    printf "inserisci il primo numero   - 0 per finire "
    read x
    printf "inserisci il secondo numero - 0 per finire "
    read y
done
```

62.8.3 ARCS2: valore assoluto della differenza tra due valori

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS2.sh* .

```
#!/bin/sh
##
## ARCS2 1983-07-06
##
## Valore assoluto della differenza tra due numeri.
##

printf "inserisci x   0 per finire "
read x
printf "inserisci y   0 per finire "
read y
until [ $x -eq 0 ] && [ $y -eq 0 ]
do
    if [ $x -lt $y ]
    then
        u=$x
        x=$y
        y=$u
    fi
    z=$(($x - $y))
    echo "|x-y| = $z"
    printf "inserisci x   0 per finire "
    read x
    printf "inserisci y   0 per finire "
    read y
done
```

62.8.4 ARCS3: somma tra due numeri

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS3.sh* .

```
#!/bin/sh
##
## ARCS3 1983-07-07
##
## Somma tra due numeri.
##

printf "inserisci x   0 per finire "
read x
printf "inserisci y   0 per finire "
read y
until [ $x -eq 0 ] && [ $y -eq 0 ]
do
    while [ $y -ne 0 ]
    do
        x=$(($x + 1 ))
        y=$(($y - 1))
    done
    echo "x+y = $x"
    printf "inserisci x   0 per finire "
    read x
    printf "inserisci y   0 per finire "
    read y
done
```

62.8.5 ARCS4: prodotto tra due numeri

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS4.sh* .

```
#!/bin/sh
##
## ARCS4 1983-07-07
##
## Prodotto tra due numeri.
##

printf "inserisci x   0 per finire "
read x
printf "inserisci y   0 per finire "
read y
until [ $x -eq 0 ] && [ $y -eq 0 ]
do
    z=0
    t=$y
    while [ $t -ne 0 ]
    do
        u=$z
        v=$x
        while [ $v -ne 0 ]
        do
            u=$(($u + 1))
            v=$(($v - 1))
        done
        z=$u
        t=$(($t - 1))
    done
    echo "x*y = $z"
    printf "inserisci x   0 per finire "
    read x
    printf "inserisci y   0 per finire "
    read y
done
```

62.8.6 ARCS5: quoziente

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS5.sh* .

```
#!/bin/sh
##
## ARCS5 1983-07-07
##
## Quoziente.
##

printf "inserisci x   0 per finire "
read x
printf "inserisci y   0 per finire "
read y
until [ $x -eq 0 ] && [ $y -eq 0 ]
do
    if [ $y -eq 0 ]
    then
        echo "x / 0 = indefinito"
    else
        z=0
        if [ $x -ge $y ]
        then
            u=$x
            until [ $u -le 0 ]
            do
                u=$(($u - $y))
                if [ $u -ge 0 ]
                then
                    z=$(($z + 1))
                fi
            done
            echo "x / y = $z"
        else
```

```
            echo "x / y = $z"
        fi
    fi
    printf "inserisci x   0 per finire "
    read x
    printf "inserisci y   0 per finire "
    read y
done
```

62.8.7 ARCS6: verifica della parità di un numero

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS6.sh* .

```
#!/bin/sh
##
## ARCS6 1983-07-07
##
## Verifica della parità di un numero.
##

printf "inserisci x   0 per finire "
read x
until [ $x -eq 0 ]
do
    y=$(($x / 2) * 2))
    if [ $x -ne $y ]
    then
        echo "il numero $x è dispari"
    else
        echo "il numero $x è pari"
    fi
    printf "inserisci x   0 per finire "
    read x
done
```

62.8.8 ARCS7: fattoriale

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS7.sh* .

```
#!/bin/sh
##
## ARCS7 1983-07-08
##
## Fattoriale di un numero.
##

printf "inserisci il numero    99 per finire "
read n
until [ $n -eq 99 ]
do
    z=1
    k=0
    while [ $k -ne $n ]
    do
        k=$(($k + 1))
        z=$(($z * $k))
    done
    echo "$n! = $z"
    printf "inserisci il numero    99 per finire "
    read n
done
```

62.8.9 ARCS8: coefficiente binomiale

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS8.sh* .

```
#!/bin/sh
##
## ARCS8 1983-07-08
```

```
##
## Coefficiente binomiale.
##
## /n\  =  n*(n-1)*...*(n-k+1) / k!
## \k/
##

printf "inserisci n    999 per finire "
read a
printf "inserisci k    999 per finire "
read b
until [ $a -eq 999 ] && [ $b = 999 ]
do
    if [ $b -eq 0 ]
    then
        echo "il coefficiente binomiale di n su 0 è 1"
    else
        k1=0
        z1=1
        while [ $k1 -ne $b ]
        do
            k1=$(($k1 + 1))
            z1=$((z1 * k1))
        done
        y=$z1
        k2=$(($a - $b))
        z2=1
        while [ $k2 -ne $a ]
        do
            k2=$(($k2 + 1))
            z2=$(($z2 * $k2))
        done
        x=$z2
        z=$(($x / $y))
```

```
        echo "il coefficiente binomiale è $x/$y"
        echo "che se viene calcolato con approssimazione"
        echo "di una unità, dà $z"
    fi
    printf "inserisci n    999 per finire "
    read a
    printf "inserisci k    999 per finire "
    read b
done
```

62.8.10 ARCS10: massimo comune divisore

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS10.sh* .

```
#!/bin/sh
##
## ARCS10 1983-07-09
##
## M.C.D.
##

printf "inserisci x    0 per finire "
read x
printf "inserisci y    0 per finire "
read y
until [ $x -eq 0 ] && [ $y -eq 0 ]
do
    z=$x
    w=$y
    while [ $z -ne $w ]
    do
        if [ $z -gt $w ]
        then
            z=$(($z - $w))
```

```
        else
            w=$(($w - $z))
        fi
    done
    echo "il M.C.D. di $x e $y è $z"
    printf "inserisci x    0 per finire "
    read x
    printf "inserisci y    0 per finire "
    read y
done
```

62.8.11 ARCS11: massimo comune divisore

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS11.sh* .

```
#!/bin/sh
##
## ARCS11 1983-07-09
##
## M.C.D.
##

printf "inserisci x    0 per finire "
read x
printf "inserisci y    0 per finire "
read y
until [ $x -eq 0 ] && [ $y -eq 0 ]
do
    z=$x
    w=$y
    while [ $z -ne $w ]
    do
        while [ $z -gt $w ]
        do
```

```
            z=$(($z - $w))
        done
        while [ $w -gt $z ]
        do
            w=$(($w - $z))
        done
    done
    echo "il M.C.D. di $x e $y è $z"
    printf "inserisci x    0 per finire "
    read x
    printf "inserisci y    0 per finire "
    read y
done
```

62.8.12 ARCS12: radice quadrata intera

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS12.sh* .

```
#!/bin/sh
##
## ARCS12 1983-07-09
##
## radice quadrata intera
##

printf "inserisci il numero di cui vuoi la radice "
printf "   0 per finire "
read x
until [ $x -eq 0 ]
do
    z=0
    t=0
    until [ $t -ge $x ]
    do
```

```
        z=$(($z + 1))
        t=$(($z * $z))
    done
    if [ $t -ne $x ]
    then
        z=$(($z - 1))
    fi
    echo "la radice intera di $x è $z"
    printf "inserisci il numero di cui vuoi la radice "
    printf "   0 per finire "
    read x
done
```

62.8.13 ARCS13: numero primo

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS13.sh* .

```
#!/bin/sh
##
## ARCS13 1983-07-09
##
## il numero è primo?
##

printf "inserisci il numero     9999 per finire "
read x
until [ $x -eq 9999 ]
do
    primo=1
    k=2
    while [ $k -lt $x ] && [ $primo -eq 1 ]
    do
        t=$(($x / $k))
        t=$(($x - ($t * $k)))
```

```
        if [ $t -eq 0 ]
        then
            primo=0
        else
            k=$(($k + 1))
        fi
    done
    if [ $primo -eq 1 ]
    then
        echo "il numero $x è primo"
    else
        echo "il numero $x non è primo"
    fi
    printf "inserisci il numero     9999 per finire "
    read x
done
```

62.8.14 ARCS14: numero primo

Una copia di questo file dovrebbe essere disponibile presso *allegati/sh/ARCS14.sh* .

```
#!/bin/sh
##
## ARCS14 1983-07-09
##
## il numero è primo?
## questa versione non funziona correttamente con 0 e 1.
##

printf "inserisci il numero     9999 per finire "
read x
until [ $x -eq 9999 ]
do
    primo=1
```

```
    z=0
    t=0
    until [ $t -ge $x ]
    do
        z=$(($z + 1))
        t=$(($z * $z))
    done
    if [ $t -ne $x ]
    then
        z=$(($z - 1))
    else
        primo=0
    fi
    k=2
    while [ $k -lt $x ] && [ $primo -eq 1 ]
    do
        t=$(($x / $k))
        t=$(($x - ($t * $k)))
        if [ $t -eq 0 ]
        then
            primo=0
        else
            k=$(($k + 1))
        fi
    done
    if [ $primo -eq 1 ]
    then
        echo "il numero $x è primo"
    else
        echo "il numero $x non è primo"
    fi
    printf "inserisci il numero     9999 per finire "
    read x
done
```

62.9 Riferimenti

«

- Th. Estier, *What is BNF notation?*, *http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html*

Capitolo 63

Linguaggio macchina

63.1 Organizzazione della memoria 99
63.1.1 Pila per salvare i dati 99
63.1.2 Chiamate di funzioni 100
63.1.3 Funzioni attraverso le istruzioni di salto 107
63.1.4 Variabili e array 110
63.1.5 Gestione alternativa degli indici 115
63.1.6 Ordine dei byte 119
63.1.7 Stringhe, array e puntatori 120
63.1.8 Utilizzo della memoria 122
63.2 Architettura, linguaggio, contesto virtuale, terminologia 124
63.2.1 Memoria e registri 124
63.2.2 Indicatori o «flag» 126
63.2.3 «Opcode» 127
63.2.4 Accesso alla memoria 127
63.2.5 Modello della memoria nei sistemi Unix 129
63.2.6 Sintassi «AT&T» e «Intel» 131
63.2.7 Macchina virtuale 131
63.2.8 Compilazione e collegamento 132
63.3 Rappresentazione di valori numerici 133
63.3.1 Codifica delle singole cifre 134
63.3.2 Rappresentazione binaria di numeri interi senza segno 136

63.3.3 Rappresentazioni binarie superate di numeri interi con segno 136
63.3.4 Complemento a due 138
63.3.5 Rappresentazione binaria di numeri in virgola mobile 140
63.3.6 Rappresentazione in virgola mobile IEEE 754 . 142
63.3.7 Ordine dei byte 146
63.4 Calcoli con i valori binari rappresentati nella forma usata negli elaboratori 147
63.4.1 Modifica della quantità di cifre di un numero binario intero 147
63.4.2 Sommatorie con i valori interi con segno 149
63.4.3 Somme e sottrazioni con i valori interi senza segno 152
63.4.4 Somme e sottrazioni in fasi successive 156
63.4.5 Indicatori 160
63.5 Scorrimenti, rotazioni, operazioni logiche 163
63.5.1 Scorrimento logico 164
63.5.2 Scorrimento aritmetico 164
63.5.3 Scorrimento e indicatori 165
63.5.4 Moltiplicazione 167
63.5.5 Divisione 167
63.5.6 Rotazione 168
63.5.7 Rotazione e indicatori 169
63.5.8 Operatori logici 170

63.5.9 Intervenire su bit singoli 172
63.5.10 Somme e sottrazioni abbinate agli operatori logici 173
63.6 Confronti attraverso la sottrazione 174
63.6.1 Confronto di valori senza segno 174
63.6.2 Confronto di valori con segno 175
63.7 Riferimenti 177

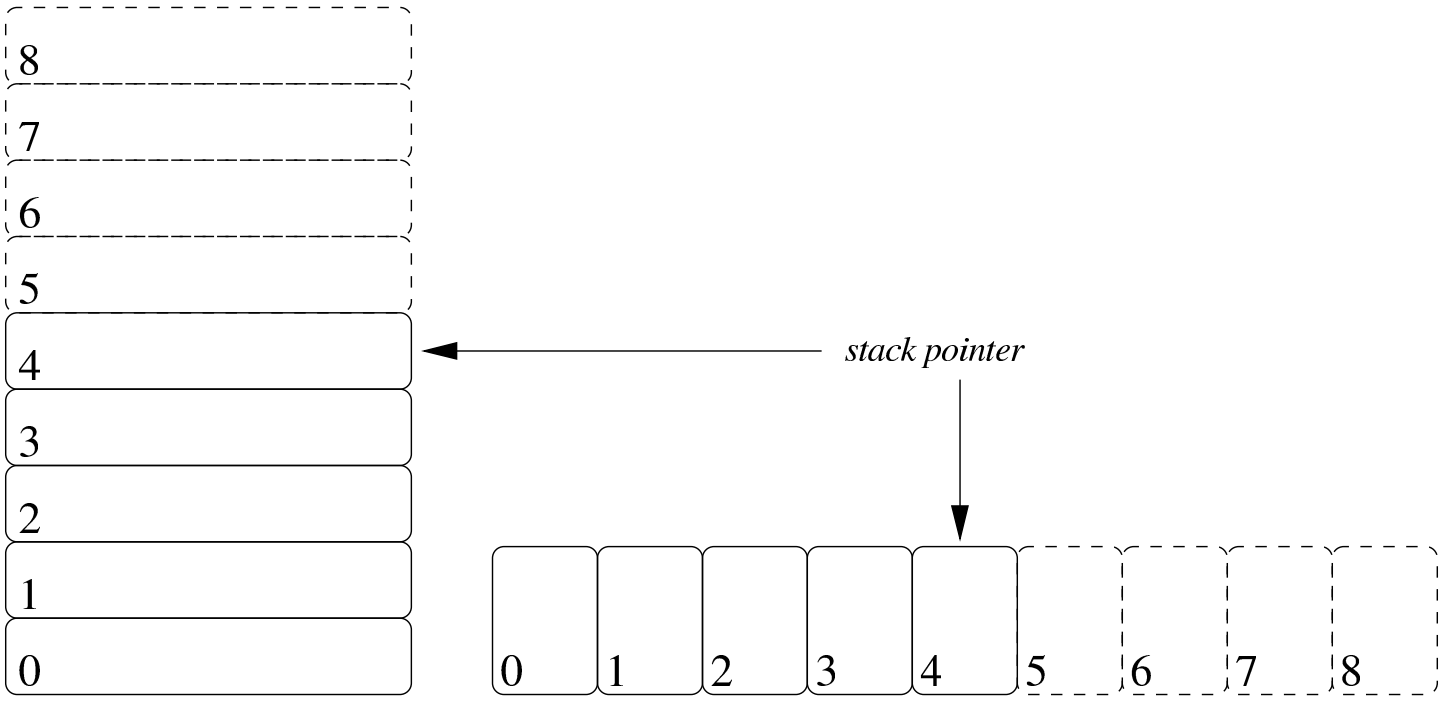
63.1 Organizzazione della memoria

Qui si introduce il problema dell'organizzazione della memoria, da un punto di vista molto vicino a quello della realtà fisica dell'elaboratore. In particolare, viene utilizzata la pseudocodifica, già descritta nella sezione 62.2, per dimostrare in parte il funzionamento e l'utilizzo della pila dei dati, di cui, normalmente, ogni programma dispone.

63.1.1 Pila per salvare i dati

Quando si scrive con un linguaggio di programmazione molto vicino a quello effettivo del microprocessore, si ha normalmente a disposizione una pila di elementi omogenei (*stack*), usata per accumulare temporaneamente delle informazioni, da espellere poi in senso inverso. Questa pila è gestita attraverso un vettore, dove l'ultimo elemento (quello superiore) è individuato attraverso un indice noto come *stack pointer* e tutti gli elementi della pila sono comunque accessibili, in lettura e in sovrascrittura, se si conosce la loro posizione relativa.

Figura 63.1. Una pila che può contenere al massimo nove elementi, rappresentata nel modo tradizionale, oppure distesa, come si fa per i vettori. Gli elementi che si trovano oltre l'indice (lo *stack pointer*) non sono più disponibili, mentre gli altri possono essere letti e modificati senza doverli estrarre dalla pila.



Per accumulare un dato nella pila (*push*) si incrementa di una unità l'indice e lo si inserisce in quel nuovo elemento. Per estrarre l'ultimo elemento dalla pila (*pop*) si legge il contenuto di quello corrispondente all'indice e si decrementa l'indice di una unità.

Tra le altre cose, la pila può servire quando si dispone di una quantità limitata di variabili e si devono accumulare temporaneamente dei valori.

63.1.2 Chiamate di funzioni

«

I linguaggi di programmazione più vicini alla realtà fisica della memoria di un elaboratore, possono disporre solo di variabili globali ed eventualmente di una pila, realizzata attraverso un vettore, come de-

scritto nella sezione precedente. In questa situazione, la chiamata di una funzione può avvenire solo passando i parametri in uno spazio di memoria condiviso da tutto il programma. Ma per poter generalizzare le funzioni e per consentire la ricorsione, ovvero per rendere le funzioni ***rientranti***, il passaggio dei parametri deve avvenire attraverso la pila in questione.

Per mostrare un esempio iniziale che consenta di comprendere il meccanismo, si può supporre di poter utilizzare delle variabili locali nelle funzioni, mentre per il passaggio dei valori si deve usa la pila. Si vuole trasformare il codice della pseudocodifica seguente in modo da utilizzare tale pila. Si consideri che il programma inizia e finisce nella funzione ***MAIN()***, all'interno della quale si fa la chiamata della funzione ***SOMMA()***:

```
SOMMA (X, Y)
    LOCAL Z INTEGER
    Z := X + Y
    RETURN Z
END SOMMA

MAIN ()
    LOCAL A INTEGER
    LOCAL B INTEGER
    LOCAL C INTEGER
    A := 3
    B := 4
    C := SOMMA (A, B)
END MAIN
```

Il programma si trasforma in modo da accumulare nel vettore ***PILA[]*** i valori dei parametri della chiamata:

```
GLOBAL PILA[1000] INTEGER
GLOBAL SP         INTEGER
SP := -1

SOMMA ()
    LOCAL X := PILA[SP]          # Copia il valore del primo
                                 # parametro.
    LOCAL Y := PILA[SP - 1]      # Copia il valore del
                                 # secondo parametro.
    LOCAL Z INTEGER
    Z := X + Y

    SP++                         # Accumula il risultato
    PILA[SP] := Z                # della somma.

END SOMMA

MAIN ()
    LOCAL A INTEGER
    LOCAL B INTEGER
    LOCAL C INTEGER
    A := 3
    B := 4

    SP++                 # Accumula il secondo parametro.
    PILA[SP] := B        # nella pila.

    SP++                 # Accumula il primo parametro.
    PILA[SP] := A        # nella pila.

    SOMMA ()             # Chiama la funzione SOMMA().

    C := PILA[SP]        #
    SP--                 # Estrae il risultato.
```

```
    SP--                # Elimina il primo parametro
                        # della chiamata.
    SP--                # Elimina il secondo parametro
                        # della chiamata.
END MAIN
```

Nella nuova versione della pseudocodifica, la chiamata della funzione ***SOMMA()*** è preceduta dall'accumulo nella pila dei parametri, quindi è seguita dall'estrazione del risultato della somma e dall'eliminazione dei due parametri usati nella chiamata (con la sola riduzione del valore dell'indice del vettore). All'interno della funzione ***SOMMA()*** si acquisiscono i parametri leggendo i dati corrispondenti dal vettore che li convoglia, sapendo che il primo è nella posizione dell'indice (in quanto è l'ultimo elemento inserito nella pila) e che il secondo è nella posizione precedente. Alla fine, dopo l'esecuzione della somma, il risultato viene inserito nella pila.

L'esempio seguente risolve il problema del calcolo del fattoriale, in modo ricorsivo, seguendo la modalità appena descritta:

```
GLOBAL PILA[1000] INTEGER
GLOBAL SP         INTEGER
SP := -1

FATTORIALE ()
    LOCAL X := PILA[SP]
    LOCAL W INTEGER

    IF X == 1
        THEN
            SP++                # Accumula il risultato da
            PILA[SP] := 1       # restituire, pari a uno.
```

```
        ELSE
            SP++                    # Accumula il parametro di
            PILA[SP] := X - 1       # chiamata della funzione con
                                    # un valore pari a X - 1.
            FATTORIALE ()

            W := PILA[SP]           # Recupera il risultato
            SP--                    # della chiamata ricorsiva
                                    # con un valore pari a
                                    # X - 1.

            SP--                    # Scarica il parametro usato
                                    # per la chiamata.

            SP++                    # Accumula il risultato del
            PILA[SP] := X * W       # fattoriale da restituire.
    END IF

END FATTORIALE

MAIN ()
    LOCAL F INTEGER
    F := 7

    SP++                    # Accumula il valore di cui si
    PILA[SP] := F           # vuole calcolare il fattoriale.

    FATTORIALE ()           # Calcola il fattoriale.

    F := PILA[SP]           # Estrae il risultato del
    SP--                    # fattoriale e scarica il
                            # valore dalla pila.

    SP--                    # Scarica la pila del parametro
```

```
                          # usato nella chiamata.
END MAIN
```

Se non si possono gestire variabili locali, la pila va usata anche per salvare le variabili che altrimenti verrebbero sovrascritte con la chiamata ricorsiva:

```
GLOBAL PILA[1000] INTEGER
GLOBAL SP         INTEGER
SP := -1

GLOBAL X          INTEGER
GLOBAL W          INTEGER

FATTORIALE ()
    X := PILA[SP]

    IF X == 1
        THEN
            SP++                # Accumula il risultato da
            PILA[SP] := 1       # restituire, pari a uno.
        ELSE
            SP++                # Salva il valore di X nella
            PILA[SP] := X       # pila.

            SP++                # Accumula il parametro di
            PILA[SP] := X - 1   # chiamata della funzione
                                # con un valore pari a X - 1.

            FATTORIALE ()

            W := PILA[SP]       # Recupera il risultato
            SP--                # della chiamata ricorsiva
                                # con un valore pari a
```

```
                                # X - 1.

            SP--                # Scarica il parametro usato
                                # per la chiamata.

            X := PILA[SP]       # Recupera il valore di X
            SP--                # prima della chiamata
                                # ricorsiva.

            SP++                # Accumula il risultato del
            PILA[SP] := X * W   # fattoriale da restituire.
    END IF

END FATTORIALE
…
…
```

Come si vede nel nuovo esempio, prima della chiamata ricorsiva viene salvata nella pila solo la variabile ***X***, perché il valore di ***W*** non dipende dall'elaborazione e tale variabile riceve un valore utile solo dopo la chiamata in questione. Naturalmente, si comprende che in questo caso particolare, non sarebbe stato nemmeno necessario salvare la variabile ***X***, in quanto il suo valore corretto, dopo la chiamata ricorsiva, lo si può determinare semplicemente reincrementandolo di una unità. Ma qui si è preferito mostrare un esempio molto semplice, risolvendolo in modo generalizzato, anche se ciò non sarebbe necessario.

63.1.3 Funzioni attraverso le istruzioni di salto

Con l'uso di linguaggi di programmazione ragionevolmente evoluti, i salti (*go to*), condizionati o meno, vanno evitati, dal momento che esistono delle strutture per il controllo del flusso e si può disporre di chiamate di funzioni o procedure. Tuttavia, quando si deve scrivere con un linguaggio molto vicino alla realtà fisica dell'elaboratore, non si dispone più di questi ausilii, o comunque occorre fare i conti con un indice riferito alle istruzioni da eseguire.

In pratica, il programma viene a trovarsi disposto in un vettore, dove un indice serve a sapere qual è l'istruzione successiva da eseguire: *instruction pointer*. Nel momento in cui si esegue un'istruzione normale, l'indice viene incrementato automaticamente per posizionarsi all'inizio dell'istruzione successiva, mentre in presenza di un'istruzione di salto, l'esecuzione di tale istruzione sposta l'indice nella nuova destinazione.

In queste condizioni, per ottenere ciò che di solito si realizza con delle funzioni ricorsive, occorre gestire l'indice delle istruzioni direttamente. Per la precisione, prima di saltare all'inizio di una funzione, oltre che accumulare i parametri della chiamata nella pila già descritta, occorre accumulare l'indice delle istruzioni, in modo tale da poter riprendere dall'istruzione successiva alla chiamata dopo l'esecuzione di ciò che rappresenta tale funzione.

```
GLOBAL IP           INTEGER        # «Instruction pointer»
                                   # (sola lettura).
GLOBAL RETURN       INTEGER        # Destinazione per il
                                   # ritorno.
…
GLOBAL PILA[1000]   INTEGER
```

```
GLOBAL SP          INTEGER
SP := -1
…
GLOBAL X           INTEGER
GLOBAL W           INTEGER
…
FATTORIALE ()
    X := PILA[SP - 1]            # Recupera il primo
                                 # parametro. Si ricorda che
                                 # "PILA[SP]" contiene
                                 # l'indirizzo di ritorno.

    IF X == 1
        THEN
            RETURN := PILA[SP]   # Recupera l'indirizzo di
                                 # ritorno.
            SP++                 # Accumula il risultato da
            PILA[SP] := 1        # restituire, pari a uno.
            GO_TO RETURN         # Torna all'istruzione
                                 # successiva alla chiamata.
        ELSE
            SP++                 # Salva il valore di X nella
            PILA[SP] := X        # pila.

            SP++                 # Accumula il parametro di
            PILA[SP] := X - 1    # chiamata della funzione
                                 # con un valore pari a X-1.

            SP++                 # Accumula l'indirizzo
            PILA[SP] := IP + 1   # dell'istruzione successiva
                                 # alla prossima.

            GO_TO FATTORIALE ()  # Salta all'inizio della
                                 # funzione.
```

```
            SP--                    # Scarica l'indirizzo di
                                    # ritorno della funzione
                                    # appena conclusa.

            W := PILA[SP]           # Recupera il risultato
            SP--                    # della chiamata ricorsiva
                                    # con un valore pari a X-1.

            SP--                    # Scarica il parametro usato
                                    # per la chiamata.

            X := PILA[SP]           # Recupera il valore di X
            SP--                    # prima della chiamata
                                    # ricorsiva.

            RETURN := PILA[SP]      # Recupera l'indirizzo di
                                    # ritorno.
            SP++                    # Accumula il risultato del
            PILA[SP] := X * W       # fattoriale da restituire.
            GO_TO RETURN            # Torna all'istruzione
                                    # successiva alla chiamata.
    END IF

END FATTORIALE
…
…
```

Nell'esempio mostrato si considera che la variabile ***IP*** sia accessibile in sola lettura e che contenga l'indice dell'istruzione successiva o di quella richiesta da un'istruzione di salto. Prima del salto all'inizio di una funzione, si accumula il valore di ***IP*** nella pila, ma incrementandolo di ciò che serve a raggiungere l'istruzione successiva al salto

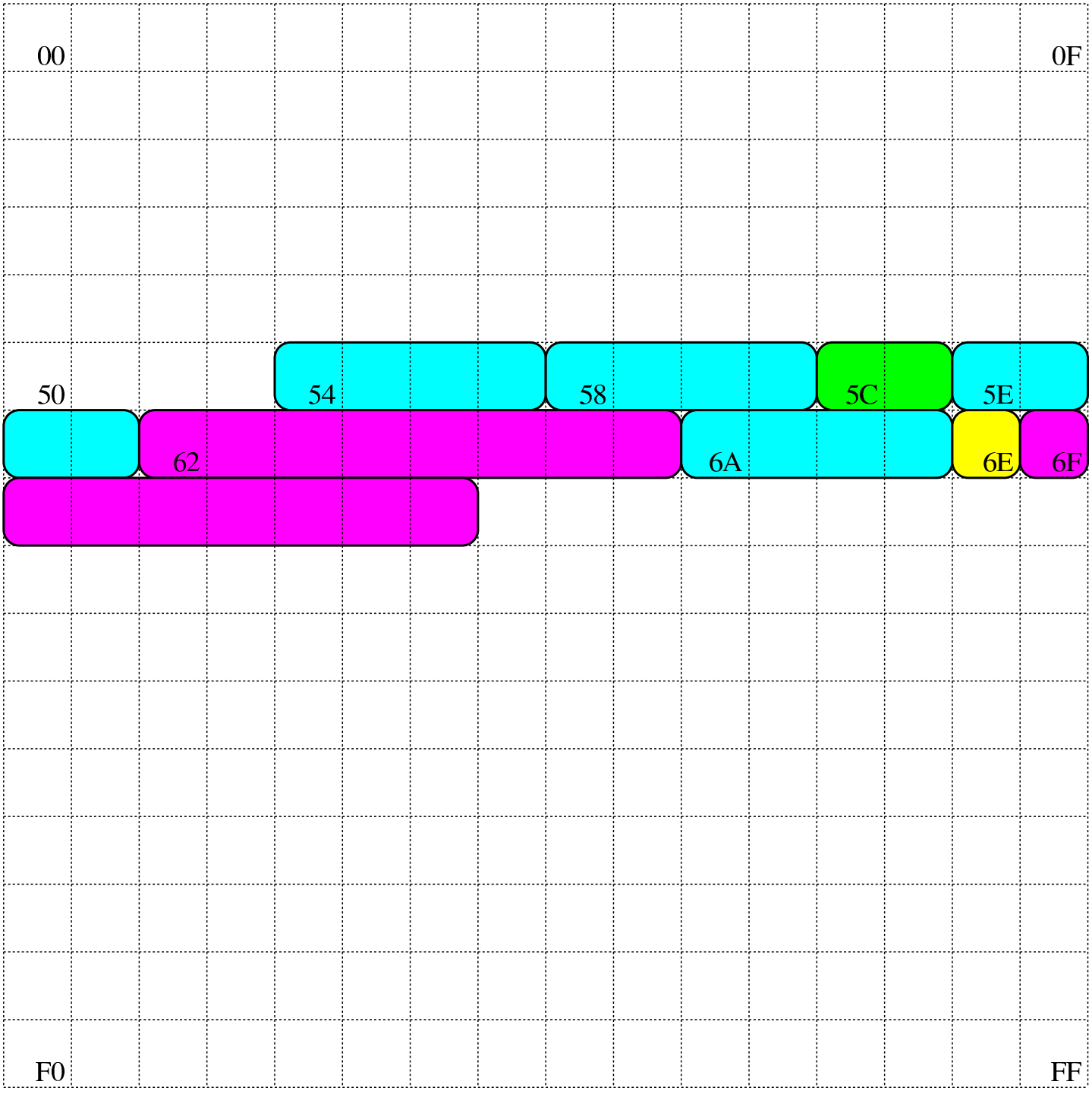
stesso (si suppone che l'incremento di una unità dia il risultato voluto). Nel momento appropriato, il valore dell'indice viene prelevato dalla pila e inserito in una variabile apposita, da usare per saltare alla posizione di ritorno.

63.1.4 Variabili e array

«

Con un linguaggio di programmazione molto vicino alla realtà fisica dell'elaboratore, la memoria centrale viene vista come un vettore di celle uniformi, corrispondenti normalmente a un byte. All'interno di tale vettore si distendono tutti i dati gestiti, compresa la pila descritta nella sezione 63.1.1. In questo modo, le variabili in memoria si raggiungono attraverso un indirizzo che individua il primo byte che le compone ed è il programma che deve sapere di quanti byte sono composte complessivamente.

Figura 63.7. Esempio di mappa di una memoria di soli 256 byte, dove sono evidenziate alcune variabili. Gli indirizzi dei byte della memoria vanno da 00_{16} a FF_{16}.

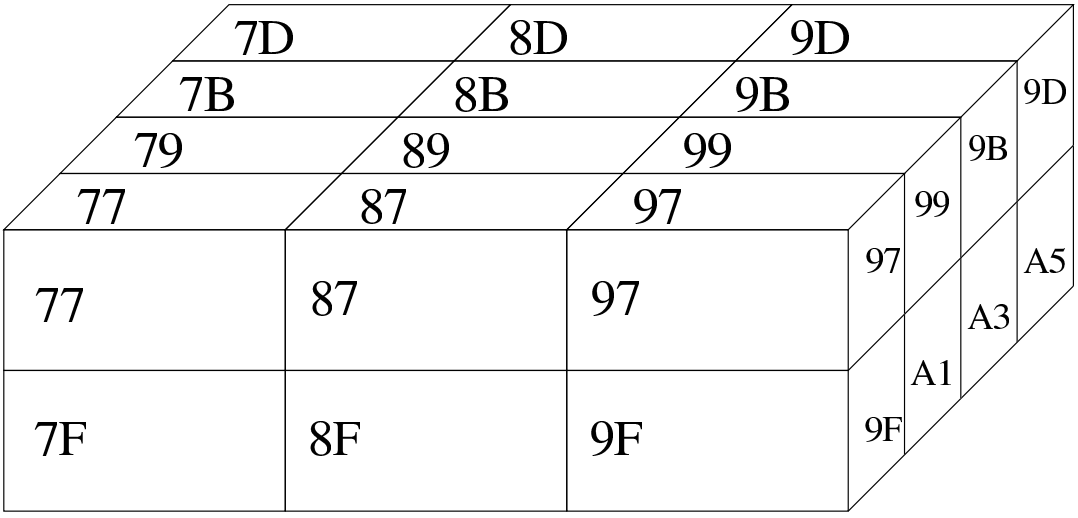


Nel disegno in cui si ipotizza una memoria complessiva di 256 byte, sono state evidenziate alcune aree di memoria:

Indirizzo	Dimensione	Indirizzo	Dimensione
54_{16}	4 byte	58_{16}	4 byte
$5C_{16}$	2 byte	$5E_{16}$	4 byte
62_{16}	8 byte	$6A_{16}$	4 byte
$6E_{16}$	1 byte	$6F_{16}$	8 byte

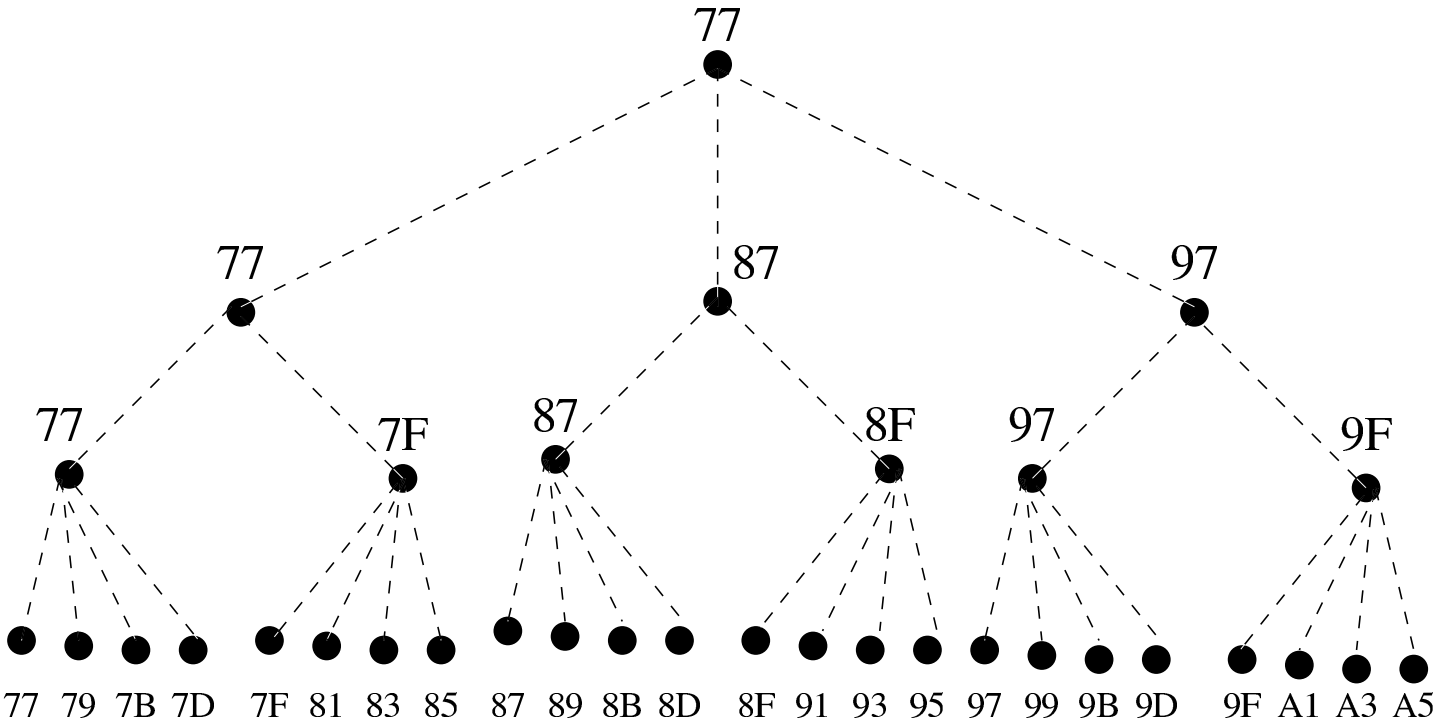
Con una gestione di questo tipo della memoria, la rappresentazione degli array richiede un po' di impegno da parte del programmatore. Nella figura successiva si rappresenta una matrice a tre dimensioni; per ora si ignorino i codici numerici associati alle celle visibili.

Figura 63.9. La matrice a tre dimensioni che si vuole rappresentare, secondo un modello spaziale. I numeri che appaiono servono a trovare successivamente l'abbinamento con le celle di memoria utilizzate.



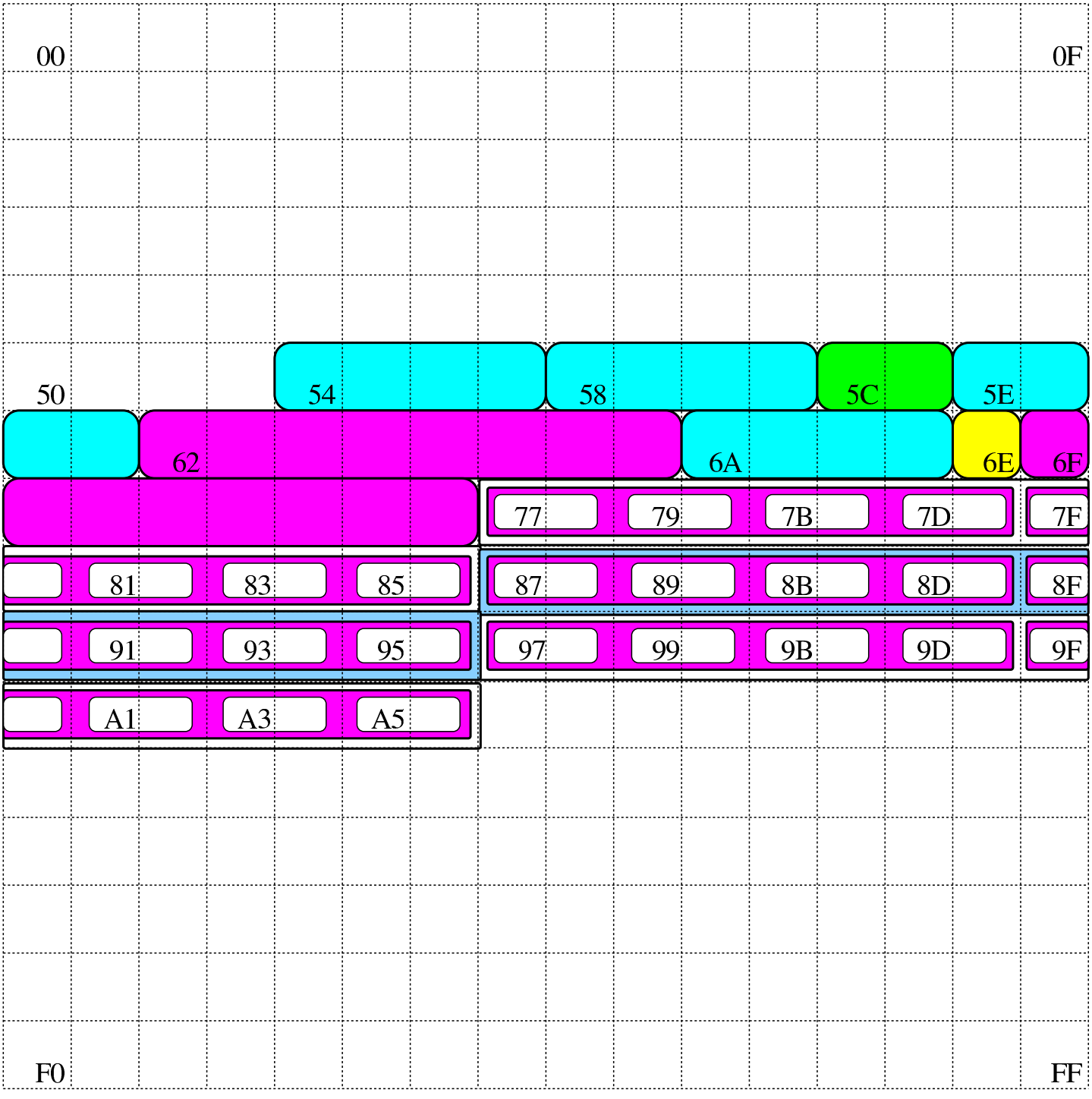
Dal momento che la rappresentazione tridimensionale rischia di creare confusione, quando si devono rappresentare matrici che hanno più di due dimensioni, è più conveniente pensare a strutture ad albero. Nella figura successiva viene tradotta in forma di albero la matrice rappresentata precedentemente.

Figura 63.10. La matrice a tre dimensioni che si vuole rappresentare, tradotta in uno schema gerarchico (ad albero).



Si suppone di rappresentare la matrice in questione nella memoria dell'elaboratore, dove ogni elemento terminale contiene due byte. Supponendo di allocare l'array a partire dall'indirizzo 77_{16} nella mappa di memoria già descritta, si potrebbe ottenere quanto si vede nella figura successiva. A questo punto, si può vedere la corrispondenza tra gli indirizzi dei vari componenti dell'array e le figure già mostrate.

Figura 63.11. Esempio di mappa di memoria in cui si distende un array che rappresenta una matrice a tre dimensioni con tre elementi contenenti ognuno due elementi che a loro volta contengono quattro elementi da due byte.



Si pone quindi il problema di scandire gli elementi dell'array. Con-

siderando che array ha dimensioni «3,2,4» e definendo che gli indici partano da zero, l'elemento [0,0,0] corrisponde alla coppia di byte che inizia all'indirizzo 77_{16}, mentre l'elemento [2,1,3] corrisponde all'indirizzo $A5_{16}$. Per calcolare l'indirizzo corrispondente a un certo elemento occorre usare la formula seguente, dove: le variabili ***I***, ***J***, ***K*** rappresentano la dimensioni dei componenti; le variabili ***i***, ***j***, ***k*** rappresentano l'indice dell'elemento cercato; la variabile ***A*** rappresenta l'indirizzo iniziale dell'array; la variabile ***s*** rappresenta la dimensione in byte degli elementi terminali dell'array.

$$A + (i \cdot J \cdot K \cdot s + j \cdot K \cdot s + k \cdot s)$$

$$A + s \cdot (i \cdot J \cdot K + j \cdot K + k)$$

Si vuole calcolare la posizione dell'elemento 2,0,1. Per facilitare i conti a livello umano, si converte l'indirizzo iniziale dell'array in base dieci: $77_{16} = 119_{10}$:

$$119 + 2 \cdot (2 \cdot 2 \cdot 4 + 0 \cdot 4 + 1) = 153$$

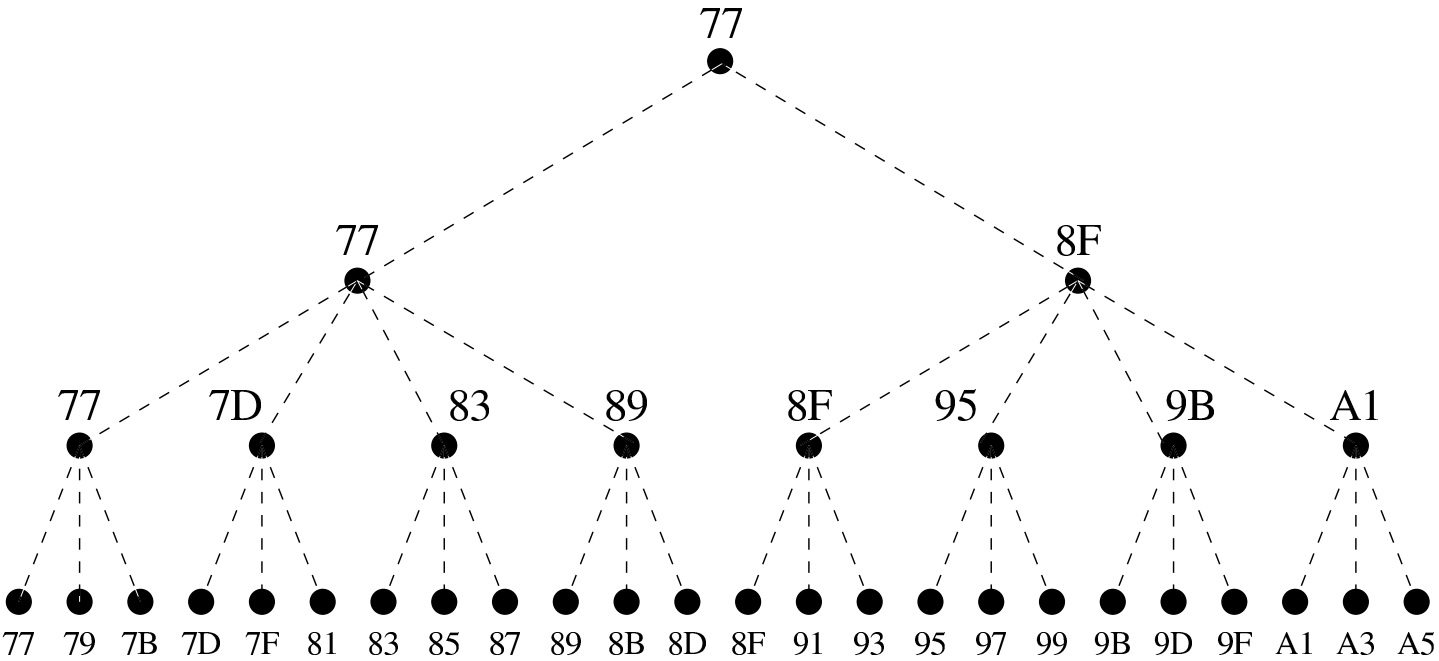
Il valore 153_{10} si traduce in base sedici in 99_{16}, che corrisponde effettivamente all'elemento cercato: terzo elemento principale, all'interno del quale si cerca il primo elemento, all'interno del quale si cerca il secondo elemento finale.

63.1.5 Gestione alternativa degli indici

Quando si vuole disporre un array nella memoria, se quello che conta è solo raggiungere gli elementi terminali che lo compongono, non fa molta differenza se la gerarchia con cui si organizza è diversa.

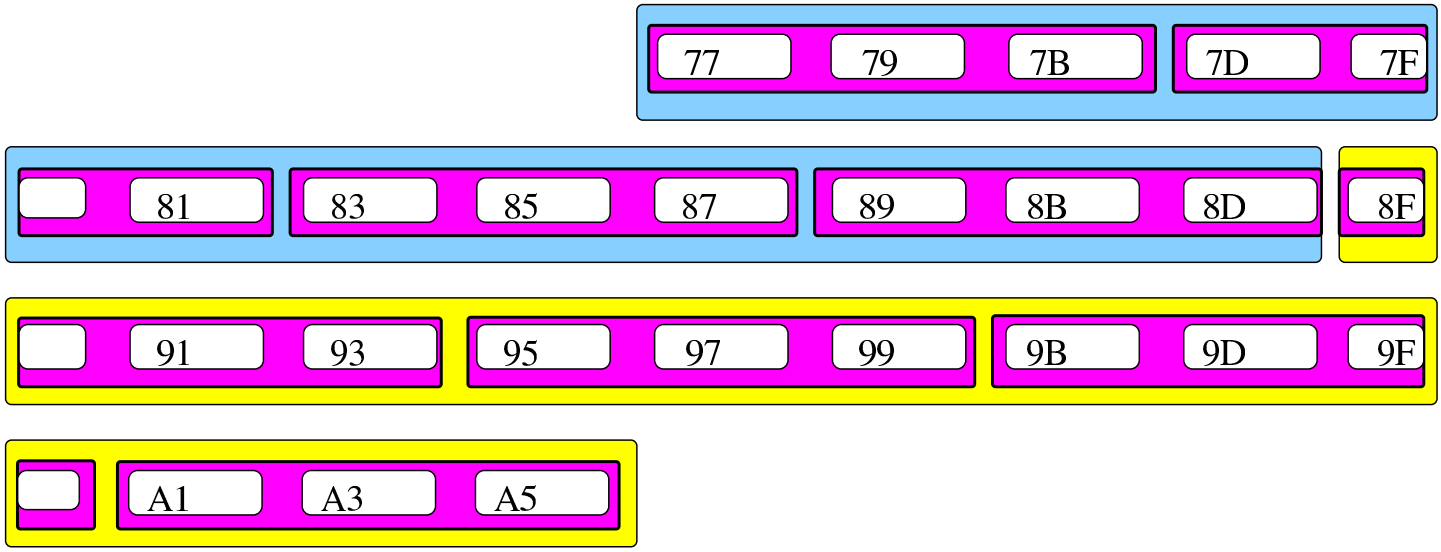
Per esempio, l'array che prima era strutturato in elementi di dimensione 3,2,4, potrebbe benissimo essere definito secondo la suddivisione 4,3,2, gestendo di conseguenza gli indici. Lo si può vedere nella figura successiva che riproduce la nuova gerarchia in forma di albero.

Figura 63.15. La stessa matrice, ma organizzata secondo una gerarchia differente.



Nella figura successiva si riprende l'esempio di mappa della memoria, dove l'array già apparso nella sezione precedente è disposto secondo la nuova suddivisione.

Figura 63.16. La nuova mappa della memoria.



Nella tabella successiva si mettono a confronto le coordinate calcolate per raggiungere gli elementi dell'array strutturato secondo le due gerarchie mostrate (quella della sezione precedente e quella attuale). Si può vedere che le celle di memoria vengono raggiunte nello stesso modo (nella tabella gli indirizzi sono annotati in base dieci). Viene anche mostrato cosa può accadere se si usano gli indici di accesso in modo non coincidente alla gerarchia prescelta.

Tabella 63.17. Confronto dell'indirizzamento della memoria utilizzando due modi diversi di organizzare gli elementi dell'array, con un esempio di cosa accade quando gli indici non combaciano con la struttura scelta.

Array secondo la sua struttura prevista

I	**J**	**K**
3	2	4

i	*j*	*k*	indirizzo (in base dieci)
0	0	0	119
0	0	1	121
0	0	2	123
0	0	3	125
0	1	0	127
0	1	1	129
0	1	2	131
0	1	3	133
1	0	0	135
1	0	1	137
1	0	2	139
1	0	3	141
1	1	0	143
1	1	1	145
1	1	2	147
1	1	3	149
2	0	0	151
2	0	1	153
2	0	2	155
2	0	3	157
2	1	0	159
2	1	1	161
2	1	2	163
2	1	3	165

Array con una suddivisione alternativa

I	**J**	**K**
2	4	3

i	*j*	*k*	indirizzo (in base dieci)
0	0	0	119
0	0	1	121
0	0	2	123
0	1	0	125
0	1	1	127
0	1	2	129
0	2	0	131
0	2	1	133
0	2	2	135
0	3	0	137
0	3	1	139
0	3	2	141
1	0	0	143
1	0	1	145
1	0	2	147
1	1	0	149
1	1	1	151
1	1	2	153
1	2	0	155
1	2	1	157
1	2	2	159
1	3	0	161
1	3	1	163
1	3	2	165

Array con indici di accesso scambiati (ma il risultato è errato)

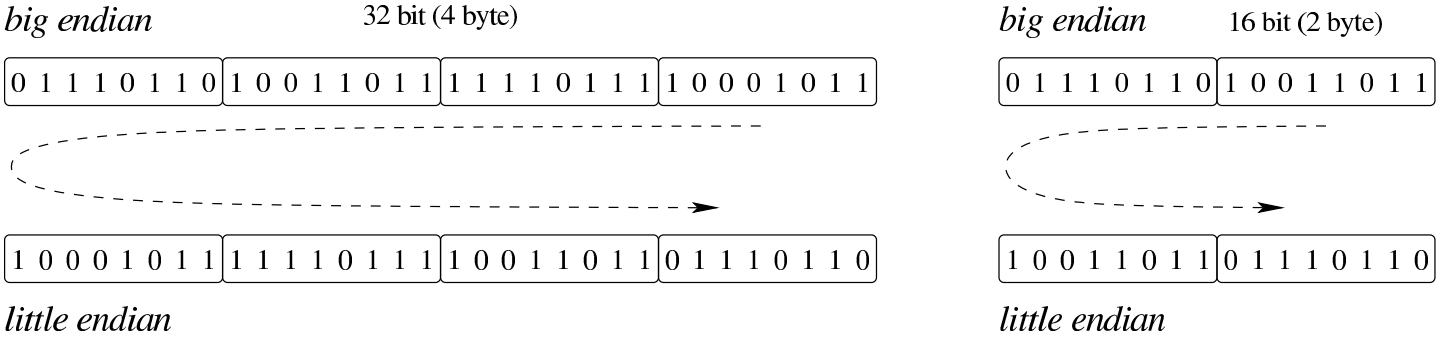
I	**J**	**K**
3	2	4

j	***k***	***i***	indirizzo errato (in base dieci)
0	0	0	119
0	0	1	121
0	0	2	123
0	1	0	127
0	1	1	129
0	1	2	131
0	2	0	135
0	2	1	137
0	2	2	139
0	3	0	143
0	3	1	145
0	3	2	147
1	0	0	135
1	0	1	137
1	0	2	139
1	1	0	143
1	1	1	145
1	1	2	147
1	2	0	151
1	2	1	153
1	2	2	155
1	3	0	159
1	3	1	161
1	3	2	163

63.1.6 Ordine dei byte

Come già descritto, normalmente l'accesso alla memoria avviene conoscendo l'indirizzo iniziale dell'informazione cercata, sapendo poi per quanti byte questa si estende. Il microprocessore, a seconda delle proprie caratteristiche e delle istruzioni ricevute, legge e scrive la memoria a gruppetti di byte, più o meno numerosi. Ma l'ordine dei byte che il microprocessore utilizza potrebbe essere diverso da quello che si immagina di solito.

Figura 63.18. Confronto tra *big endian* e *little endian*.



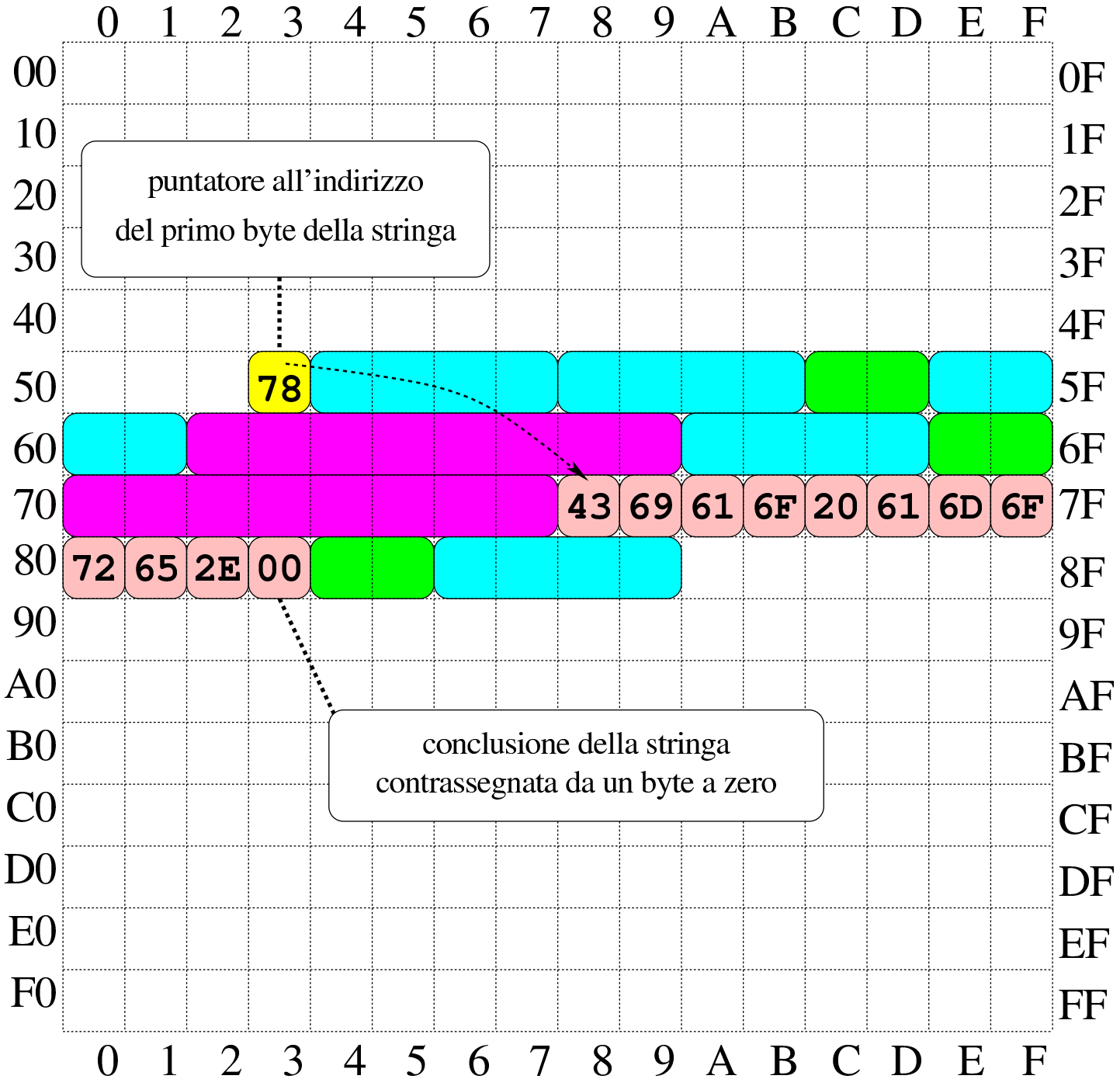
A questo proposito, per quanto riguarda la rappresentazione dei dati nella memoria, si distingue tra *big endian*, corrispondente a una rappresentazione «normale», dove il primo byte è quello più significativo (*big*), e *little endian*, dove la sequenza dei byte è invertita (ma i bit di ogni byte rimangono nella stessa sequenza standard) e il primo byte è quello meno significativo (*little*). La cosa importante da chiarire è che l'effetto dell'inversione nella sequenza porta a risultati differenti, a seconda della quantità di byte che compongono l'insieme letto o scritto simultaneamente dal microprocessore, come si vede nella figura.

63.1.7 Stringhe, array e puntatori

«

Le stringhe sono rappresentate in memoria come array di caratteri, dove il carattere può impiegare un byte o dimensioni multiple (nel caso di UTF-8, un carattere viene rappresentato utilizzando da uno a quattro byte, a seconda del punto di codifica raggiunto). Il riferimento a una stringa viene fatto come avviene per gli array in generale, attraverso un puntatore all'indirizzo della prima cella di memoria che lo contiene; tuttavia, per non dovere annotare la dimensione di tale array, di norma si conviene che la fine della stringa sia delimitata da un byte a zero, come si vede nell'esempio della figura.

Figura 63.19. Stringa conclusa da un byte a zero (*zero terminated string*), a cui viene fatto riferimento per mezzo di una variabile che contiene il suo indirizzo iniziale. La stringa contiene il testo '`Ciao amore.`', secondo la codifica ASCII.



Nella figura si vede che la variabile scalare collocata all'indirizzo 53_{16} contiene un valore da intendere come indirizzo, con il quale si

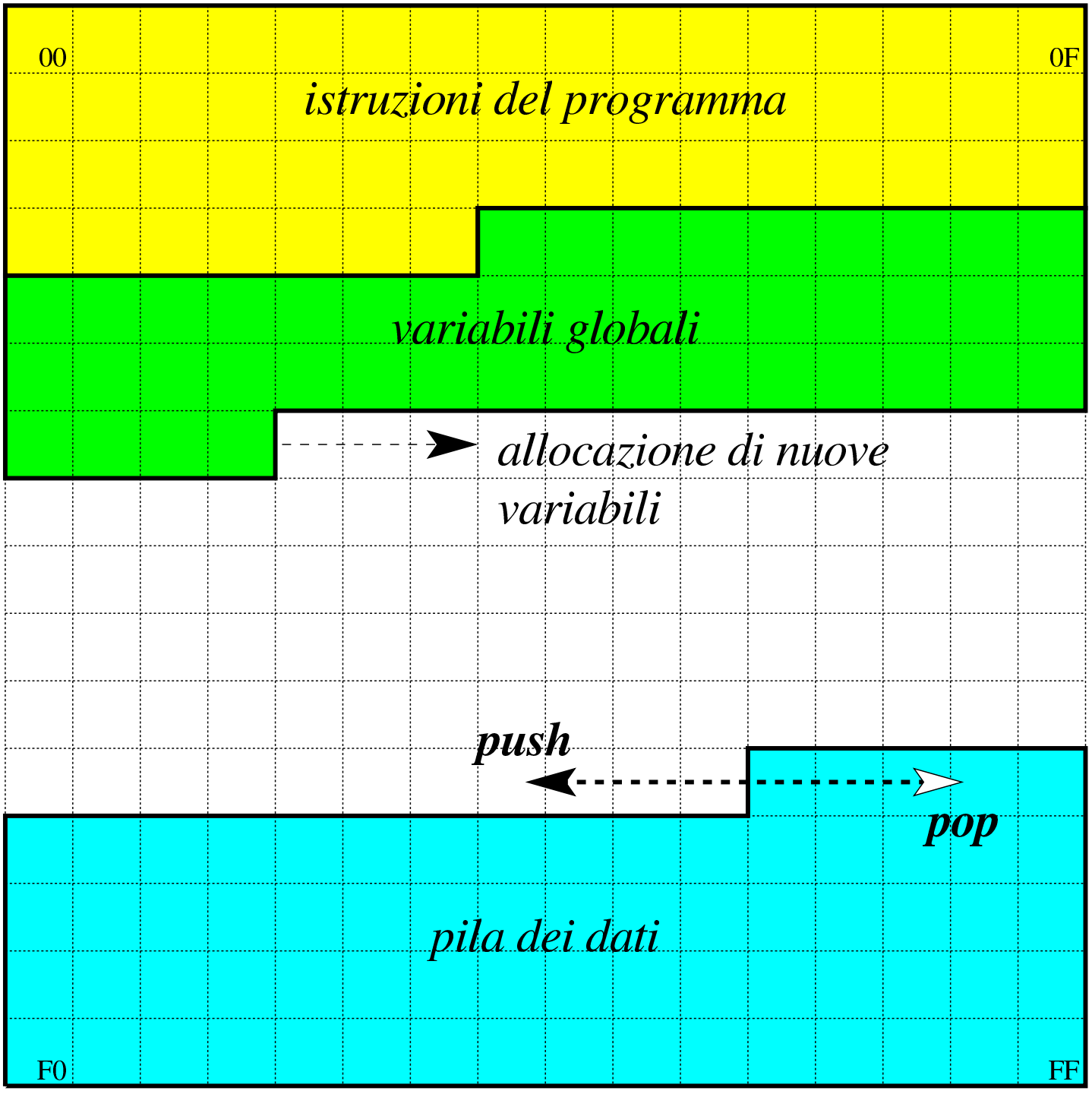
fa riferimento al primo byte dell'array che rappresenta la stringa (in posizione 78_{16}). La variabile collocata in 53_{16} assume così il ruolo di ***variabile puntatore*** e, secondo il modello ridotto di memoria della figura, è sufficiente un solo byte per rappresentare un tale puntatore, dal momento che servono soltanto valori da 00_{16} a FF_{16}.

63.1.8 Utilizzo della memoria

«

La memoria dell'elaboratore viene utilizzata sia per contenere i dati, sia per il codice del programma che li utilizza. Ogni programma ha un proprio spazio in memoria, che può essere reale o virtuale; all'interno di questo spazio, la disposizione delle varie componenti potrebbe essere differente. Nei sistemi che si rifanno al modello di Unix, nella parte più «bassa» della memoria risiede il codice che viene eseguito; subito dopo vengono le variabili globali del programma, mentre dalla parte più «alta» inizia la pila dei dati che cresce verso indirizzi inferiori. Si possono comunque immaginare combinazioni differenti di tale organizzazione, pur rispettando il vincolo di avere tre zone ben distinte per il loro contesto (codice, dati, pila); tuttavia, ci sono situazioni in cui i dati si trovano mescolati al codice, per qualche ragione.

Figura 63.20. Esempio di disposizione delle componenti di un programma in esecuzione in memoria, secondo il modello Unix.



63.2 Architettura, linguaggio, contesto virtuale, terminologia

«

Ciò che un microprocessore esegue sono istruzioni in linguaggio macchina, composte secondo la sintassi che il microprocessore stesso è in grado di interpretare. Il linguaggio macchina è fatto esclusivamente di numeri (da gestire in base due) e per questo, di norma, non viene utilizzato direttamente a livello umano.

Quando si deve intervenire al livello più basso possibile della programmazione, ci si avvale generalmente di un linguaggio «assemblatore» (*assembly*), ovvero un linguaggio che, pur rimanendo legato alle caratteristiche del microprocessore e in generale a quelle dell'architettura dell'elaboratore, esprime le istruzioni in una forma simbolica più comprensibile. Naturalmente, un programma scritto secondo un linguaggio assemblatore deve essere elaborato da un compilatore (*assembler*) per generare il linguaggio macchina effettivo.

Non esiste un'architettura standard, né un linguaggio macchina standard e di conseguenza non esiste nemmeno un linguaggio assemblatore standard. Un programma scritto con un linguaggio assemblatore adatto a un certo tipo di architettura non può funzionare in un'architettura differente. Pertanto, se si usa un tale linguaggio, lo si fa soprattutto per quelle cose che altrimenti non potrebbero essere risolte (come il codice necessario all'avvio del sistema operativo).

63.2.1 Memoria e registri

«

Le architetture per elaboratore più diffuse prevedono un microprocessore in grado di comunicare con una memoria centrale, organizzata come un vettore di celle, contenenti una quantità uniforme di

bit, accessibili attraverso un indice che ne rappresenta l'indirizzo. Oltre a questo, il microprocessore dispone internamente di alcuni ***registri***, ovvero delle celle singole di memoria con compiti più o meno specializzati.

La dimensione dei registri condiziona la capacità del microprocessore di eseguire dei calcoli e la capacità di indirizzamento della memoria. La dimensione dei registri più comuni di un microprocessore corrisponde alla dimensione della ***parola*** (*word*).

Generalmente, la memoria centrale è organizzata in celle di byte (intesi come gruppi di otto bit), ma possono esistere architetture in cui tali celle corrispondono alla dimensione della parola del microprocessore, o anche altre dimensioni, ma in generale una cella della memoria deve essere contenibile in un registro.

Nella memoria centrale devono risiedere sia i dati da elaborare, sia le istruzioni in linguaggio macchina. Pertanto, un registro molto importante in un microprocessore è quello che tiene traccia, nella memoria centrale, dell'istruzione successiva da eseguire: *instruction pointer* o *program counter*.

Nel caso degli elaboratori x86-32, l'architettura più comune negli anni 1990 prevede parole da 32 bit e una memoria organizzata in byte; pertanto è possibile gestire lo spazio di 4 Gibyte (2^{32}). Purtroppo, nella documentazione originale di questo tipo di architettura si usa il termine *word* per identificare una dimensione a 16 bit, come era nel primo microprocessore di quella serie (8088/8086), per motivi di compatibilità.

63.2.2 Indicatori o «flag»

Un ***indicatore***, ovvero un *flag*, è un'informazione costituita da un solo valore binario (*Vero* o *Falso*) che serve a tenere traccia dell'esito delle operazioni svolte all'interno del microprocessore. In generale, gli indicatori sono raccolti assieme in un registro specializzato.

Gli indicatori più importanti in assoluto sono due: «riporto» o *carry* che serve a conoscere l'esito delle somme (e delle sottrazioni) di valori senza segno; «traboccamento» o *overflow* che serve a conoscere l'esito delle somme (e delle sottrazioni) di valori con segno. Bisogna osservare che, tra le varie architetture, non è detto che gli indicatori funzionino sempre nella stessa maniera.

Tabella 63.21. Indicatori comuni tra le varie architetture.

Indicatore (*flag*)	Descrizione
carry	È l'indicatore del riporto che diventa utile per le operazioni con valori senza segno.
borrow	È l'indicatore della richiesta del prestito di una cifra nelle sottrazioni che diventa utile per le operazioni con valori senza segno. Di solito questo indicatore è costituito dallo stesso *carry*, il cui risultato va inteso in questo senso quando si eseguono delle sottrazioni.
overflow	È l'indicatore di traboccamento per le operazioni che riguardano valori con segno.
zero	Viene impostato dopo un'operazione che dà come risultato il valore zero.
sign	In linea di massima, riproduce il bit più significativo di un valore, dopo un'operazione. Se il valore è da intendersi con segno, l'indicatore serve a riprodurre il segno stesso.

Indicatore (*flag*)	Descrizione
parity	In linea di massima, si attiva quando l'ultima operazione produce un risultato contenente una quantità pari di bit a uno (ma ciò non significa che il valore corrispondente sia pari).

63.2.3 «Opcode»

Nel linguaggio macchina, il codice numerico che descrive le istruzioni è definito *operation code* (codice operazione) e si abbrevia come *opcode* (o solo «op»). La lunghezza complessiva dell'istruzione può cambiare a seconda degli operandi che il codice di operazione prevede di avere.

63.2.4 Accesso alla memoria

Le istruzioni fornite al microprocessore (in linguaggio macchina o secondo la simbologia del linguaggio assemblatore) contengono dati o riferimenti a dei dati. A questo proposito, ogni architettura definisce le proprie tipologie e, di conseguenza, non esiste una denominazione uniforme.

Spesso si distingue tra le modalità di indirizzamento riferite al codice del programma, rispetto a quelle relative ai dati veri e propri. Le forme di indirizzamento più semplici riferite al codice possono essere assolute, quando si specifica un indirizzo preciso, oppure relative, quando si specifica uno spostamento relativo dalla posizione corrente. Si usa l'indirizzamento al codice con i salti e con le chiamate di subroutine. L'indirizzamento ai dati potrebbe comprendere le forme dell'elenco seguente:

- valori numerici costanti, incorporati nell'istruzione, che spesso sono chiamati «immediati»;
- registri da intendere come tali;
- aree di memoria indicate direttamente con un indirizzo;
- aree di memoria indicate da un indirizzo composto da un valore di riferimento e l'aggiunta di un indice o di uno scostamento;
- aree di memoria indicate attraverso un registro che ne contiene l'indirizzo;
- aree di memoria indicate con un indirizzo composto dalla somma tra un valore costante, il contenuto di un registro ed eventualmente uno scostamento.

In generale, il termine «valore immediato» si riferisce a un'informazione numerica costante, incorporata nell'istruzione in linguaggio macchina. Ogni volta che si indica un riferimento fisso alla memoria (di solito lo si fa attraverso un «simbolo», rappresentato da etichetta, che il compilatore traduce in un indirizzo, in uno scostamento o in un dislocamento), sia per ciò che riguarda il codice, sia per i dati veri e propri, si sta utilizzando un valore immediato, anche se è compito del compilatore tradurlo effettivamente in un numero. Tale valore è «immediato» in quanto il microprocessore non deve eseguire alcun calcolo per interpretarlo.

Come si può intuire, le forme più complesse di rappresentazione delle variabili in memoria consentono una scansione utile per rappresentare gli array di dati.

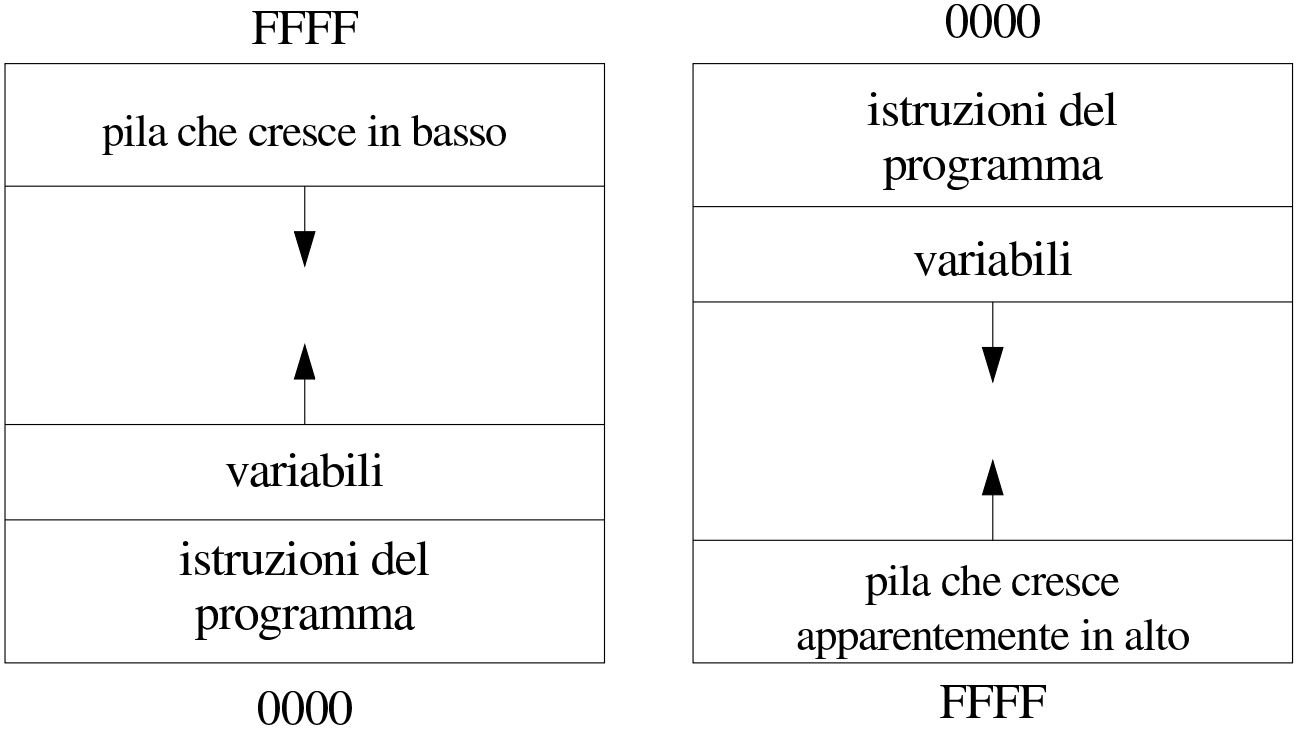
È però importante distinguere i contesti: un conto è l'istruzione macchina, un altro è l'istruzione scritta nel linguaggio assemblatore. Generalmente gli indirizzi della memoria non vengono scritti materialmente in forma numerica, lasciando che sia il compilatore a tradurli nella realtà concreta. Ma questo comporta spesso anche la scelta di un tipo di istruzione macchina rispetto a un altro, in base al contesto, cosa che rimane sempre a carico del compilatore. D'altro canto, certi tipi di indirizzamento complesso, vengono elaborati e semplificati dal compilatore stesso.

È importante sottolineare che le istruzioni in linguaggio macchina (*opcode*) possono essere diverse, anche se riferite a uno stesso tipo di operazione, quando cambia l'entità di un dislocamento o il tipo di indirizzamento; pertanto, quando si legge il manuale di riferimento per un certo microprocessore, si trova l'elenco delle istruzioni e la descrizione degli operandi previsti, ma non è detto che nel linguaggio assemblatore si debba usare esattamente la stessa modalità.

63.2.5 Modello della memoria nei sistemi Unix[1]

Nei sistemi Unix, inclusi i sistemi liberi che si rifanno a quel modello tradizionale, i processi elaborativi vedono la memoria come un solo vettore contenente: le istruzioni da eseguire, lo spazio previsto per i dati e una pila, utilizzata sostanzialmente nel modo descritto nelle sezioni precedenti. La pila inizia però da una posizione elevata di questo vettore e si espande in posizioni inferiori. Pertanto, l'indice della pila viene decrementato quando la si carica di un nuovo elemento (*push*) e viene incrementato quando invece la si scarica (*pop*). Ciò è come dire che la pila è rovesciata e si estende «verso il basso».

Figura 63.22. Semplificazione del modo in cui un processo elaborativo Unix vede la memoria. La dimensione della memoria virtuale a disposizione di un processo elaborativo dipende normalmente dall'architettura dell'elaboratore; il valore indicato nel disegno serve solo come semplificazione. A sinistra si vedono gli indirizzi di memoria partire dal basso ed estendersi in altro; a destra si vede l'opposto. Nella seconda forma visuale la pila cresce «dal basso», ma rimane il fatto che il modo di gestire l'indice sia lo stesso.



La rappresentazione che si vede nella parte sinistra della figura è quella tradizionale, ma se si ragiona «in senso di lettura», potrebbe essere più logico rappresentare gli indirizzi più bassi in altro, progredendo verso il basso. In tal caso, la pila si estende come si è abituati normalmente a pensarla, ma resta il fatto che l'indice di gestione della pila deve essere decrementato per aggiungere degli elementi sulla stessa.

63.2.6 Sintassi «AT&T» e «Intel»

Quando si utilizza l'architettura x86 si trovano generalmente compilatori per linguaggi assemblatori di due tipi: uno conforme allo stile usato nei sistemi Unix del PDP-11; l'altro conforme alla simbologia usata dalla documentazione della casa produttrice dei primi microprocessori di questo tipo. Dal momento che Unix è nato nei laboratori Bell AT&T, la prima notazione è nota come «sintassi AT&T»; per converso, l'altra è la «sintassi Intel».

Generalmente, negli ambienti legati ai sistemi Unix e simili, GNU/Linux incluso, si preferisce usare compilatori con sintassi AT&T.

63.2.7 Macchina virtuale

Quando si scrive un programma in linguaggio assemblatore, occorre tenere in considerazione il contesto di funzionamento. Di norma questo contesto è dato dal sistema operativo, attraverso il quale il programma viene caricato in memoria e poi eseguito.

In effetti, l'uso diretto di un linguaggio assemblatore è appropriato quando si opera al di fuori del sistema operativo, per esempio, proprio per il codice di avvio di un sistema. Tuttavia, quando si inizia lo studio di un tale linguaggio, i programmi che si realizzano sono fatti per un sistema già funzionante che quindi si sottomettono al controllo di questo.

L'ambiente in cui si trova a funzionare il programma avviato attraverso il sistema operativo è una macchina virtuale che può avere caratteristiche differenti rispetto alla «macchina reale», soprattutto per

ciò che riguarda l'indirizzamento della memoria e per le funzioni a cui è possibile accedere.

63.2.8 Compilazione e collegamento

«

Nei sistemi operativi che si rifanno al modello di Unix, la compilazione di un programma scritto secondo un linguaggio assemblatore segue un procedimento comune. Una prima fase interpreta un file sorgente e produce un file «oggetto», ripetendo eventualmente il procedimento per altri file che servono a produrre lo stesso programma. I file oggetto ottenuti in questa fase sono file binari che non sono ancora pronti per essere eseguiti, in quanto alcune informazioni sono rimaste da definire. Nella seconda fase (nota come *link*) i file oggetto che servono a comporre un certo programma vengono collegati assieme, generando il file eseguibile vero e proprio.

In pratica, un programma eseguibile viene ottenuto da almeno un file oggetto, ma spesso i file oggetto che servono a produrre un programma sono più di uno. Infatti, nei file che costituiscono i sorgenti possono esserci dei riferimenti a zone di memoria e a funzioni descritte in altri file, pertanto è compito della fase di «collegamento» il comporre assieme i file oggetto in modo che questi riferimenti reciproci vengano consolidati.

Secondo la tradizione, in modo predefinito, il compilatore di un linguaggio assemblatore genera il file oggetto con il nome '`a.out`', ma anche il *linker*, ovvero il programma che collega assieme i file oggetto, creerebbe un file eseguibile con lo stesso nome (naturalmente, di solito si dichiara esplicitamente il nome del file da generare). È bene sapere che il nome «a.out» deriva dalle primissime edizioni di Unix e significa *Assembler output*.

Quando si usano linguaggi di programmazione più evoluti rispetto al codice che si rifà direttamente alle caratteristiche del microprocessore, spesso il procedimento di compilazione passa per la produzione di un sorgente in linguaggio assemblatore, che poi viene compilato secondo la modalità consueta. In ogni caso, se la compilazione prevede la produzione intermedia di file oggetto, teoricamente, questi possono essere collegati assieme ad altri file oggetto prodotti da altri linguaggi. Perché ciò sia possibile effettivamente, occorre comunque che siano compatibili nel modo di condividere la memoria e di eseguire le chiamate delle funzioni, al livello del linguaggio macchina.

Rimane da tenere presente che i file oggetto e i file eseguibili hanno un formato definito da un certo standard. Di questi standard ne esistono molti, anche se nei sistemi Unix e simili si è affermato prevalentemente il formato ELF (*Executable and linkable format*). I primi formati usati nei sistemi Unix sono noti con come «a.out», confondendosi con il nome del file generato in modo predefinito dal compilatore. Si osservi che i compilatori attuali, in mancanza di altre indicazioni, producono file con il nome ‘`a.out`’, indipendentemente dal formato che questi hanno, formato che può benissimo essere ELF o altro.

63.3 Rappresentazione di valori numerici

La memoria di un elaboratore consente di annotare esclusivamente delle cifre binarie e in uno spazio di dimensione prestabilita e fissa. Nelle sezioni successive si descrivono alcune forme di rappresentazione dei valori numerici, nell’ambito di queste limitazioni.

63.3.1 Codifica delle singole cifre

«

Un valore numerico potrebbe essere espresso come una stringa di caratteri, corrispondenti alle cifre numeriche che lo rappresentano secondo la notazione in base dieci. Naturalmente, una rappresentazione del genere implica uno spreco di spazio nel sistema di memorizzazione e richiede una trasformazione prima di poter procedere all'esecuzione di calcoli numerici.

Esistono diverse forme di rappresentazioni numeriche, intese come sequenze di cifre in base dieci, che utilizzano quattro bit per ogni cifra. Il sistema più comune è noto con il nome BCD: *Binary coded decimal*.

I sistemi di rappresentazione numerica che utilizzano quattro bit per ogni cifra di un valore in base dieci, si utilizzano per esempio nel linguaggio COBOL, per le variabili scalari di tipo *computational*.

Tabella 63.23. Alcune codifiche per la rappresentazione di cifre numeriche (in base dieci) a gruppi di quattro bit.

Cifra decimale	Codice BCD (8421)	Codice «eccesso 3»	Codice 2421	Codice 5211	Codice 631-1	Codice 732-1
0	0000_2	0011_2	0000_2	0000_2	0000_2 0011_2	0000_2
1	0001_2	0100_2	0001_2	0001_2 0010_2	0010_2	0011_2
2	0010_2	0101_2	0010_2 1000_2	0100_2 0011_2	0101_2	0010_2
3	0011_2	0110_2	0011_2 1001_2	0101_2 0110_2	0100_2	0100_2

Cifra decimale	Codice BCD (8421)	Codice «ec-cesso 3»	Codice 2421	Codice 5211	Codice 631-1	Codice 732-1
4	0100_2	0111_2	0100_2 1010_2	0111_2	0110_2	0111_2
5	0101_2	1000_2	0101_2 1011_2	1000_2	1001_2	0110_2
6	0110_2	1001_2	0110_2 1100_2	1010_2 1001_2	1000_2	1001_2
7	0111_2	1010_2	0111_2 1101_2	1011_2 1100_2	1010_2	1000_2
8	1000_2	1011_2	1110_2	1110_2 1101_2	1101_2	1011_2
9	1001_2	1100_2	1111_2	1111_2	1100_2 1111_2	1010_2

La codifica BCD e altre sono ***codici pesati***, in quanto a ogni bit viene attribuito un peso, da sommare per determinare il valore. I pesi per la codifica BCD sono 8, 4, 2 e 1; pertanto, il codice BCD 1001_2 corrisponde a 1*8+0*4+0*2+1*1 = 9. Nella tabella riepilogativa, i codici pesati sono: BCD, 2421, 5211, 631-1 e 732-1. I nomi usati per questi codici sono costituiti dalla sequenza dei pesi stessi.

Alcuni codici pesati prevedono la rappresentazione di qualche cifra in più in un modo alternativo. Per esempio, nel codice 2421, il numero due si può ottenere sia come 1000_2, sia come 0010_2.

I codici pesati come BCD (ovvero 8421), 2421 e 5211, prevedono pesi positivi; i codici come 631-1 e 732-1, prevedono anche pesi negativi. Per esempio, con il codice 732-1, si ottiene il valore uno con il codice 0011_2, perché il secondo bit (a destra) vale come il numero due, mentre il primo bit (a destra) sottrae una unità.

Dei codici che appaiono nella tabella, il codice a eccesso tre, non è un codice pesato, in quanto corrisponde al codice BCD, a cui si aggiunge il valore tre.

È necessario sottolineare che il codice BCD, a seconda del contesto, può essere riferito anche a un codice a otto bit, dove i primi quattro, più significativi, sono posti a zero.

63.3.2 Rappresentazione binaria di numeri interi senza segno

«

Quando si rappresentano dei valori numerici in forma binaria, senza passare per una conversione di ogni singola cifra decimale, si usa tutta la sequenza di bit per il valore. La rappresentazione di un valore intero senza segno coincide normalmente con il valore binario contenuto nella variabile. Pertanto, una variabile della dimensione di 8 bit, può rappresentare valori da zero a 2^8-1:

00000000_2 (0_{10})
00000001_2 (1_{10})
00000010_2 (2_{10})
...
11111110_2 (254_{10})
11111111_2 (255_{10})

63.3.3 Rappresentazioni binarie superate di numeri interi con segno

«

Un numero binario, inserito nella memoria di un elaboratore, può contenere esclusivamente cifre numeriche; pertanto, la rappresen-

tazione del segno può avvenire solo attraverso cifre numeriche. A partire approssimativamente dal 1965, il segno di un numero intero si rappresenta attraverso il complemento alla base (complemento a due) che ha delle proprietà importanti, ma per comprenderle occorre vedere quali sono state le alternative precedenti.

Il primo modo utilizzato per rappresentare un numero intero con segno è stato quello di attribuire a un bit (probabilmente quello più significativo) il ruolo di indicatore del segno. Per esempio, 00001010_2 andrebbe interpretato come $+10_{10}$, mentre 10001010_2 rappresenterebbe il valore -10_{10}. In questo modo, disponendo di otto cifre binarie, dovendone riservare una per il segno, si potrebbero rappresentare valori da -127 (11111111_2) a $+127$ (01111111_2); inoltre, lo zero potrebbe essere rappresentato indifferentemente come 00000000_2 o come 1000000_2.

Il secondo metodo (usato per esempio nel PDP-1) prevede la rappresentazione dei numeri negativi come complemento a uno del valore positivo corrispondente. Il complemento a uno si ottiene invertendo le cifre del numero binario. In questo modo, per esempio, 00001010_2 rappresenterebbe sempre il valore $+10_{10}$, mentre 11110101_2 corrisponderebbe a -10_{10}. Anche in questo caso la prima cifra rappresenta il segno (dove la cifra uno indica un valore negativo), ma il segno si aggiorna automaticamente con la semplice inversione del valore. Utilizzando il complemento a uno per rappresentare i valori negativi, come nel caso precedente, su otto cifre complessive si possono indicare valori da -127 a $+127$ e lo zero si può rappresentare ancora in due modi differenti: 00000000_2 o 11111111_2.

Disponendo di una variabile per rappresentare valori interi con segno, considerato che il bit più significativo serve a rappresentare il segno stesso, si dispone di un bit in meno per indicare il valore. Pertanto, se si dispone di n bit, si possono rappresentare valori fino a $n-1$ bit, ovvero valori fino a $2^{(n-1)}-1$. Per i numeri negativi, il calcolo è lo stesso, anche se si considera che si fa riferimento a valori complementati: si può rappresentare fino a $-(2^{(n-1)}-1)$.

Il complemento alla base (ovvero il complemento a due) che è invece il metodo attuale per rappresentare i valori interi negativi, ha i vantaggi del metodo del complemento a uno, ma in più ha un solo modo per rappresentare lo zero.

63.3.4 Complemento a due

«

Attualmente, per rappresentare valori interi con segno (positivo o negativo), si utilizza il metodo del complemento alla base, ovvero del complemento a due, dove il primo bit indica sempre il segno. Il complemento a due si ottiene facilmente calcolando prima il complemento a uno e poi aggiungendo una unità al risultato. Per esempio, se si prende un valore positivo rappresentato in otto cifre binarie come 00010100_2 (pari a $+20_{10}$), il complemento a uno è 11101011_2; aggiungendo una unità si ottiene il complemento a due: 11101100_2.

Utilizzando questo metodo, per cambiare di segno a un valore è sufficiente calcolarne il complemento a due (esattamente come si farebbe con il metodo del complemento a uno). Lo si verifica facilmente: riprendendo l'esempio già fatto, partendo da -20_{10} che si rappresenta come 11101100_2, si calcola prima il complemento a uno, ottenendo

così 00010011_2, quindi si somma una unità e si ottiene 00010100_2, pari a $+20_{10}$.

Con il complemento a due, disponendo di ***n*** cifre binarie, si possono rappresentare valori da $-2^{(n-1)}$ a $+2^{(n-1)}-1$ ed esiste un solo modo per rappresentare lo zero: quando tutte le cifre binarie sono pari a zero. Infatti, rimanendo nell'ipotesi di otto cifre binarie, il complemento a uno di 00000000_2 è 11111111_2, ma aggiungendo una unità per ottenere il complemento a due si ottiene di nuovo 00000000_2, perdendo il riporto.

Si osservi che il valore negativo più grande rappresentabile non può essere trasformato in un valore positivo corrispondente, perché si creerebbe un traboccamento. Per esempio, utilizzando sempre otto bit (segno incluso), il valore minimo che possa essere rappresentato è 1000000_2, pari a -128_{10}, ma se si calcola il complemento a due, si ottiene di nuovo lo stesso valore binario, che però non è valido. Infatti, il valore positivo massimo che si possa rappresentare in questo caso è solo $+127_{10}$.

Figura 63.24. Confronto tra due valori interi con segno.

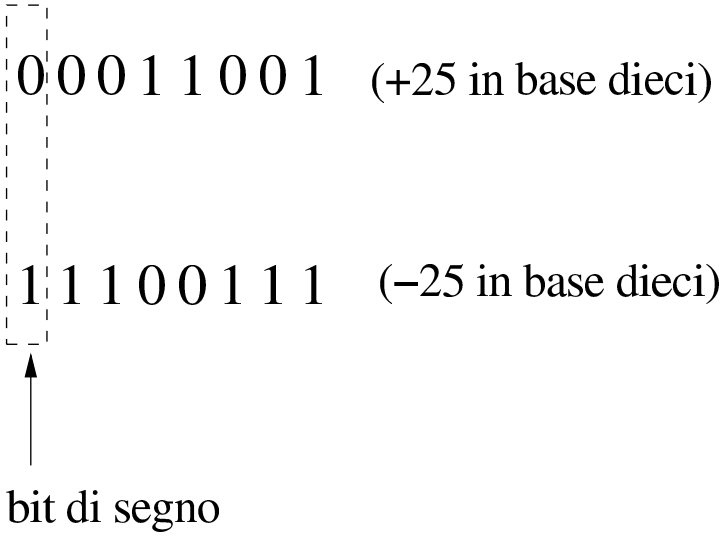
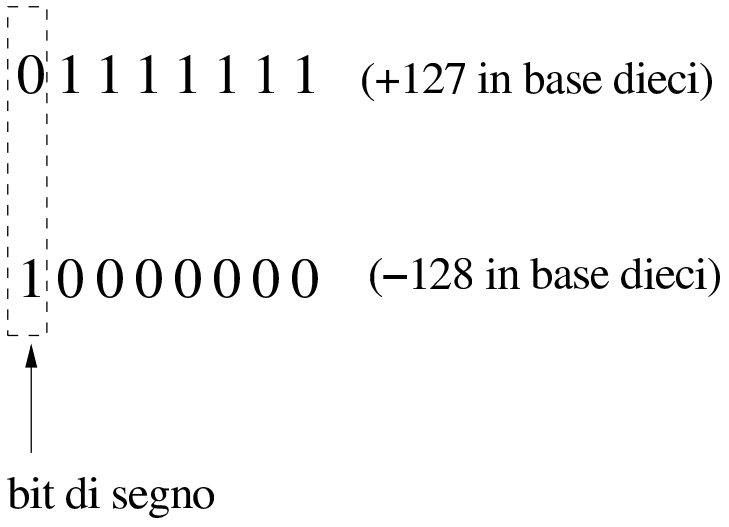


Figura 63.25. Valori massimi rappresentabili con soli otto bit.



Il meccanismo del complemento a due ha il vantaggio di trasformare la sottrazione in una semplice somma algebrica.

63.3.5 Rappresentazione binaria di numeri in virgola mobile

«

Una forma diffusa per rappresentare dei valori molto grandi, consiste nell'indicare un numero con dei decimali moltiplicato per un valore costante elevato a un esponente intero. Per esempio, per rappresentare il numero 123000000 si potrebbe scrivere $123 \cdot 10^6$, oppure anche $0{,}123 \cdot 10^9$. Lo stesso ragionamento vale anche per valori molti piccoli; per esempio 0,000000123 che si potrebbe esprimere come $0{,}123 \cdot 10^{-6}$.

Per usare una notazione uniforme, si può convenire di indicare il numero che appare prima della moltiplicazione per la costante elevata a una certa potenza come un valore che più si avvicina all'unità, essendo minore o al massimo uguale a uno. Pertanto, per gli esempi già mostrati, si tratterebbe sempre di $0{,}123 \cdot 10^n$.

Per rappresentare valori a ***virgola mobile*** in modo binario, si usa

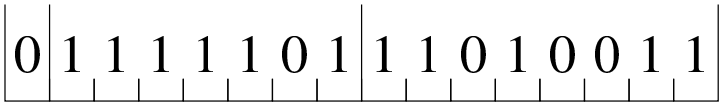
un sistema simile, dove i bit a disposizione della variabile vengono suddivisi in tre parti: segno, esponente (di una base prestabilita) e mantissa, come nell'esempio che appare nella figura successiva.[2]

Figura 63.26. Ipotesi di una variabile a 16 bit per rappresentare dei numeri a virgola mobile.

Nella figura si ipotizza la gestione di una variabile a 16 bit per la rappresentazione di valori a virgola mobile. Come si vede dallo schema, il bit più significativo della variabile viene utilizzato per rappresentare il segno del numero; i sette bit successivi si usano per indicare l'esponente (con segno) e gli otto bit finali per la mantissa (senza segno perché indicato nel primo bit), ovvero il valore da moltiplicare per una certa costante elevata all'esponente.

Quello che manca da decidere è come deve essere interpretato il numero della mantissa e qual è il valore della costante da elevare all'esponente indicato. Sempre a titolo di esempio, si conviene che il valore indicato nella mantissa esprima precisamente «0,***mantissa***» e che la costante da elevare all'esponente indicato sia 16 (ovvero 2^4), che si traduce in pratica nello spostamento della virgola di quattro cifre binarie alla volta.[3]

Figura 63.27. Esempio di rappresentazione del numero 0,051513671875 ($211 \cdot 16^{-3}$), secondo le convenzioni stabilite. Si osservi che il valore dell'esponente è negativo ed è così rappresentato come complemento alla base (due) del valore assoluto relativo.



$$+211 \cdot 16^{-3}$$

0,0000000000011010011

Naturalmente, le convenzioni possono essere cambiate: per esempio il segno lo si può incorporare nella mantissa; si può rappresentare l'esponente attraverso un numero al quale deve essere sottratta una costante fissa; si può stabilire un valore diverso della costante da elevare all'esponente; si possono distribuire diversamente gli spazi assegnati all'esponente e alla mantissa.

63.3.6 Rappresentazione in virgola mobile IEEE 754

«

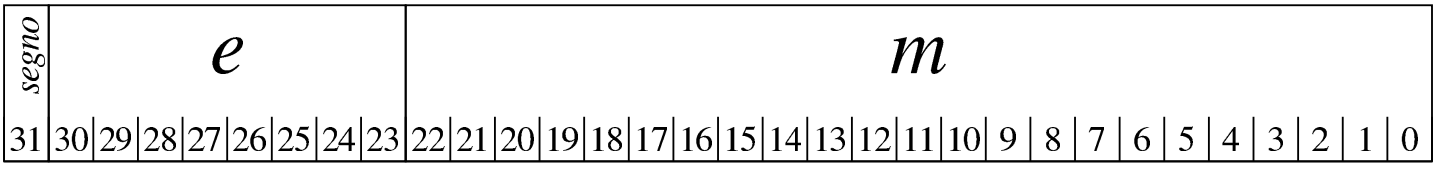
Per la rappresentazione dei valori in virgola mobile esiste uno standard importante, IEEE 754 (ripreso anche da altri enti di standardizzazione), con il quale si definiscono due formati, per la precisione singola e doppia. Secondo questo standard, un valore in virgola mobile a precisione singola richiede 32 bit, mentre per la precisione doppia sono necessari 64 bit. Per prima cosa si definisce un «numero normalizzato», corrispondente a:[4]

$$1,\boldsymbol{significante}_2 \times 2^{\boldsymbol{esponente}}$$

Di questo si utilizza solo il significante (mantissa) e l'esponente (caratteristica), omettendo il numero uno iniziale. Nella forma prevista

dallo standard IEEE 754 si annota separatamente il segno del numero, quindi l'esponente, che però è «polarizzato» (nel senso che al valore dell'esponente originario occorre sommare un certo valore fisso), quindi si mettono le cifre del significante (tutte quelle che possono starci). Si osservi che il significante viene indicato sempre come valore assoluto, pertanto non si applica il complemento per i valori negativi; inoltre, l'esponente viene indicato sommando al valore originale un numero fisso che è costituito da tutti i bit a uno, tranne quello più significativo (quando l'esponente è formato da otto bit, il numero da sommare è 01111111_2, pari a 127_{10}; quando l'esponente è formato da 11 bit, il numero da sommare è 01111111111_2, pari a 1023_{10}).

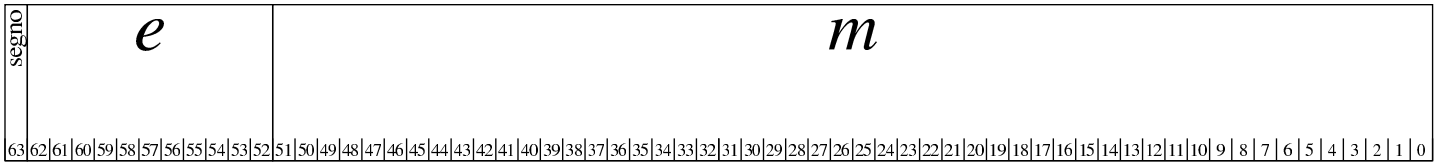
Figura 63.28. IEEE 754 a precisione singola.



bit 31	(1)	segno: 0 = positivo; 1 = negativo
bit 23–30	(8)	esponente in eccesso 127 = esponente originale + 127
bit 0–22	(23)	significante (mantissa)

e = 0	*m = 0*	zero, che può essere positivo o negativo
e = 0	*m <> 0*	numero «denormalizzato»
e = 255	*m = 0*	infinito, che può essere positivo o negativo
e = 255	*m <> 0*	indefinito

Figura 63.29. IEEE 754 a precisione doppia.



bit 63 (1) segno: 0 = positivo; 1 = negativo
bit 52–62 (11) esponente in eccesso 1023 = esponente originale + 1023
bit 0–51 (52) significante (mantissa)

e = 0	*m = 0*	zero, che può essere positivo o negativo
e = 0	*m <> 0*	numero «denormalizzato»
e = 2047	*m = 0*	infinito, che può essere positivo o negativo
e = 2047	*m <> 0*	indefinito

Conviene fare un esempio con la precisione singola: si vuole rappresentare il valore $21{,}11_{10}$. Si procede convertendo separatamente la parte intera e poi quella decimale. Il numero 21_{10} si converte facilmente in 10101_2, mentre per la parte decimale occorre fare qualche calcolo in più:

$0{,}11 \times 2 = 0{,}22 \rightarrow {,}0_2$
$0{,}22 \times 2 = 0{,}44 \rightarrow {,}00_2$
$0{,}44 \times 2 = 0{,}88 \rightarrow {,}000_2$
$0{,}88 \times 2 = 1{,}76 \rightarrow {,}0001_2$
$0{,}76 \times 2 = 1{,}52 \rightarrow {,}00011_2$
$0{,}52 \times 2 = 1{,}04 \rightarrow {,}000111_2$
$0{,}04 \times 2 = 0{,}08 \rightarrow {,}0001110_2$
$0{,}08 \times 2 = 0{,}16 \rightarrow {,}00011100_2$
$0{,}16 \times 2 = 0{,}32 \rightarrow {,}000111000_2$
$0{,}32 \times 2 = 0{,}64 \rightarrow {,}0001110000_2$
$0{,}64 \times 2 = 1{,}28 \rightarrow {,}00011100001_2$
$0{,}28 \times 2 = 0{,}56 \rightarrow {,}000111000010_2$
$0{,}56 \times 2 = 1{,}12 \rightarrow {,}0001110000101_2$

$0{,}12 \times 2 = 0{,}24 \rightarrow {,}000111000010{10}_2$
$0{,}24 \times 2 = 0{,}48 \rightarrow {,}0001110000101{00}_2$
$0{,}48 \times 2 = 0{,}96 \rightarrow {,}00011100001010{00}_2$
$0{,}96 \times 2 = 1{,}92 \rightarrow {,}000111000010100{01}_2$
$0{,}92 \times 2 = 1{,}84 \rightarrow {,}000111000010100{011}_2$
$0{,}84 \times 2 = 1{,}68 \rightarrow {,}0001110000101000{111}_2$
$0{,}68 \times 2 = 1{,}36 \rightarrow {,}0001110000101000{1111}_2$
$0{,}36 \times 2 = 0{,}72 \rightarrow {,}00011100001010001{1110}_2$
...

Pertanto, $21{,}11_{10}$ corrisponde approssimativamente a $10101{,}00011100001010001{1110}_2$. Per normalizzare il numero occorre spostare la virgola a sinistra e moltiplicare per una potenza di due: $1{,}01010001110000101000{1110}_2 \times 2^4$.

A questo punto si prelevano 23 cifre dopo la virgola, ma si richiede un arrotondamento (in questo caso avviene per eccesso), pertanto le cifre che compongono il significante diventano: $01010001110000101001000_2$. L'esponente va sommato al valore costante stabilito: 4+127 = 131. Quindi l'esponente si rappresenta così: 10000011_2. Trattandosi di un numero positivo, il bit del segno deve essere zero. Ecco il numero in virgola mobile, a precisione singola, espresso secondo la notazione standard (gli spazi aggiunti servono a facilitarne la lettura):

$0\ 10000011\ 01010001110000101001000_2$

Con questo metodo, un numero a precisione singola può avere un valore assoluto da $1_2 \times 2^{-126}$ a $1{,}1111\ldots_2 \times 2^{+127}$; mentre un numero a precisione doppia può avere un valore assoluto da $1_2 \times 2^{-1022}$ a $1{,}1111\ldots_2 \times 2^{+1023}$. Quando si devono rappresentare va-

lori molto bassi, si azzerano i bit dell'esponente e si usa una forma «denormalizzata» di questo tipo per la precisione singola:

$0{,}\boldsymbol{significante}_2 \times 2^{-127}$

Per la precisione doppia:

$0{,}\boldsymbol{significante}_2 \times 2^{-1024}$

63.3.7 Ordine dei byte[5]

«

Generalmente si distinguono i microprocessori in base a una caratteristica legata al modo di ordinare i bit di un numero, presi a gruppi di otto. In pratica, di norma la memoria centrale degli elaboratori è organizzata a celle di otto bit (un byte), mentre il microprocessore è in grado di elaborare dati numerici con una quantità di bit maggiore (ma sempre multipli di otto). Nel momento in cui il microprocessore accede alla memoria centrale per leggere o scrivere un valore, lo fa secondo un ordine che dipende dalla sua progettazione.

Supponendo di avere a che fare con il valore $13579BDF_{16}$, se il microprocessore lo memorizza secondo la stessa sequenza (ovvero memorizza i byte 13_{16}, 57_{16}, $9B_{16}$ e DF_{16}), allora si dice che la sua architettura è *big endian*; diversamente, se il microprocessore memorizza invertendo la sequenza di byte (quindi DF_{16}, $9B_{16}$, 57_{16} e 13_{16}) si dice che questo lavora in modalità *little endian*.

Naturalmente, il microprocessore che scrive in memoria un valore secondo una sequenza di byte invertita, quando va a leggerlo dalla memoria si aspetta di trovarlo invertito nello stesso modo.

Sia chiaro che, all'interno di ogni byte, l'ordine dei bit non viene modificato. Inoltre, nel momento in cui si pensa a un'elaborazione all'interno del microprocessore, con dati contenuti nei suoi registri, non ha importanza conoscere qual è l'ordine dei byte.

63.4 Calcoli con i valori binari rappresentati nella forma usata negli elaboratori

Una volta chiarito il modo in cui si rappresentano comunemente i valori numerici elaborati da un microprocessore, in particolare per ciò che riguarda i valori negativi con il complemento a due, occorre conoscere in che modo si trattano o si possono trattare questi dati (indipendentemente dall'ordine dei byte usato).

Questi concetti tornano utili nella programmazione in linguaggio macchina o nei linguaggi assemblatori equivalenti, ma servono anche per linguaggi evoluti che conservano una rappresentazione dei valori conforme all'architettura dell'elaboratore.

63.4.1 Modifica della quantità di cifre di un numero binario intero

Un numero intero senza segno, espresso con una certa quantità di cifre, può essere trasformato in una quantità di cifre maggiore, aggiungendo degli zeri nella parte più significativa. Per esempio, il numero 0101_2 può essere trasformato in 00000101_2 senza cambiarne il valore. Nello stesso modo, si può fare una copia di un valore in un contenitore più piccolo, perdendo le cifre più significative, purché queste siano a zero, altrimenti il valore risultante sarebbe alterato.

Quando si ha a che fare con valori interi con segno, nel caso di valori positivi, l'estensione e la riduzione funzionano come per i valori senza segno, con la differenza che nella riduzione di cifre, la prima deve ancora rappresentare un segno positivo. Se invece si ha a che fare con valori negativi, l'aumento di cifre richiede l'aggiunta di cifre a uno nella parte più significativa, mentre la riduzione comporta l'eliminazione di cifre a uno nella parte più significativa, con il vincolo di mantenere inalterato il segno.

Figura 63.30. Aumento e riduzione delle cifre di un numero intero senza segno.

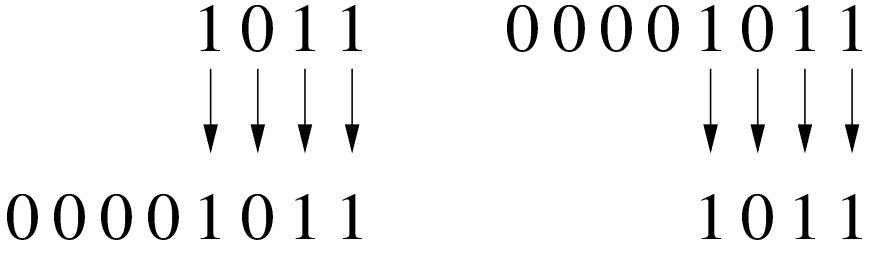


Figura 63.31. Aumento e riduzione delle cifre di un numero intero positivo.

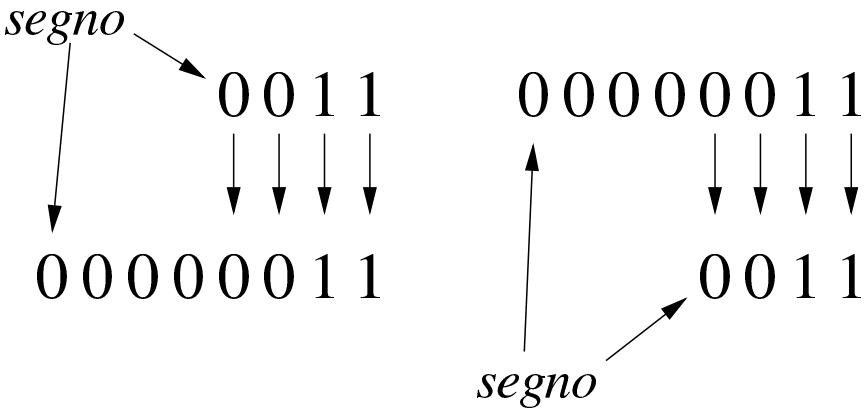
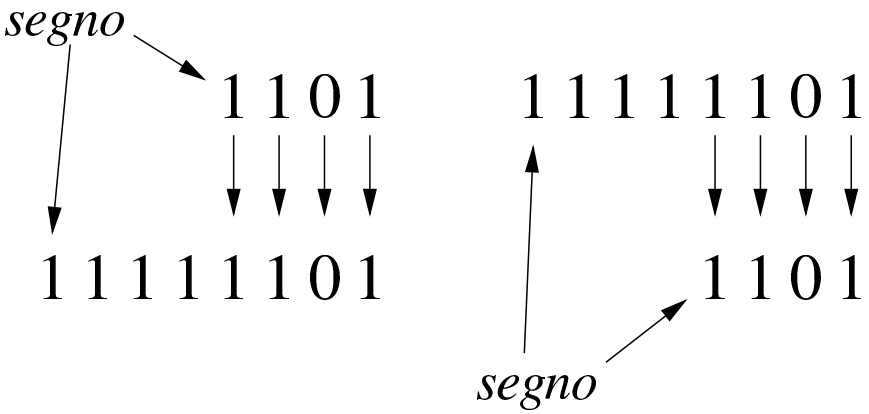


Figura 63.32. Aumento e riduzione delle cifre di un numero intero negativo.



63.4.2 Sommatorie con i valori interi con segno

Vengono proposti alcuni esempi che servono a dimostrare le situazioni che si presentano quando si sommano valori con segno, ricordando che i valori negativi sono rappresentati come complemento alla base del valore assoluto corrispondente.

Figura 63.33. Somma di due valori positivi che genera un risultato valido.

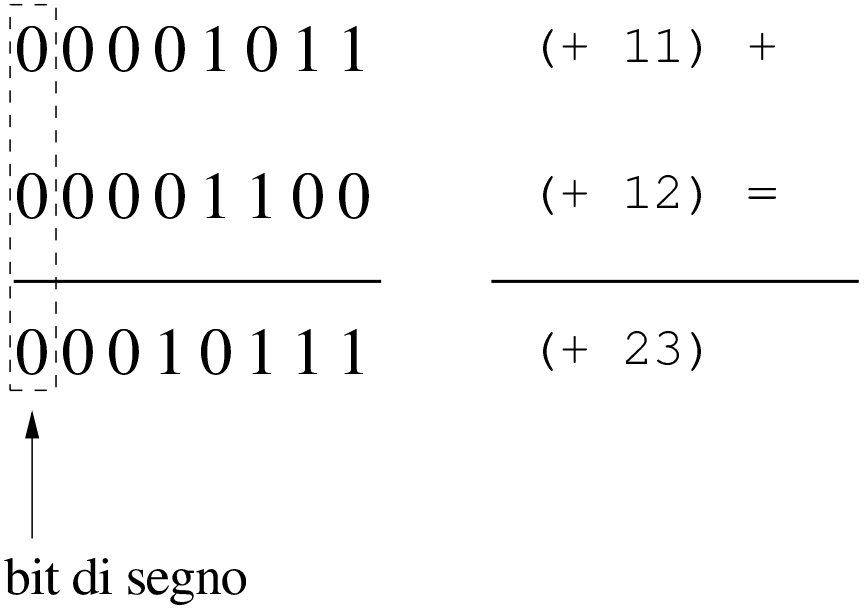


Figura 63.34. Somma di due valori positivi, dove il risultato apparentemente negativo indica la presenza di un traboccamento.

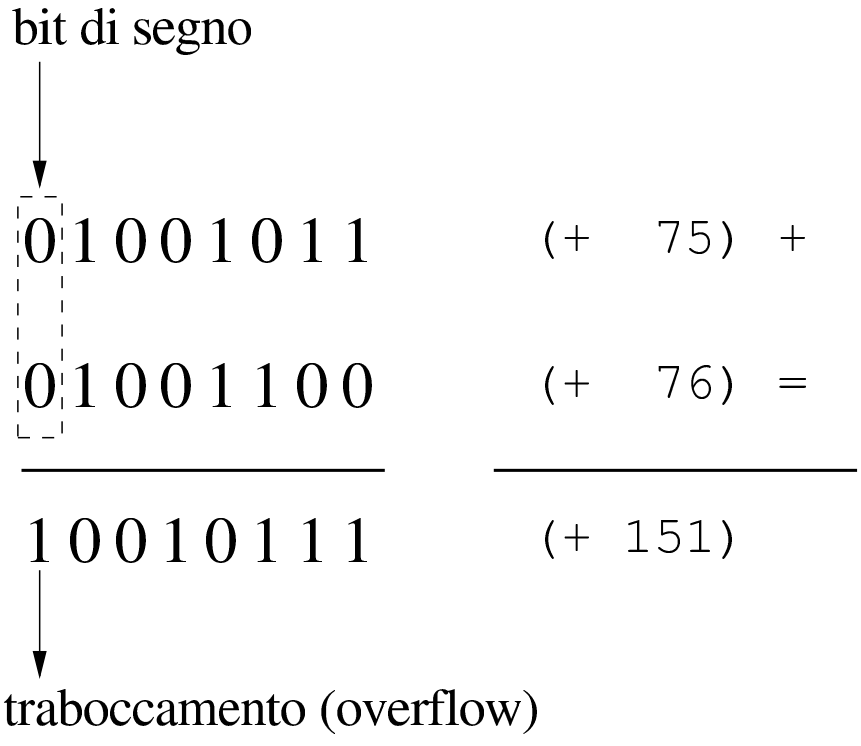


Figura 63.35. Somma di un valore positivo e di un valore negativo: il risultato è sempre valido.

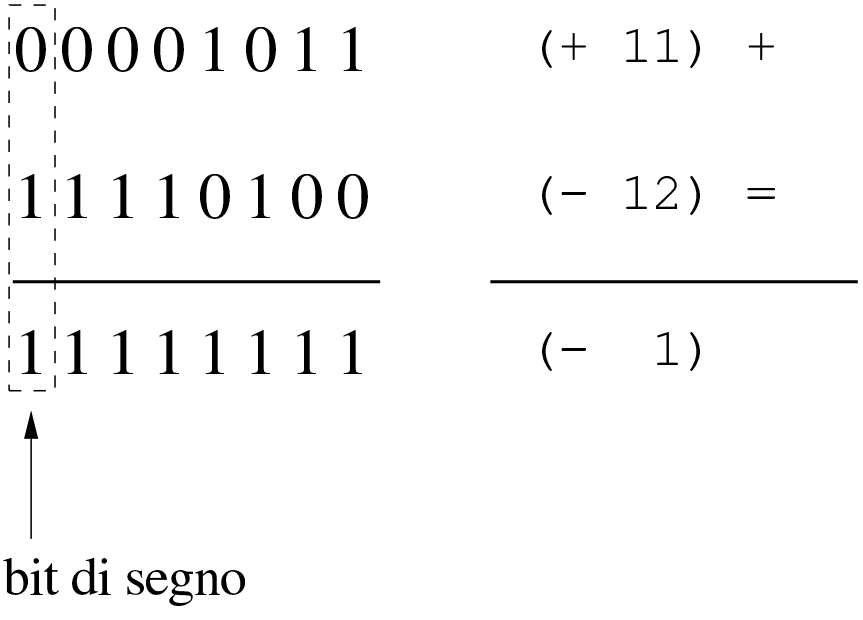


Figura 63.36. Somma di un valore positivo e di un valore negativo: in tal caso il risultato è sempre valido e se si manifesta un riporto, come in questo caso, va ignorato semplicemente.

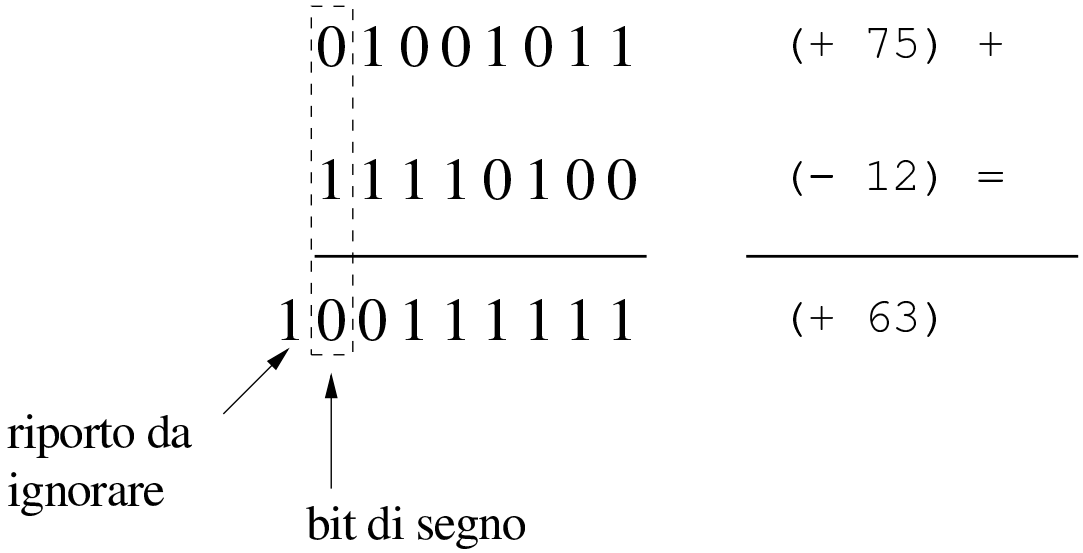


Figura 63.37. Somma di due valori negativi che produce un segno coerente e un riporto da ignorare.

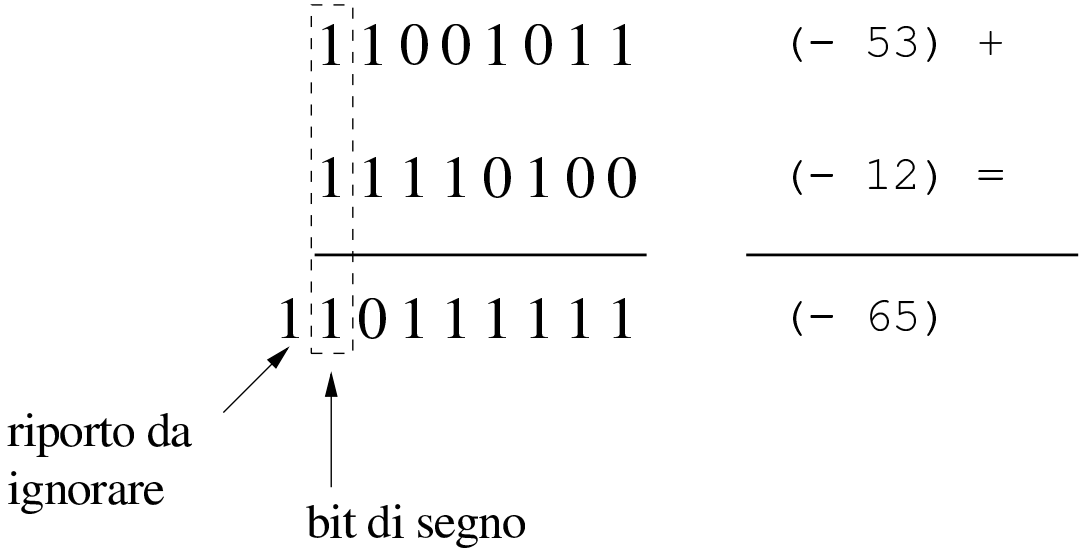
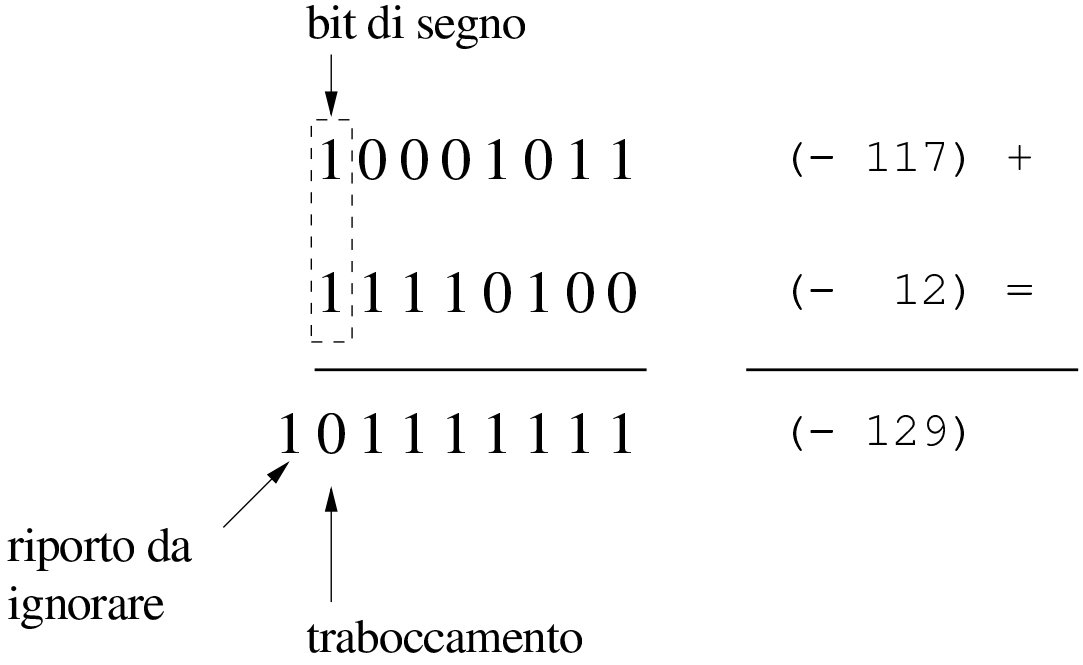


Figura 63.38. Somma di due valori negativi che genera un traboccamento, evidenziato da un risultato con un segno incoerente.



Dagli esempi mostrati si comprende facilmente che la somma di due valori con segno va fatta ignorando il riporto, perché quello che conta è che il segno risultante sia coerente: se si sommano due valori positivi, perché il risultato sia valido deve essere positivo; se si somma un valore positivo con uno negativo il risultato è sempre valido; se si sommano due valori negativi, perché il risultato sia valido deve rimanere negativo.

63.4.3 Somme e sottrazioni con i valori interi senza segno

La somma di due numeri interi senza segno avviene normalmente, senza dare un valore particolare al bit più significativo, pertanto, se si genera un riporto, il risultato non è valido (salva la possibilità di considerarlo assieme al riporto). Se invece si vuole eseguire una sottrazione, il valore da sottrarre va «invertito», con il complemento a due, ma sempre evitando di dare un significato particolare al bit più

significativo. Il valore «normale» e quello «invertito» vanno sommati come al solito, ma **se il risultato non genera un riporto**, allora **è sbagliato**, in quanto il sottraendo è più grande del minuendo.

Per comprendere come funziona la sottrazione, si consideri di volere eseguire un'operazione molto semplice: 1−1. Il minuendo (il primo valore) sia espresso come 00000001_2; il sottraendo (il secondo valore) che sarebbe uguale, va trasformato attraverso il complemento a due, diventando così pari a 11111111_2. A questo punto si sommano algebricamente i due valori e si ottiene 00000000_2 con riporto di uno. Il riporto di uno dà la garanzia che il risultato è corretto. Volendo provare a sottrarre un valore più grande, si vede che il riporto non viene ottenuto: 1−2. In questo caso il minuendo si esprime come nell'esempio precedente, mentre il sottraendo è 00000010_2 che si trasforma nel complemento a due 11111110_2. Se si sommano i due valori si ottiene semplicemente 11111111_2, senza riporto, ma questo valore che va inteso senza segno è evidentemente errato.

Figura 63.39. Sottrazione tra due numeri interi senza segno, dove il sottraendo ha un valore assoluto minore di quello del minuendo: la presenza del riporto conferma la validità del risultato.

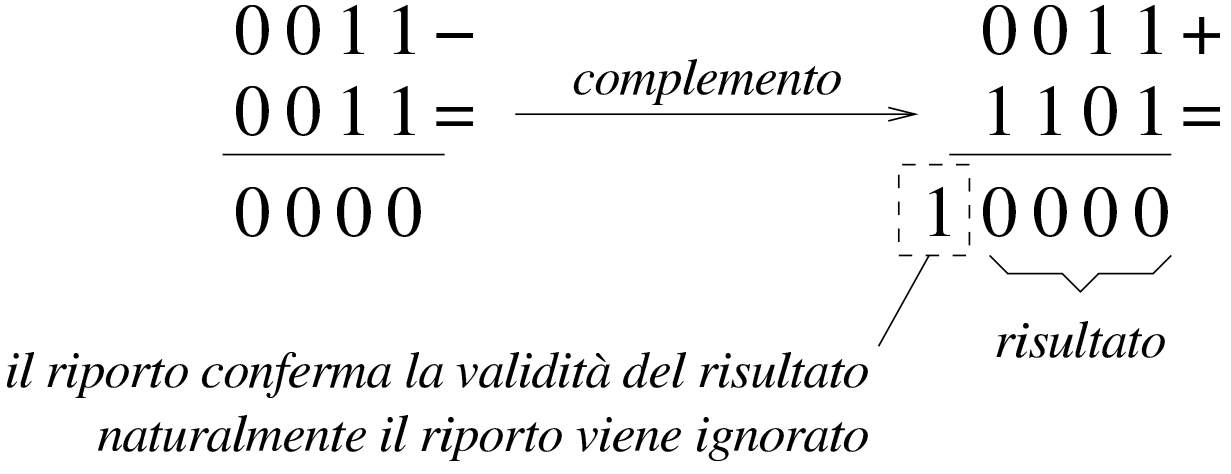
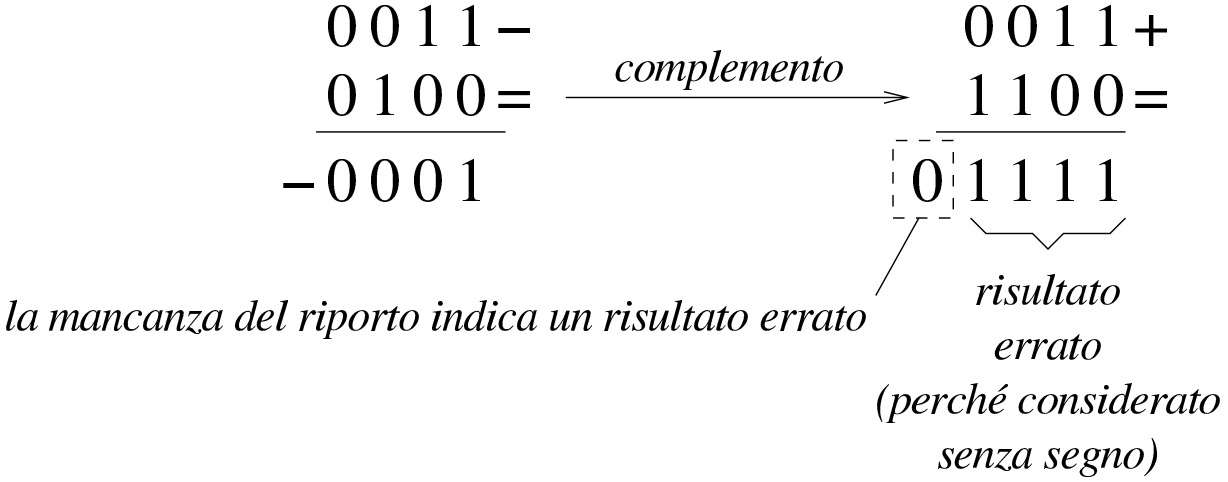
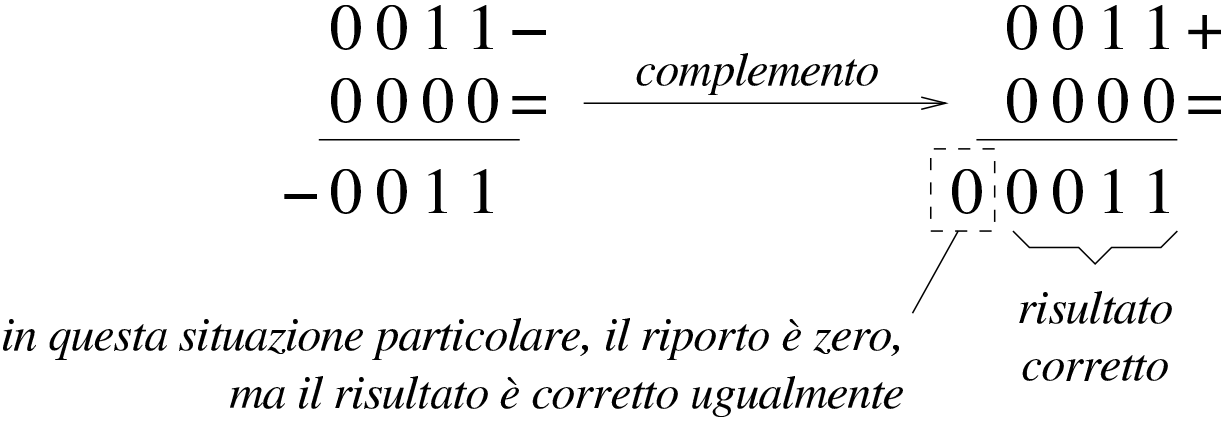


Figura 63.40. Sottrazione tra due numeri interi senza segno, dove il sottraendo ha un valore assoluto maggiore di quello del minuendo: l'assenza di un riporto indica un risultato errato della sottrazione.



Sulla base della spiegazione data, c'è però un problema, dovuto al fatto che il complemento a due di un valore a zero dà sempre zero: se si fa la sottrazione con il complemento, il risultato è comunque corretto, ma non si ottiene un riporto.

Figura 63.41. Sottrazione con sottraendo a zero: non si ottiene riporto, ma il risultato è corretto ugualmente.



Per correggere questo problema, il complemento a due del numero da sottrarre, va eseguito in due fasi: prima si calcola il complemento a uno, poi si somma il minuendo al sottraendo complementato,

aggiungendo una unità ulteriore. Le figure successive ripetono gli esempi già mostrati, attuando questo procedimento differente.

Figura 63.42. Il complemento a due viene calcolato in due fasi: prima si calcola il complemento a uno, poi si sommano il minuendo e il sottraendo invertito, più una unità.

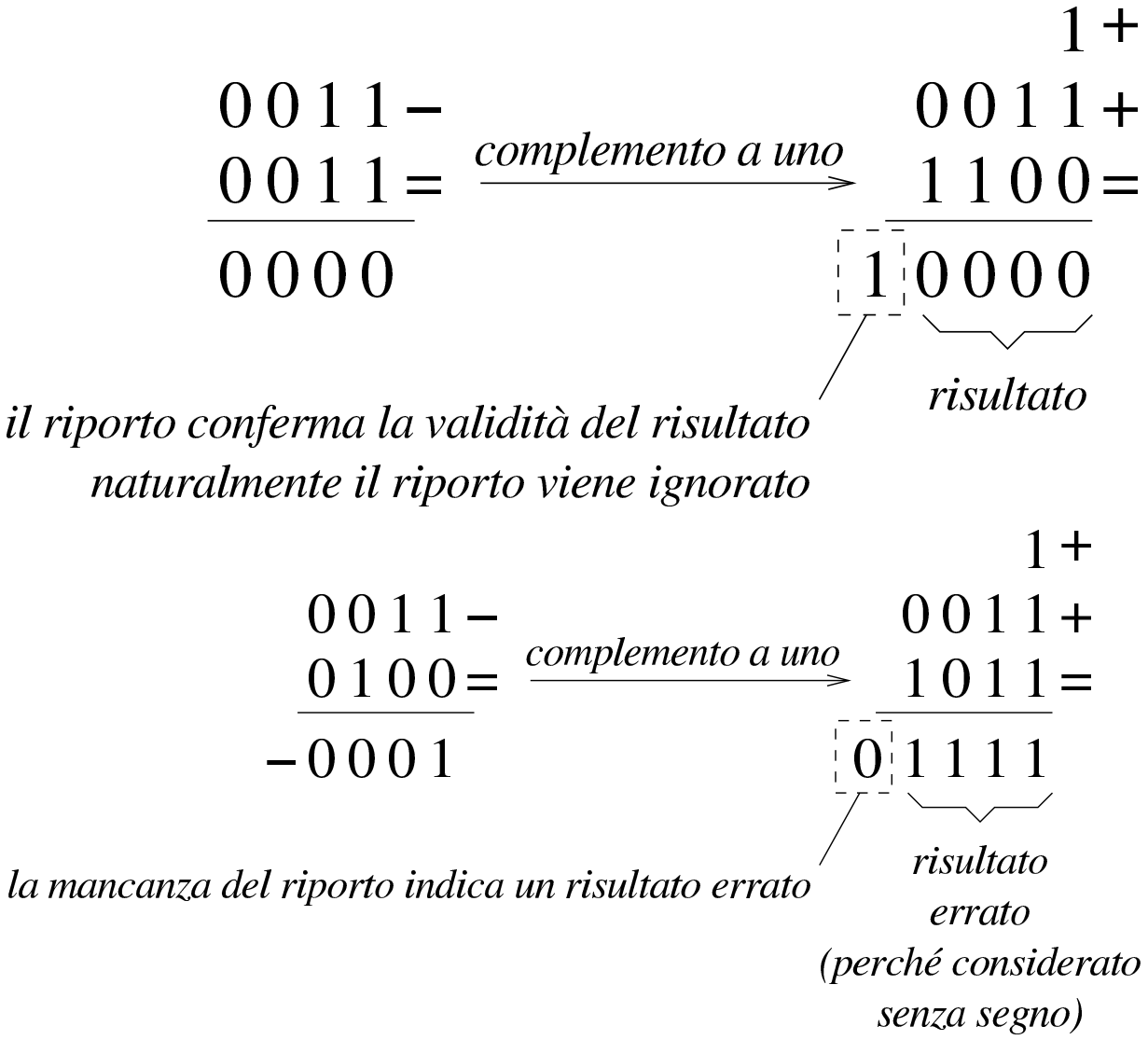
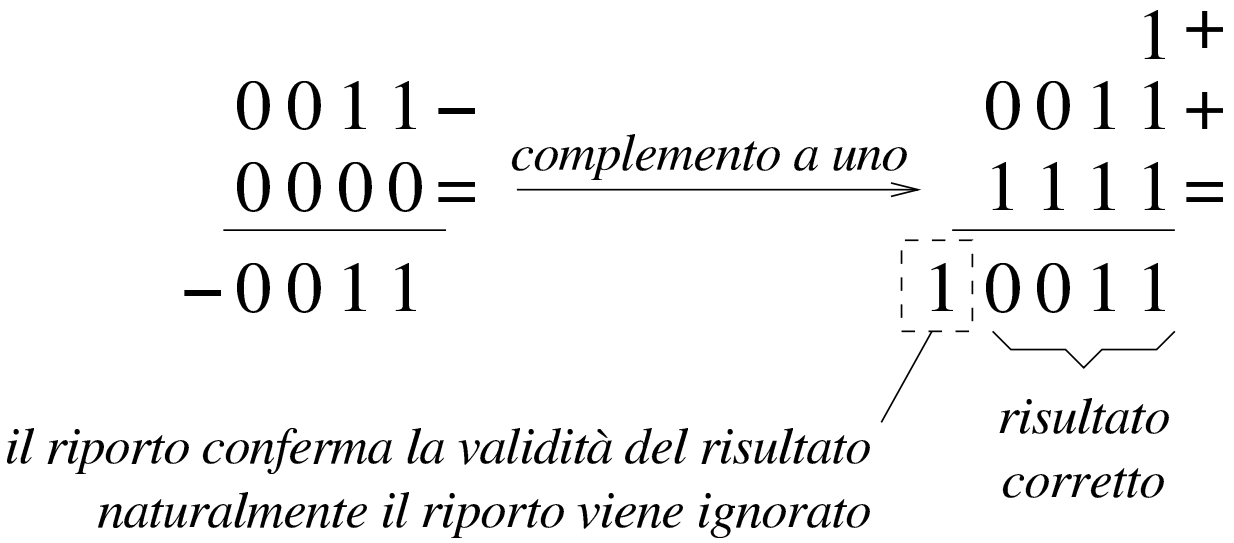


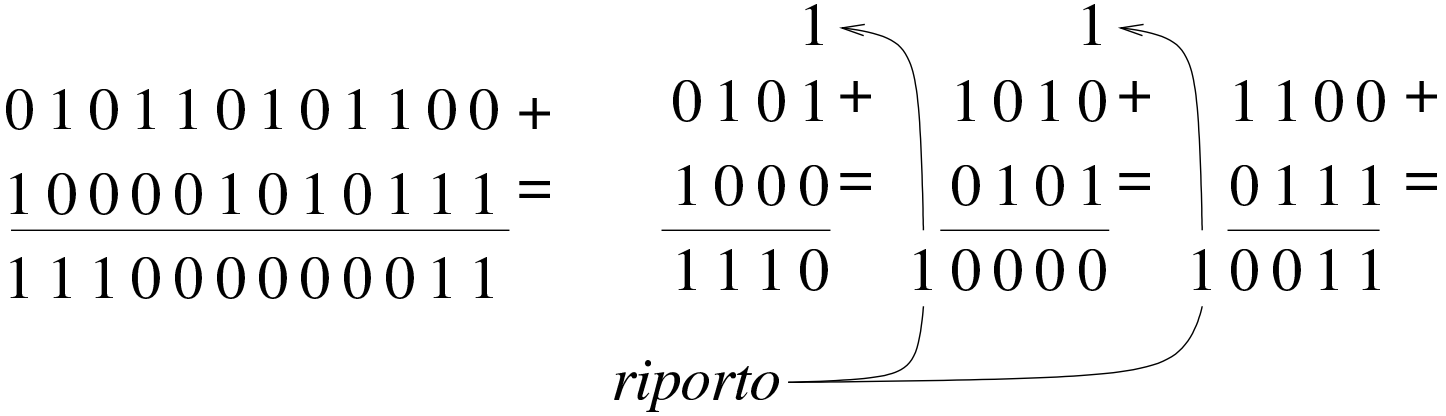
Figura 63.44. Sottrazione con sottraendo a zero: calcolando il complemento a due attraverso il complemento a uno, si ottiene un riporto coerente.



63.4.4 Somme e sottrazioni in fasi successive

Quando si possono eseguire somme e sottrazioni solo con una quantità limitata di cifre, mentre si vuole eseguire un calcolo con numeri più grandi della capacità consentita, si possono suddividere le operazioni in diverse fasi. La somma tra due numeri interi è molto semplice, perché ci si limita a tenere conto del riporto ottenuto nelle fasi precedenti. Per esempio, dovendo sommare $0101\ 1010\ 1100_2$ a $1000\ 0101\ 0111_2$ e potendo operare solo a gruppi di quattro bit per volta: si parte dal primo gruppo di bit meno significativo, 1100_2 e 0111_2, si sommano i due valori e si ottiene 0011_2 con riporto di uno; si prosegue sommando 1010_2 con 0101_2 aggiungendo il riporto e ottenendo 0000_2 con riporto di uno; si conclude sommando 0101_2 e 1000_2, aggiungendo il riporto della somma precedente e si ottiene così 1110_2. Quindi, il risultato è $1110\ 0000\ 0011_2$.

Figura 63.45. Somma per fasi successive, tenendo conto del riporto.



Nella sottrazione tra numeri senza segno, il sottraendo va trasformato secondo il complemento a due, quindi si esegue la somma e si considera che ci deve essere un riporto, altrimenti significa che il sottraendo è maggiore del minuendo. Quando si deve eseguire la sottrazione a gruppi di cifre più piccoli di quelli che richiede il valore per essere rappresentato, si può procedere in modo simile a quello che si usa con la somma, con la differenza che «l'assenza del riporto» indica la richiesta di prendere a prestito una cifra.

Per comprendere il procedimento è meglio partire da un esempio. In questo caso si utilizzano i valori già visti, ma invece di sommarli si vuole eseguire la sottrazione. Per la precisione, si intende prendere $1000\ 0101\ 0111_2$ come minuendo e $0101\ 1010\ 1100_2$ come sottraendo. Anche in questo caso si suppone di poter eseguire le operazioni solo a gruppi di quattro bit. Si esegue il complemento a due dei tre gruppetti di quattro bit del sottraendo, in modo indipendente, ottenendo: 1011_2, 0110_2, 0100_2. A questo punto si eseguono le somme, a partire dal gruppo meno significativo. La prima somma, $0111_2 + 0100_2$, dà 1011_2, senza riporto, pertanto occorre prendere a prestito una cifra dal gruppo successivo: ciò significa che va eseguita

la somma del gruppo successivo, sottraendo una unità dal risultato: $0101_2 + 0110_2 - 0001_2 = 1010_2$. Anche per il secondo gruppo non si ottiene il riporto della somma, così, anche dal terzo gruppo di bit occorre prendere a prestito una cifra: $1000_2 + 1011_2 - 0001_2 = 0010_2$. L'ultima volta la somma genera il riporto (da ignorare) che conferma la correttezza del risultato complessivo, ovvero che la sottrazione è avvenuta con successo.

Va però ricordato il problema legato allo zero, il cui complemento a due dà sempre zero. Se si cambiano i valori dell'esempio, lasciando come minuendo quello precedente, $1000\ 0101\ 0111_2$, ma modificando il sottraendo in modo da avere le ultime quattro cifre a zero, $0101\ 1010\ 0000_2$, il procedimento descritto non funziona più. Infatti, il complemento a due di 0000_2 rimane 0000_2 e se si somma questo a 0111_2 si ottiene lo stesso valore, ma senza riporti. In questo caso, nonostante l'assenza del riporto, il gruppo dei quattro bit successivi, del sottraendo, va trasformato con il complemento a due, senza togliere l'unità che sarebbe prevista secondo l'esempio precedente. In pratica, per poter eseguire la sottrazione per fasi successive, occorre definire un concetto diverso: il prestito (*borrow*) che non deve scattare quando si sottrae un valore pari a zero.

Se il complemento a due viene ottenuto passando per il complemento a uno, con l'aggiunta di una cifra, si può spiegare in modo più semplice il procedimento della sottrazione per fasi successive: invece di calcolare il complemento a due dei vari tronconi, si calcola semplicemente il complemento a uno e al gruppo meno significativo si aggiunge una unità per ottenere lì l'equivalente di un complemento a due. Successivamente, il riporto delle somme eseguite va aggiunto al gruppo adiacente più significativo, come si farebbe con la somma:

se la sottrazione del gruppo precedente non ha bisogno del prestito di una cifra, si ottiene l'aggiunta una unità al gruppo successivo.

Figura 63.46. Sottrazione per fasi successive, tenendo conto del prestito delle cifre.

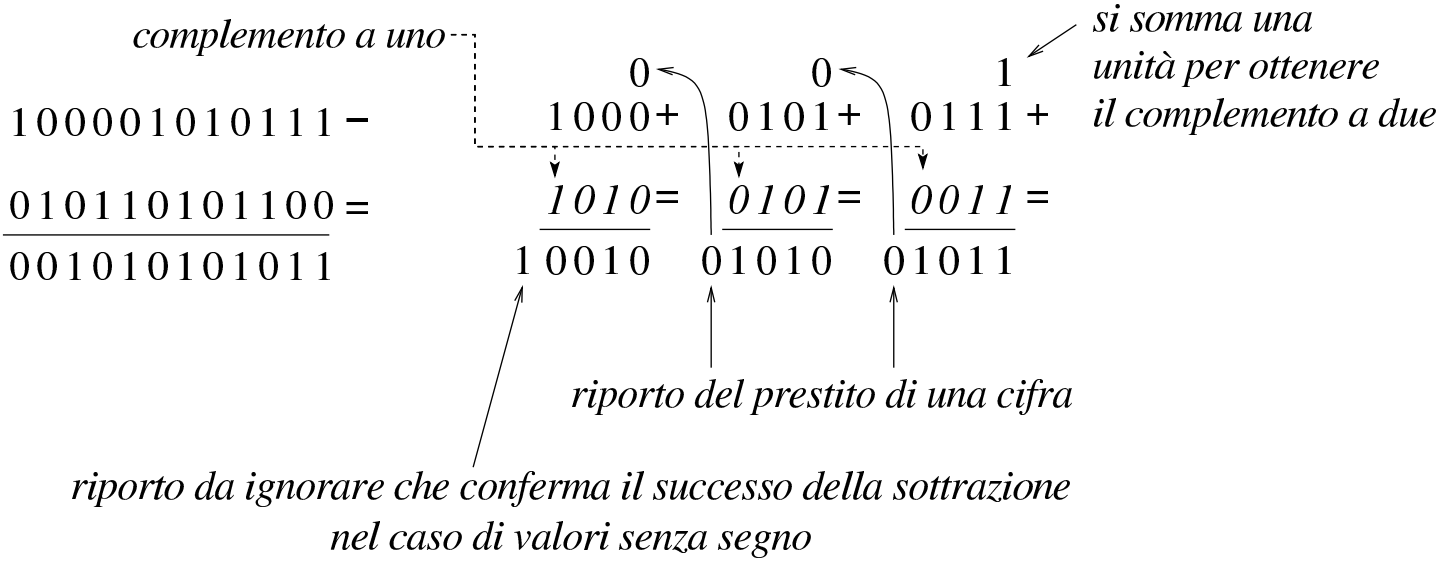
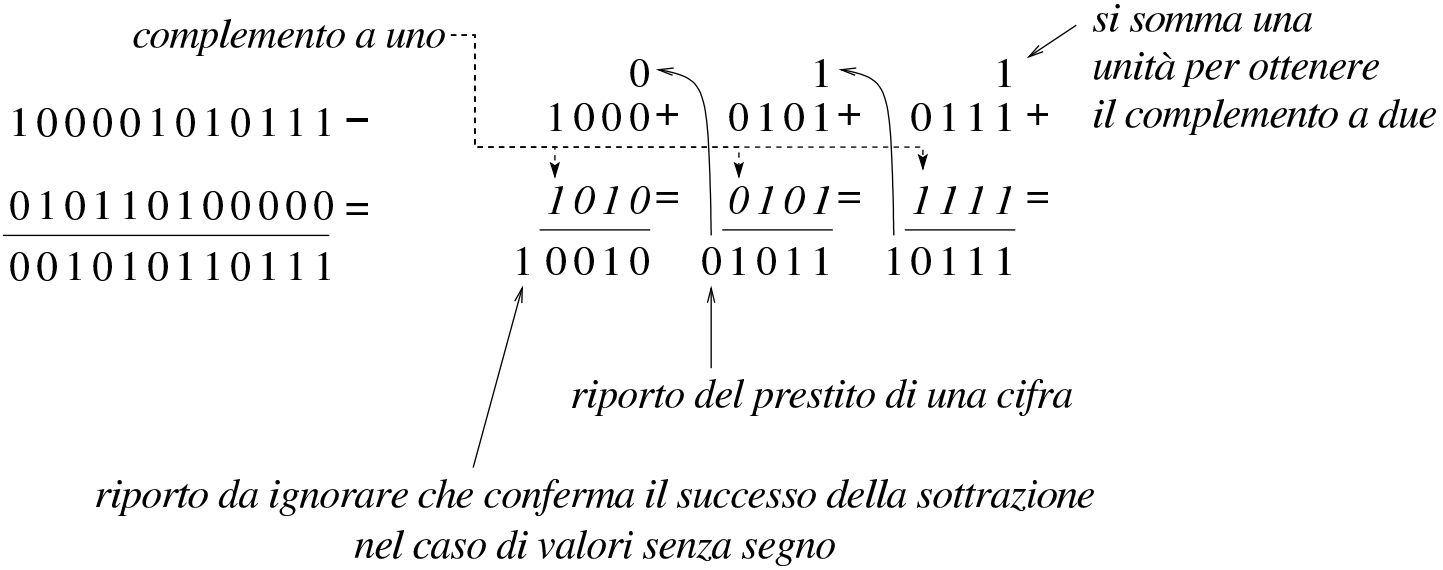


Figura 63.47. Verifica del procedimento anche in presenza di un sottraendo a zero.



La sottrazione per fasi successive funziona anche con valori che, complessivamente, hanno un segno. L'unica differenza sta nel modo di valutare il risultato complessivo: l'ultimo gruppo di cifre a es-

sere considerato (quello più significativo) è quello che contiene il segno ed è il segno del risultato che deve essere coerente, per stabilire se ciò che si è ottenuto è valido. Pertanto, nel caso di valori con segno, il riporto finale si ignora, esattamente come si fa quando la sottrazione avviene in una fase sola, mentre l'esistenza o meno del traboccamento deriva dal confronto della cifra più significativa: se la sottrazione, dopo la trasformazione in somma con il complemento, implica la somma valori con lo stesso segno, il risultato deve ancora avere quel segno, altrimenti c'è il traboccamento.

Se si volessero considerare gli ultimi due esempi come la sottrazione di valori con segno, il minuendo si intenderebbe un valore negativo, mentre il sottraendo sarebbe un valore positivo. Attraverso il complemento si passa alla somma di due valori negativi, ma dal momento che si ottiene un risultato con segno positivo, ciò manifesta un traboccamento, ovvero un risultato errato, perché non contenibile nello spazio disponibile.

63.4.5 Indicatori

«

Quando si esegue un calcolo con un microprocessore, oltre al risultato puro e semplice è necessario annotare altre informazioni sull'esito dell'operazione stessa. Come già descritto, una somma può dare luogo a un traboccamento o a un riporto, così come una sottrazione può richiedere un prestito di una cifra. Queste e altre informazioni, che non possono essere incorporate nel risultato di un calcolo, finiscono all'interno di indicatori (*flag*), ovvero di bit singoli, ognuno con un proprio significato preciso. Le figure successive dimostrano il funzionamento degli indicatori più comuni.

Figura 63.48. Somma di interi: se i numeri sono da intendersi senza segno, il risultato non è completo in quanto si genera un riporto; se i numeri sono da intendersi con segno, in tal caso sono negativi, ma la loro somma produce un traboccamento.

Figura 63.49. Somma di interi: se i numeri sono da intendersi senza segno, il risultato non è completo in quanto si genera un riporto; se i numeri sono da intendersi con segno, in tal caso hanno segni diversi tra di loro e questo impedisce che si crei un traboccamento, inoltre il risultato è zero e si attiva l'indicatore relativo.

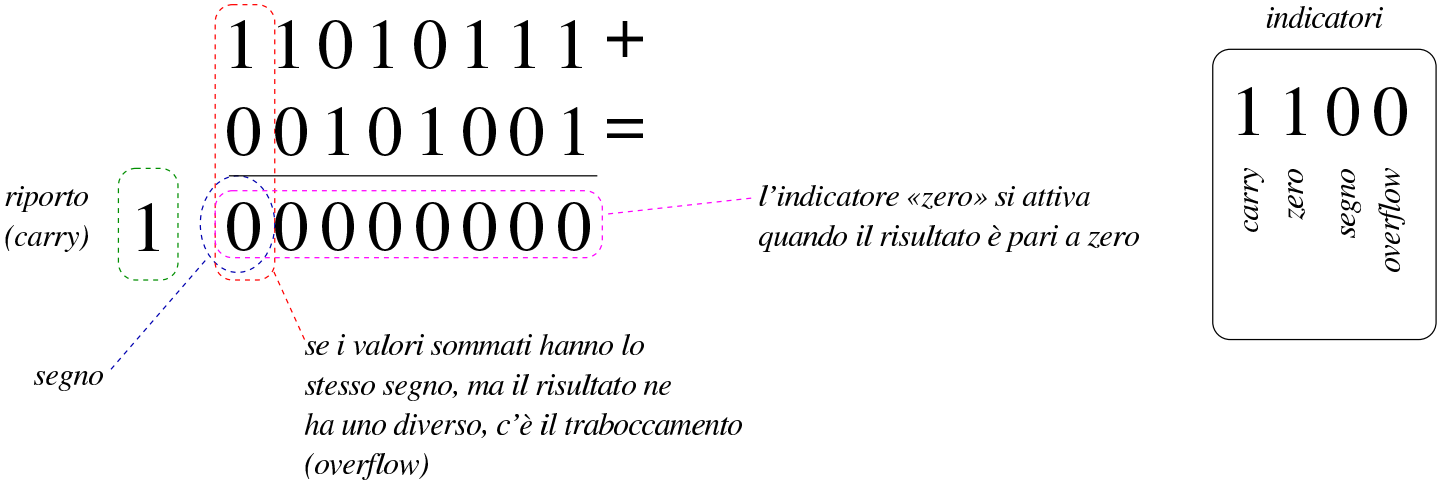
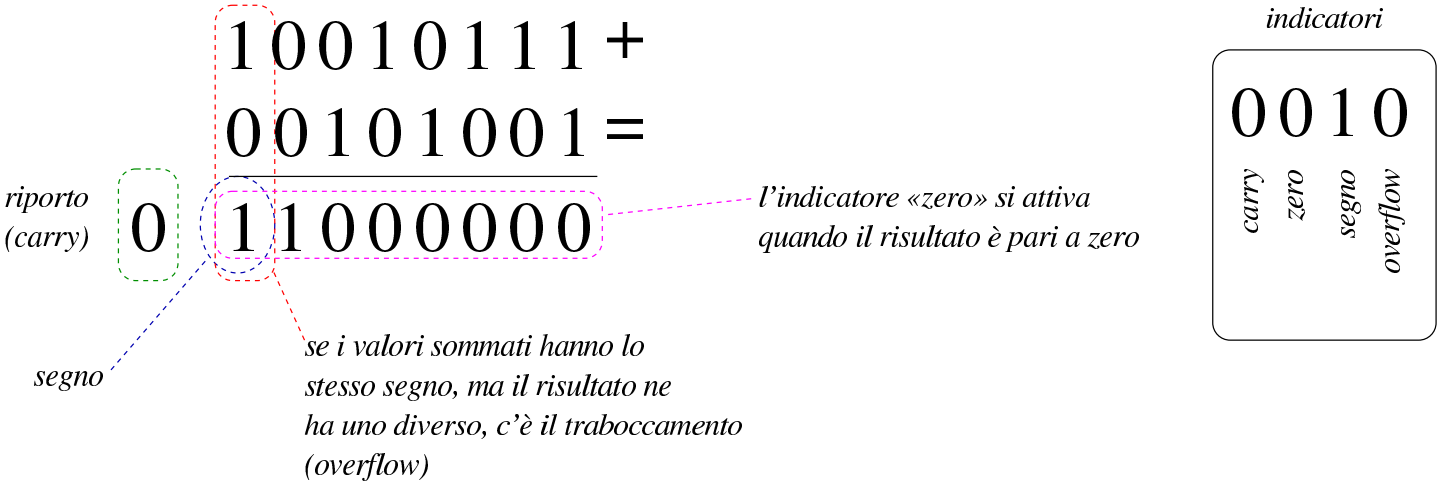


Figura 63.50. Somma di interi: se i numeri sono da intendersi senza segno, il risultato è completo in quanto non si genera un riporto; se i numeri sono da intendersi con segno, in tal caso hanno segni diversi tra di loro e questo impedisce che si crei un traboccamento, inoltre il risultato è negativo e questo attiva l'indicatore di segno.



Come già descritto in altre sezioni, le sottrazioni vanno eseguite calcolando prima il complemento a uno del sottraendo e poi aggiungendo una unità ulteriore. In questo modo, ciò che nella somma rappresenterebbe un riporto, qui va invertito per segnalare la richiesta di un prestito (*borrow*).

Figura 63.51. Sottrazione di interi: se i numeri sono da intendersi senza segno, il risultato è completo in quanto non si genera la richiesta di prestito di una cifra; se i numeri sono da intendersi con segno, si tratta di valori negativi, ma la somma genera un cambiamento di segno, pertanto si attiva l'indicatore di traboccamento.

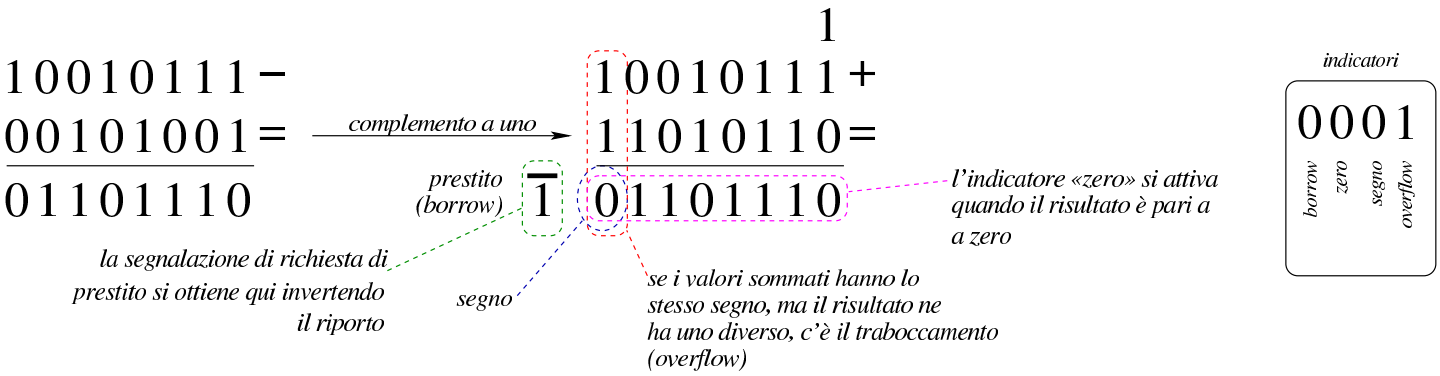
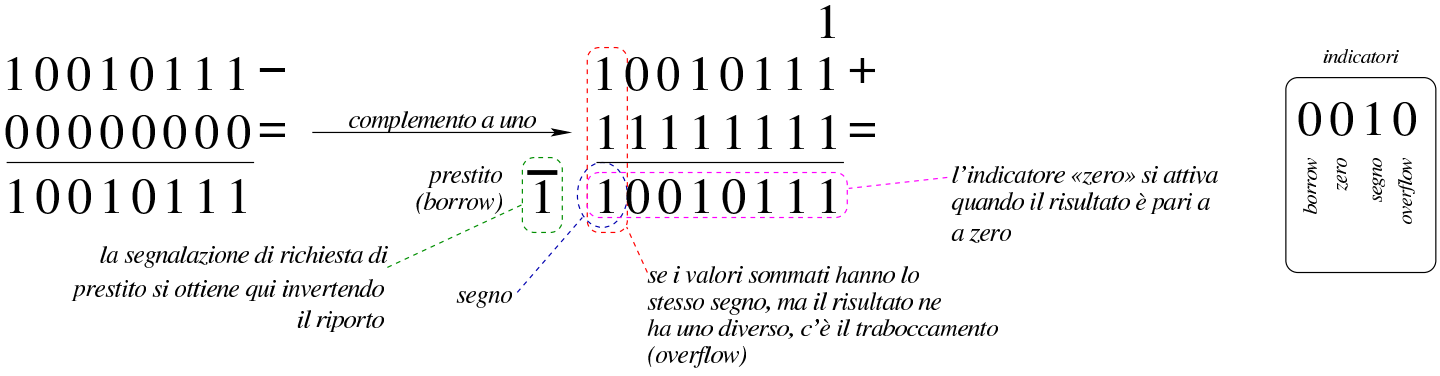


Figura 63.52. Sottrazione di interi: qui viene sottratto zero da un valore. Utilizzando il meccanismo del complemento a uno, aggiungendo una unità alla somma, si evita che scatti la richiesta di prestito, come è logico che sia. In questo caso, dato che il risultato è negativo, si attiva l'indicatore di segno.



63.5 Scorrimenti, rotazioni, operazioni logiche

Le operazioni più semplici che si possono compiere con un microprocessore sono quelle che riguardano la logica booleana e lo scorri-

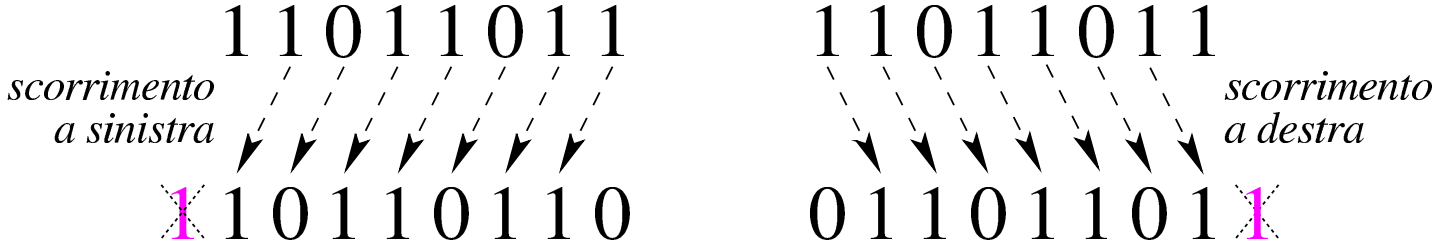
mento dei bit. Proprio per la loro semplicità è importante conoscere alcune applicazioni interessanti di questi procedimenti elaborativi.

63.5.1 Scorrimento logico

«

Lo scorrimento «logico» consiste nel fare scalare le cifre di un numero binario, verso sinistra (verso la parte più significativa) o verso destra (verso la parte meno significativa). Nell'eseguire questo scorrimento, da un lato si perde una cifra, mentre dall'altro si acquista uno zero.

Figura 63.53. Scorrimento logico a sinistra, perdendo le cifre più significative e scorrimento logico a destra, perdendo le cifre meno significative.



Lo scorrimento di una posizione verso sinistra corrisponde alla moltiplicazione del valore per due, mentre lo scorrimento a destra corrisponde a una divisione intera per due; scorrimenti di n posizioni rappresentano moltiplicazioni e divisioni per 2^n. Le cifre che si perdono nello scorrimento a sinistra si possono considerare come il riporto della moltiplicazione, mentre le cifre che si perdono nello scorrimento a destra sono il resto della divisione.

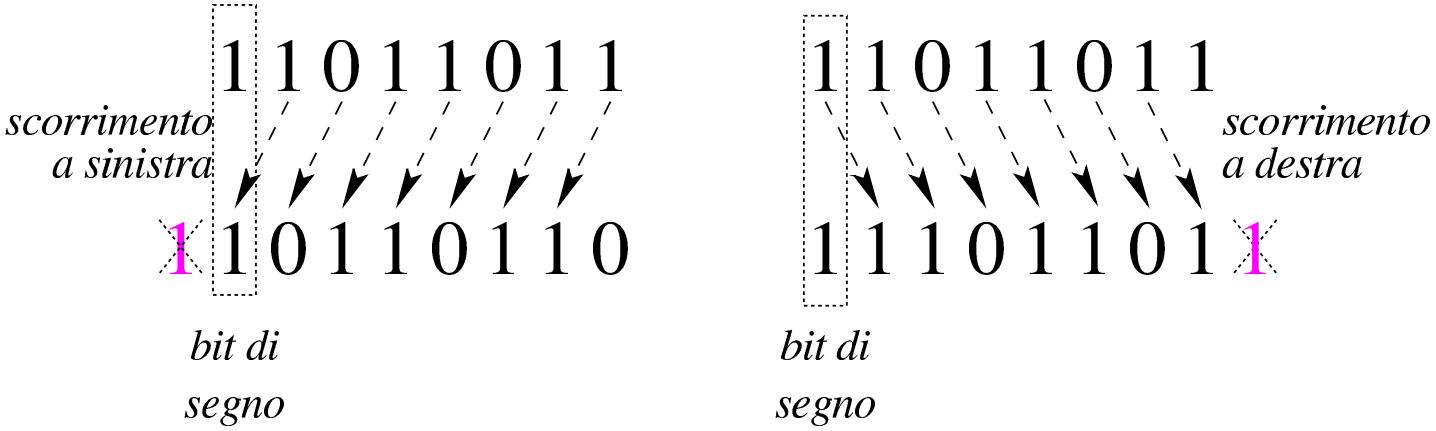
63.5.2 Scorrimento aritmetico

«

Il tipo di scorrimento descritto nella sezione precedente, se utilizzato per eseguire moltiplicazioni e divisioni, va bene solo per valori senza segno. Se si intende fare lo scorrimento di un valore con se-

gno, occorre distinguere due casi: lo scorrimento a sinistra è valido se il risultato non cambia di segno; lo scorrimento a destra implica il mantenimento del bit che rappresenta il segno e l'aggiunta di cifre uguali a quella che rappresenta il segno stesso.

Figura 63.54. Scorrimento aritmetico a sinistra e a destra, di un valore negativo.



63.5.3 Scorrimento e indicatori

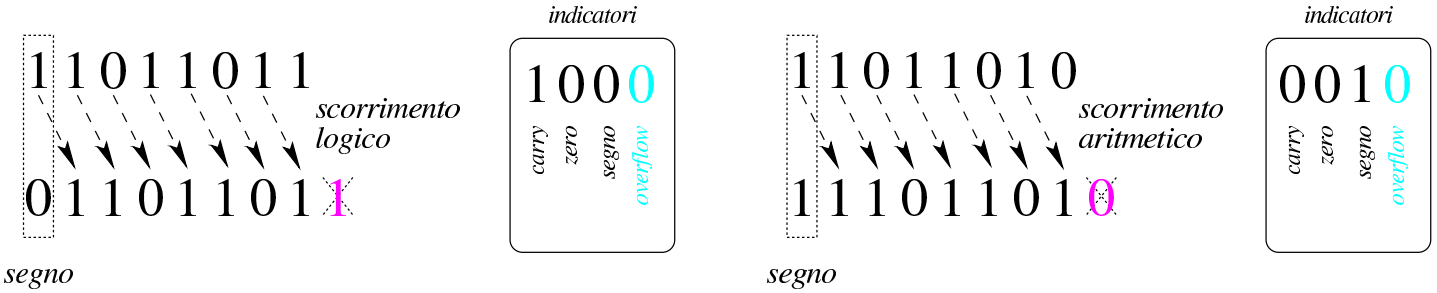
Tenendo conto che gli scorrimenti si eseguono sempre per una sola posizione alla volta e che rappresentano una moltiplicazione o una divisione per due, tornano utili gli stessi indicatori descritti a proposito di somme e sottrazioni. Come già accennato, il riporto viene usato per segnalare la cifra che viene perduta (sia per lo scorrimento verso sinistra, sia per quello verso destra), mentre l'indicatore di traboccamento (*overflow*) serve a segnalare che il risultato cambia di segno.

Lo scorrimento aritmetico verso sinistra avviene nello stesso modo di quello «logico». Nel caso il valore che viene fatto scorrere sia considerato privo di segno, il risultato della moltiplicazione per due è valido se non si presenta un riporto; se invece il valore ha un segno, il risultato è «corretto» se il segno non è cambiato.

Figura 63.55. Scorrimento a sinistra. Nel lato sinistro si vede che il risultato non è valido se si tratta di un valore senza segno, in quanto si presenta un riporto, mentre sarebbe valido se fosse un valore con segno, perché questo non cambia. A destra, invece, si vede un valore che se è da intendere senza segno, dà un risultato corretto, mentre se ha il segno, il risultato non è più valido perché il segno si inverte (*overflow*).

Lo scorrimento verso destra avviene in modo diverso se il valore va inteso con segno o senza segno, perché se si presta attenzione al segno si usa lo scorrimento aritmetico che inserisce a sinistra cifre uguali al segno precedente. Pertanto, nello scorrimento a destra si considera solo il resto, che finisce in pratica nello stesso indicatore del riporto.

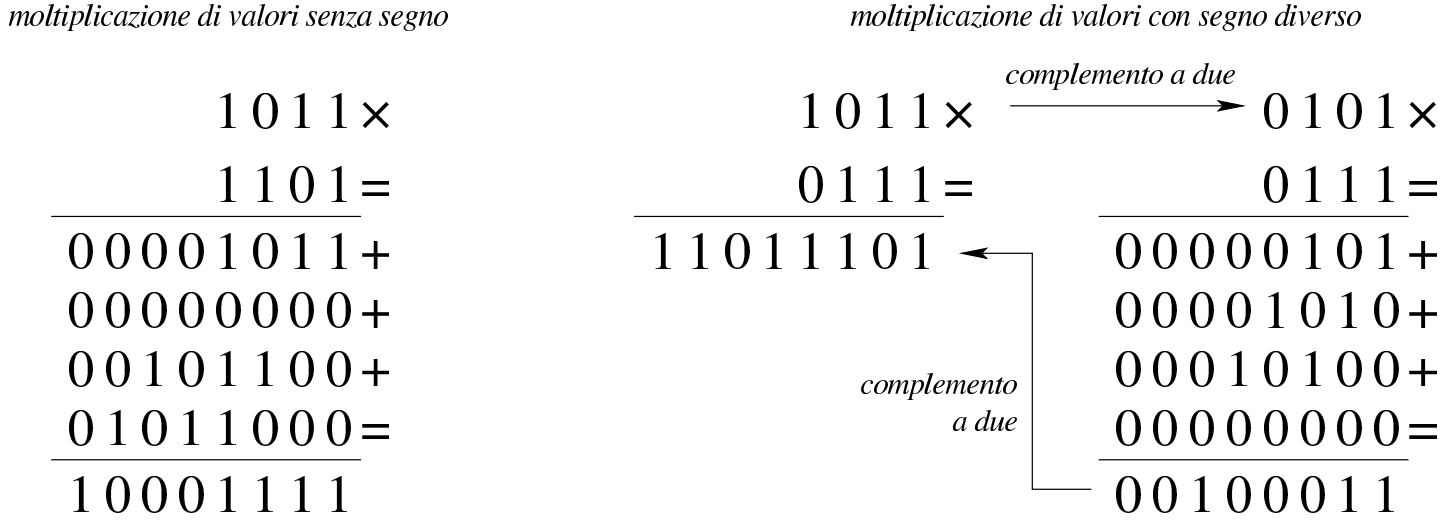
Figura 63.56. Nel lato sinistro si vede uno scorrimento «logico» che produce un resto, mentre in quello destro si vede uno scorrimento aritmetico che, in questo caso, non produce alcun resto.



63.5.4 Moltiplicazione

La moltiplicazione si ottiene attraverso diverse fasi di scorrimento e somma di un valore, dove però il risultato richiede un numero doppio di cifre rispetto a quelle usate per il moltiplicando e il moltiplicatore. Il procedimento di moltiplicazione deve avvenire sempre con valori senza segno. Se i valori si intendono con segno, quando sono negativi occorre farne prima il complemento a due, in modo da portarli a valori positivi, quindi occorre decidere se il risultato va preso così come viene o se va invertito a sua volta con il complemento a due: se i valori moltiplicati hanno segno diverso tra loro, il risultato deve essere trasformato con il complemento a due per renderlo negativo, altrimenti il risultato è sempre positivo.

Figura 63.57. Moltiplicazione.

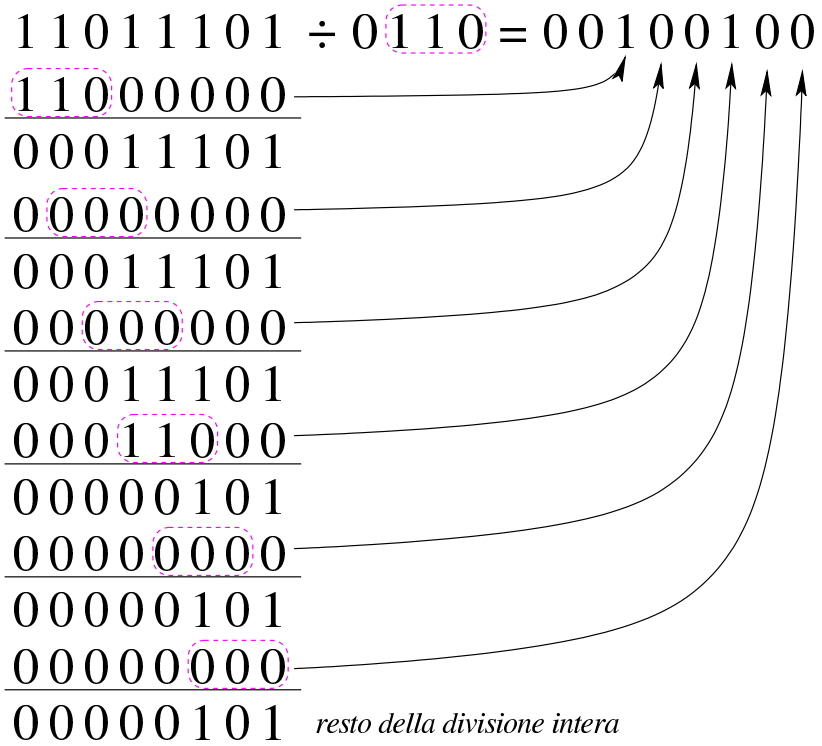


63.5.5 Divisione

La divisione si ottiene attraverso diverse fasi di scorrimento di un valore, che di volta in volta viene sottratto al dividendo, ma solo se la sottrazione è possibile effettivamente. Il procedimento di divisione deve avvenire sempre con valori senza segno. Se i valori si intendono

con segno, quando sono negativi occorre farne prima il complemento a due, in modo da portarli a valori positivi, quindi occorre decidere se il risultato va preso così come viene o se va invertito a sua volta con il complemento a due: se dividendo e divisore hanno segni diversi tra loro, il risultato deve essere trasformato con il complemento a due per renderlo negativo, altrimenti il risultato è sempre positivo.

Figura 63.58. Divisione: i valori sono intesi senza segno.



221 : 6 = 36

con il resto di 5

63.5.6 Rotazione

«

La rotazione è uno scorrimento dove le cifre che si perdono da una parte rientrano dall'altra. Esistono due tipi di rotazione; uno «normale» e l'altro che include nella rotazione il bit del riporto. Dal momento che la rotazione non si presta per i calcoli matematici, di solito non viene considerato il segno.

Figura 63.59. Rotazione normale.

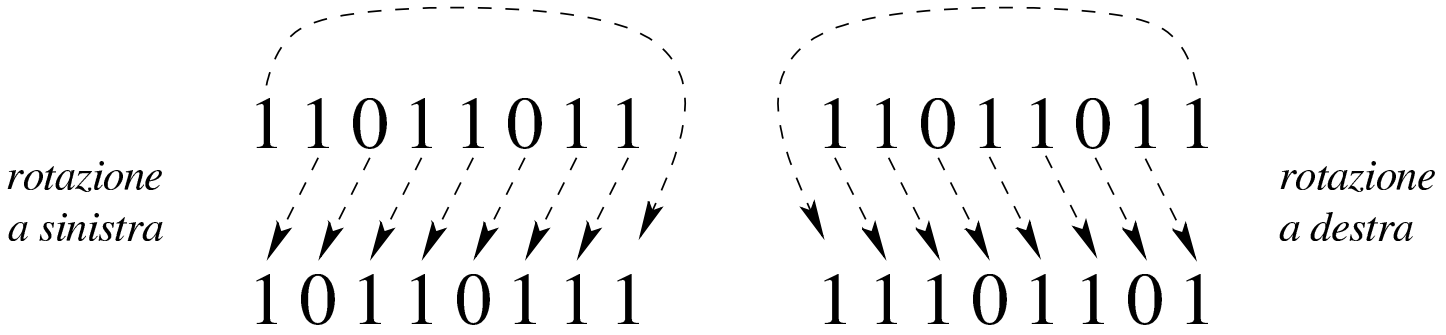
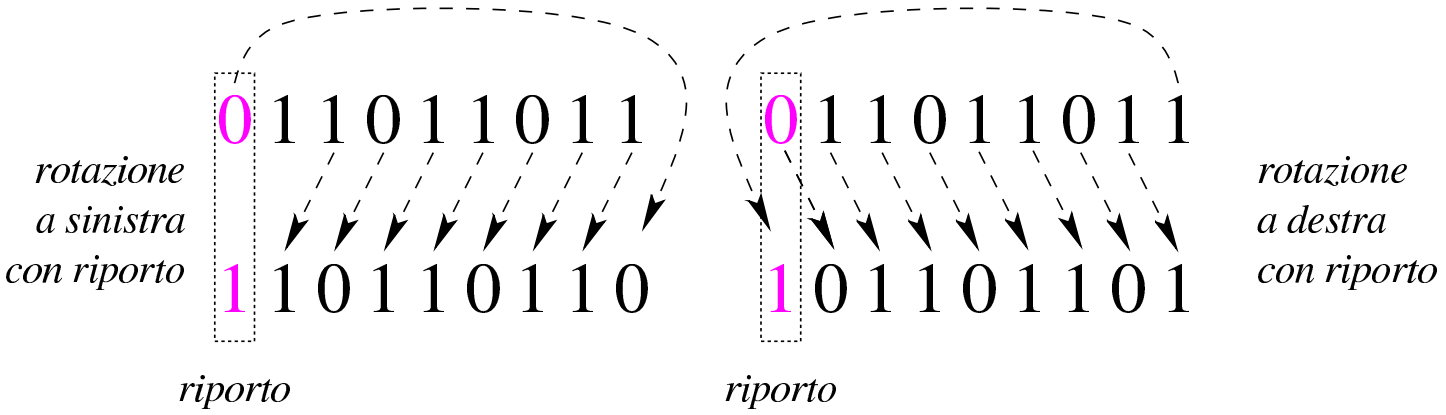


Figura 63.60. Rotazione con riporto.



63.5.7 Rotazione e indicatori

Le rotazioni non sono riconducibili a operazioni matematiche, ma si usa ugualmente l'indicatore del riporto per conservare la cifra persa; inoltre, l'indicatore di traboccamento può servire per annotare un'ipotesi di cambiamento di segno.

Figura 63.61. Rotazione normale. La cifra che fuoriesce da un lato e rientra dall'altro, rimane annotata nell'indicatore di riporto; nel caso dell'esempio di rotazione a destra, l'indicatore di traboccamento segnala che la cifra più significativa è diversa rispetto alla fase precedente.

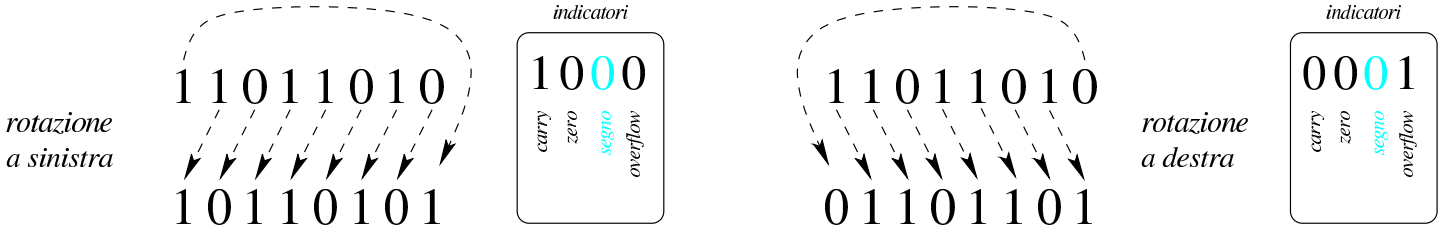
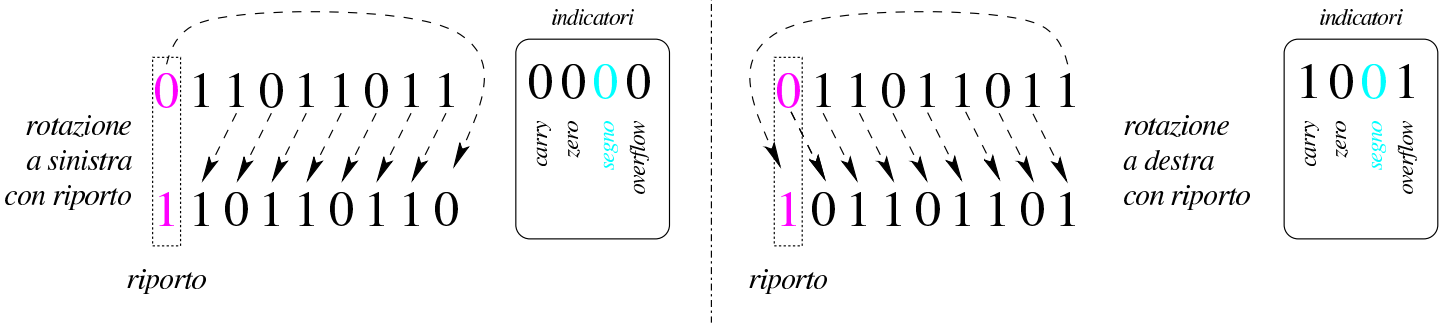


Figura 63.62. Rotazione con riporto. La cifra che fuoriesce da un lato entra nel riporto, mentre dall'altro lato entra la cifra conservata nel riporto precedente; nel caso dell'esempio di rotazione a destra, l'indicatore di traboccamento segnala che la cifra più significativa è diversa rispetto alla fase precedente.



63.5.8 Operatori logici

Gli operatori logici si possono applicare anche a valori composti da più cifre binarie.

Figura 63.63. AND e OR.

1 1 0 1 1 0 1 1	*a*	1 1 0 1 1 0 1 1	*a*
0 1 1 0 1 1 0 1	*b*	0 1 1 0 1 1 0 1	*b*
0 1 0 0 1 0 0 1	*a AND b*	1 1 1 1 1 1 1 1	*a OR b*

Figura 63.64. XOR e NOT.

1 1 0 1 1 0 1 1	*a*	1 1 0 1 1 0 1 1	*a*
0 1 1 0 1 1 0 1	*b*		
1 0 1 1 0 1 1 0	*a XOR b*	0 0 1 0 0 1 0 0	*NOT a*

È importante osservare che l'operatore NOT esegue in pratica il complemento a uno di un valore.

Capita spesso di trovare in un sorgente scritto in un linguaggio assemblatore un'istruzione che assegna a un registro il risultato dell'operatore XOR su se stesso. Ciò si fa, evidentemente, per azzerarne il contenuto, quando, probabilmente, l'assegnamento esplicito di un valore a un registro richiede una frazione di tempo maggiore per la sua esecuzione.

Figura 63.65. XOR per azzerare i valori.

1 1 0 1 1 0 1 1	*a*
1 1 0 1 1 0 1 1	*a*
0 0 0 0 0 0 0 0	*a XOR a*

63.5.9 Intervenire su bit singoli

Quando si lavora con valori binari composti da una quantità prestabilita di cifre, per intervenire singolarmente o comunque solo parzialmente sulle stesse occorre predisporre delle maschere da abbinare poi con un operatore logico appropriato. Segue la descrizione di alcuni esempi.

- Si vuole attivare il quarto bit (contando a partire dalla cifra meno significativa) nella variabile ***x***.

  x := ***x*** OR 8_{10}

  $$\begin{array}{r} 11011011\ x \\ \underline{00001000\ 8_{10}\ (mashera)} \\ 11011011\ x\ OR\ 8_{10} \end{array} \qquad \begin{array}{r} 10110110\ x \\ \underline{00001000\ 8_{10}\ (mashera)} \\ 10111110\ x\ OR\ 8_{10} \end{array}$$

- Si vuole disattivare il quarto bit (contando a partire dalla cifra meno significativa) nella variabile ***x***.

  x := ***x*** AND (NOT 8_{10})

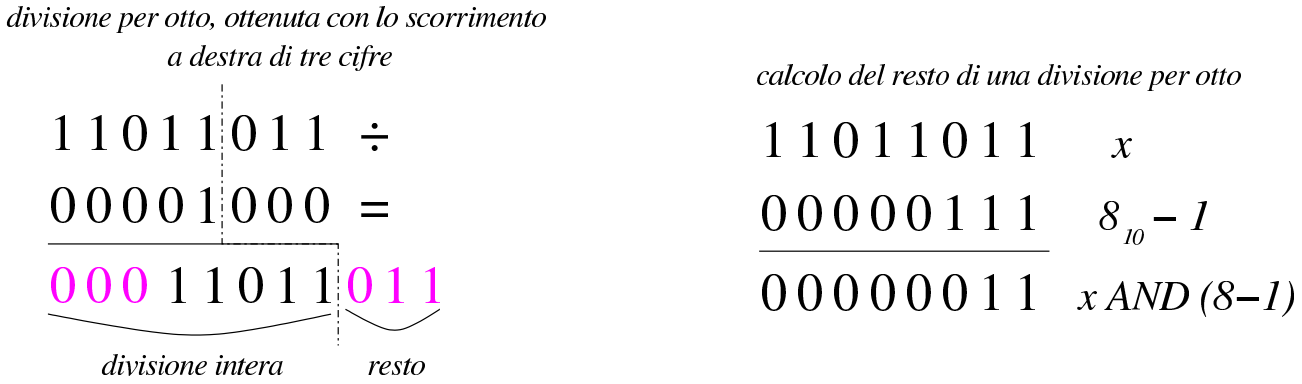
  $$\begin{array}{r} 00001000\ 8_{10} \\ \underline{} \\ 11110111\ NOT\ 8_{10} \end{array} \qquad \begin{array}{r} 11011011\ x \\ 11110111\ NOT\ 8_{10}\ (mashera) \\ \underline{} \\ 11010011\ x\ AND\ (NOT\ 8_{10}) \end{array}$$

- Si vuole invertire il quarto bit (contando a partire dalla cifra meno significativa) nella variabile ***x***.

  x := ***x*** XOR 8_{10}

  $$\begin{array}{r} 11011011\ x \\ \underline{00001000\ 8_{10}\ (mashera)} \\ 11010011\ x\ XOR\ 8_{10} \end{array} \qquad \begin{array}{r} 10110110\ x \\ \underline{00001000\ 8_{10}\ (mashera)} \\ 10111110\ x\ XOR\ 8_{10} \end{array}$$

- Si vuole dividere un valore per otto (che è una potenza di due, ovvero 2^3), calcolando il quoziente intero e il resto. Per farlo occorre far scorrere il valore verso destra, di tre posizioni. Tenendo conto che le cifre che vengono espulse sono quelle che rappresentano il resto, questo lo si può ottenere con una maschera pari a sette ($2^3 - 1$), abbinata con l'operatore AND.



63.5.10 Somme e sottrazioni abbinate agli operatori logici

Esiste una proprietà interessante della sottrazione, quando viene abbinata all'operatore logico AND. Come si vede nella figura successiva, quando si riduce un valore di una unità, quella che prima era la cifra a uno meno significativa passa a zero, mentre le cifre precedenti passano a uno. Così facendo, se si abbinano i due valori (quello originale e quello ridotto di una unità) con l'operatore AND, si ottiene un nuovo valore in cui, semplicemente, la cifra meno significativa a uno passa a zero:

$$x = 0\ 1\ 1\ 0\ 1\ 1\ 0\ 0$$
$$x-1 = 0\ 1\ 1\ 0\ 1\ 0\ 1\ 1$$
$$x\ AND\ (x-1) = 0\ 1\ 1\ 0\ 1\ 0\ 0\ 0$$

Per converso, se si incrementa di una unità un valore e poi si abbinata l'operatore logico OR (tra il valore originale e quello incrementato

di una unità), si ottiene di portare a uno la cifra meno significativa che prima era a zero:

$$x = 0\ 1\ 1\ 0\ 1\ 0\ 1\ 1$$
$$x{+}1 = 0\ 1\ 1\ 0\ 1\ 1\ 0\ 0$$
$$x\ OR\ (x{+}1) = 0\ 1\ 1\ 0\ 1\ 1\ 1\ 1$$

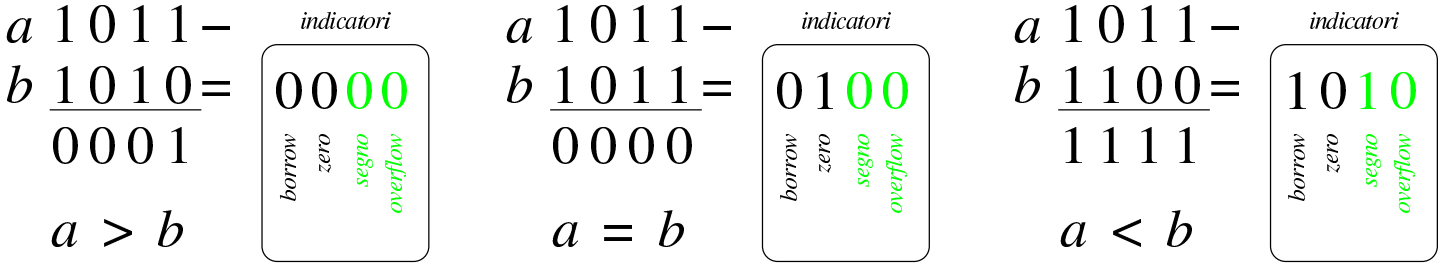
63.6 Confronti attraverso la sottrazione

Il confronto tra due valori avviene provando a sottrarne uno dall'altro. In un microprocessore, l'esito di una sottrazione, come mostrato dagli indicatori comuni, consente di confrontare i valori originali.

63.6.1 Confronto di valori senza segno

Se si esegue una sottrazione e si attiva l'indicatore del risultato zero, senza la presenza di una richiesta del prestito di una cifra, i valori sono uguali; se il valore ottenuto è diverso da zero e non c'è alcuna richiesta di prestito, vuol dire che il sottraendo ha un valore inferiore al minuendo; negli altri casi, il sottraendo ha un valore maggiore del minuendo.

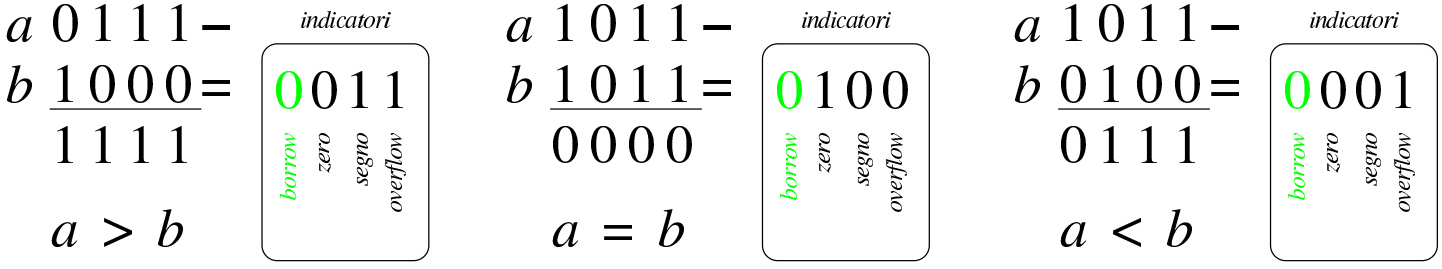
Figura 63.72. Confronto di valori senza segno.



63.6.2 Confronto di valori con segno

Il confronto tra valori con segno avviene in modo meno intuitivo di quello che invece lo ignora. Qui non si considera l'indicatore del prestito di una cifra, mentre vanno considerati al suo posto gli indicatori di segno e di traboccamento, che possono essere uguali o meno tra di loro. Pertanto: se il risultato della sottrazione dà zero, i valori confrontati sono uguali; se il risultato della sottrazione è diverso da zero, se gli indicatori di segno e di traboccamento sono uguali, vuol dire che il sottraendo è inferiore del minuendo; diversamente il sottraendo è superiore al minuendo.

Figura 63.73. Confronto di valori con segno.



Dal momento che il meccanismo del confronto di valori con segno può essere difficile da comprendere con pochi esempi, si aggiunge un prospetto con i confronti fra tutti i valori che si possono rappresentare con due soli bit, sia senza segno, sia con segno. Nel prospetto viene mostrata la sottrazione e l'addizione dopo l'inversione del sottraendo, inoltre sono annotati tutti i riporti e i prestiti parziali.

Figura 63.74. Verifica di tutti i casi di confronto per valori a due bit.

	minuendo		sottraendo		risultato	confronto	sottrazione con prestito di cifre	somma del sottraendo dopo la trasformazione con il complemento a uno e l'aggiunta di una unità	borrow	zero	segno	overflow
con segno	1	-	-2	=	3	1 > -2						
senza segno	1	-	2	=	-1	1 < 2			1	0	1	1
con segno	1	-	-1	=	2	1 > -1						
senza segno	1	-	3	=	-2	1 < 3			1	0	1	1
con segno	1	-	0	=	1	1 > 0						
senza segno	1	-	0	=	1	1 > 0			0	0	0	0
con segno	1	-	1	=	0	1 = 1						
senza segno	1	-	1	=	0	1 = 1			0	1	0	0
con segno	0	-	-2	=	2	0 > -2						
senza segno	0	-	2	=	-2	0 < 2			1	0	1	1
con segno	0	-	-1	=	1	0 > -1						
senza segno	0	-	3	=	-3	0 < 3			1	0	0	0
con segno	0	-	0	=	0	0 = 0						
senza segno	0	-	0	=	0	0 = 0			0	1	0	0
con segno	0	-	1	=	-1	0 < 1						
senza segno	0	-	1	=	-1	0 < 1			1	0	1	0
con segno	-1	-	-2	=	1	-1 > -2						
senza segno	3	-	2	=	1	3 > 2			0	0	0	0
con segno	-1	-	-1	=	0	-1 = -1						
senza segno	3	-	3	=	0	3 = 3			0	1	0	0
con segno	-1	-	0	=	-1	-1 < 0						
senza segno	3	-	0	=	3	3 > 0			0	0	1	0
con segno	-1	-	1	=	-2	-1 < 1						
senza segno	3	-	1	=	2	3 > 1			0	0	1	0
con segno	-2	-	-2	=	0	-2 = -2						
senza segno	2	-	2	=	0	2 = 2			0	1	0	0
con segno	-2	-	-1	=	-1	-2 < -1						
senza segno	2	-	3	=	-1	2 < 3			1	0	1	0
con segno	-2	-	0	=	-2	-2 < 0						
senza segno	2	-	0	=	2	2 > 0			0	0	1	0
con segno	-2	-	1	=	-3	-2 < 1						
senza segno	2	-	1	=	1	2 > 1			0	0	0	1

63.7 Riferimenti

- Jonathan Bartlett, *Programming from the ground up*, 2003, *http://savannah.nongnu.org/projects/pgubook/*
- Paul A. Carter, *PC Assembly Language*, 2006, *http://www.drpaulcarter.com/pcasm/*
- Wikipedia, *Addressing mode*, *http://en.wikipedia.org/wiki/Addressing_mode*
- Mario Italiani, Giuseppe Serazzi, *Elementi di informatica*, ETAS libri, 1973, ISBN 8845303632
- Sandro Petrizzelli, *Appunti di elettronica digitale*, *http://users.libero.it/sandry/Digitale_01.pdf*
- Tony R. Kuphaldt, *Lessons In Electric Circuits*, *http://www.faqs.org/docs/electric/*
- Wikipedia, *Sistema numerico binario* *http://it.wikipedia.org/wiki/Sistema_numerico_binario*
- Wikipedia, *IEEE 754*, *http://it.wikipedia.org/wiki/IEEE_754*

[1] Questa sezione riprende e in parte ripete, per maggiore chiarezza, un concetto già presentato nella sezione 63.1.8.

[2] Nel contesto riferito alla definizione di un numero in virgola mobile, si possono usare indifferentemente i termini ***mantissa*** o ***significante***, così come sono indifferenti i termini ***caratteristica*** o ***esponente***.

[3] Si osservi che lo standard IEEE 754 utilizza una «mantissa normalizzata» che indica la frazione di valore tra uno e due: «1,***mantissa***.

[4] Considerato che si tratta di un numero da esprimere in base due, il valore che viene moltiplicato per la potenza di due è un numero che va da uno a due.

[5] Per completezza, questa sezione riprende un concetto già descritto nella sezione 63.1.6.

64.1 Terminologia impropria 183
64.2 Registri principali fino ai 32 bit 183
64.3 Sintesi delle istruzioni principali 186
64.4 Primo approccio al linguaggio assemblatore per x86 . 213
64.4.1 Il primo programma 213
64.4.2 Utilizzo di GDB 217
64.4.3 Modalità «TUI» 226
64.4.4 Utilizzo di DDD 231
64.4.5 Alcune istruzioni comuni 233
64.4.6 Dimensione dei dati nelle istruzioni 236
64.4.7 Direttive per il compilatore 238
64.4.8 Sezioni del sorgente 241
64.4.9 Usare GNU AS con la notazione Intel 242
64.5 Esempi con le «quattro operazioni» 244
64.5.1 Somma 248
64.5.2 Sottrazione 259
64.5.3 Moltiplicazione senza segno 266
64.5.4 Moltiplicazione con segno 269
64.5.5 Divisione 272
64.6 Esempi con gli «spostamenti» 275
64.6.1 Scorrimento logico 277

64.6.2 Scorrimento aritmetico 281
64.6.3 Rotazione 284
64.6.4 Rotazione con riporto 287
64.7 Esempi con i confronti 290
64.8 Le istruzioni di salto 298
64.8.1 Portata del salto 298
64.8.2 Salto incondizionato 299
64.8.3 Salto condizionato dallo stato di un indicatore 300
64.8.4 Salto condizionato da un confronto 301
64.8.5 Cicli 304
64.9 Esempi di programmi con strutture di controllo 305
64.9.1 Somma attraverso l'incremento unitario . . . 306
64.9.2 Moltiplicazione attraverso la somma 309
64.9.3 Divisione attraverso la sottrazione 312
64.9.4 Elevamento a potenza 314
64.9.5 Moltiplicazione attraverso lo scorrimento e la somma 316
64.9.6 Conteggio dei bit a uno 319
64.10 Funzioni 322
64.10.1 Esempio banale di chiamata 322
64.10.2 Salvataggio dei registri prima della chiamata . . 326
64.10.3 Passaggio di parametri attraverso la pila 327

64.10.4 Utilizzo del registro «EBP» 334
64.10.5 Allocazione dello spazio per le variabili locali e preservazione dei registri 341
64.10.6 Convenzioni di chiamata 348
64.10.7 Nota sugli array «locali» 349
64.11 Esempi di funzioni ricorsive 350
64.11.1 Elevamento a potenza 350
64.11.2 Fattoriale 354
64.12 Indirizzamento dei dati 358
64.12.1 Gestione di array 360
64.12.2 Istruzione «LEA» 366
64.13 Rappresentazione dei dati in memoria attraverso un esempio 368
64.14 Esempi con gli array 375
64.14.1 Ricerca sequenziale 375
64.14.2 Ricerca binaria 380
64.14.3 Bubblesort 386
64.15 Calcoli con gli indirizzi in fase di compilazione 391
64.15.1 Distanza tra due indirizzi 391
64.15.2 Riempimento di spazio inutilizzato 393
64.16 Interazione con il sistema operativo 395
64.16.1 Parametri di chiamata del programma 395
64.16.2 Funzioni del sistema operativo 398

64.16.3 Esempi di lettura e scrittura con i flussi standard . 400

64.17 Riferimenti 405

.ascii 239 .bss 241 .byte 239 .data 241 .equ 238 .int 239 .lcomm 239 .text 241 ADC 189 255 ADD 189 235 248 AH 183 AL 183 AND 193 AX 183 BH 183 BL 183 BP 183 BSWAP 187 BX 183 CALL 196 322 350 CBW 187 CDQ 187 CH 183 CL 183 CLC 200 CMC 200 CMP 200 290 301 306 CWDE 187 CX 183 db 239 dd 239 DEC 189 236 306 DH 183 DI 183 DIV 189 272 DL 183 DX 183 EAX 183 EBP 183 334 EBX 183 ECX 183 EDI 183 EDX 183 EFLAGS 183 EIP 183 ENTER 341 350 equ 238 ESI 183 ESP 183 FLAGS 183 IDIV 189 272 IMUL 189 269 INC 189 236 306 INT 196 213 IP 183 JA 201 301 JAE 201 301 JB 201 301 312 JBE 201 301 JC 201 300 JCXZ 201 JE 201 301 306 JG 201 301 JGE 201 301 JL 201 301 JLE 201 301 JMP 201 299 309 JNA 201 301 JNAE 201 301 JNB 201 JNBE 201 301 JNC 201 300 322 JNE 201 301 JNG 201 301 JNGE 201 301 JNL 201 301 JNLE 301 JNO 201 300 JNP 201 300 JNS 201 300 JNZ 201 300 306 JO 201 300 JP 201 300 JS 201 300 JZ 201 300 314 LEA 187 366 LEAVE 341 350 LOOP 211 304 306 LOOPE 211 304 LOOPNE 211 304 LOOPNZ 211 304 LOOPZ 211 304 MOV 187 213 234 MOVSX 187 MOVZX 187 322 MUL 189 266 NEG 189 261 NOP 187 NOT 193 OR 193 POP 196 326 POPA 196 341 350 POPAD 196 POPF 196 PUSH 196 326 PUSHA 196 341 350 PUSHF 196 RCL 193 287 RCR 193 287 resb 239 resd 239 resw 239 RET 196 322 350 ROL 193 284 ROR 193 284 SAL 193 281 SAR 193 281 SBB 189 263 SETA 205 SETAE 205 SETB 205 SETBE 205 SETC 205 SETE 205 SETG 205 SETGE 205 SETL 205 SETLE 205 SETNA 205 SETNAE 205 SETNB 205 SETNBE 205 SETNC 205 SETNE 205

SETNG 205 SETNGE 205 SETNL 205 SETNLE 205 SETNO 205 SETNS 205 SETNZ 205 SETO 205 SETS 205 SETZ 205 SHL 193 277 316 SHR 193 277 316 SI 183 SP 183 SUB 189 235 259 TEST 200 XCHG 187 XOR 193

64.1 Terminologia impropria

È bene ricordare che nella documentazione standard sui microprocessori x86 si usa una terminologia coerente, ma impropria, riferita alla dimensione delle unità di dati. In particolare, il problema nasce dal fatto che originariamente questi microprocessori avevano parole da 16 bit, così è stato associato il termine *word* a 16 bit ed è rimasta tale l'associazione anche con la trasformazione successiva a 32 bit. Pertanto, generalmente valgono le convenzioni riportate nella tabella successiva.

Tabella 64.1. Terminologia associata alla dimensione dei dati nei microprocessori x86.

Definizione o abbreviazione	Dimensione dei dati	Definizione o abbreviazione	Dimensione dei dati
«b», byte	8 bit	«w», *word*	16 bit
«d», «dw», *dword*, *double word*	32 bit	«q», «qw», *qword*, *quad word*	64 bit

64.2 Registri principali fino ai 32 bit

I registri dei microprocessori x86 sono stati inizialmente da 16 bit; successivamente, quelli principali sono stati estesi a 32 bit. Alcuni

registri hanno una funzione ben precisa; gli altri sono utilizzabili per scopi generali, ma in pratica ognuno ha un compito preferenziale.

Tutti i registri che sono stati estesi da 16 bit; a 32 bit; hanno due nomi: uno riferito alla porzione dei 16 bit meno significativi, l'altro che riguarda il registro nel suo complesso. Inoltre, per quattro registri in particolare, è possibile individuare anche i due byte che compongono la parte meno significativa. Per esempio, il registro ***EAX*** ha una dimensione di 32 bit, di cui è possibile individuare i 16 bit meno significativi con il nome ***AX***, ma in più, il nome ***AL*** individua il byte meno significativo di ***AX*** mentre ***AH*** ne individua quello più significativo.

Il registro ***EIP*** (o ***IP*** nei microprocessori a 16 bit) viene gestito automaticamente e serve a contenere l'indirizzo dell'istruzione successiva da eseguire. In effetti, la gestione manuale di tale registro non sarebbe conveniente, dal momento che la dimensione delle istruzioni in linguaggio macchina varia in base al tipo e agli operandi.

Il registro ***ESP*** (o ***SP*** nei microprocessori a 16 bit) individua l'ultimo elemento della pila dei dati e può essere gestito manualmente, sapendo che questo indice si deve spostare a gruppi di quattro byte (o due byte nella versione a 16 bit) e che la pila cresce diminuendo l'indice.

Il registro ***EBP*** (o ***BP*** nei microprocessori a 16 bit) si affianca all'indice della pila, per tenere conto della posizione raggiunta all'inizio di una funzione.

Il registro ***EFLAGS*** (o ***FLAGS*** nei microprocessori a 16 bit) raccoglie i vari indicatori che descrivono l'esito delle operazioni svolte. In particolare sono importanti gli indicatori che appaiono nella tabella

64.3.

Figura 64.2. Registri principali.

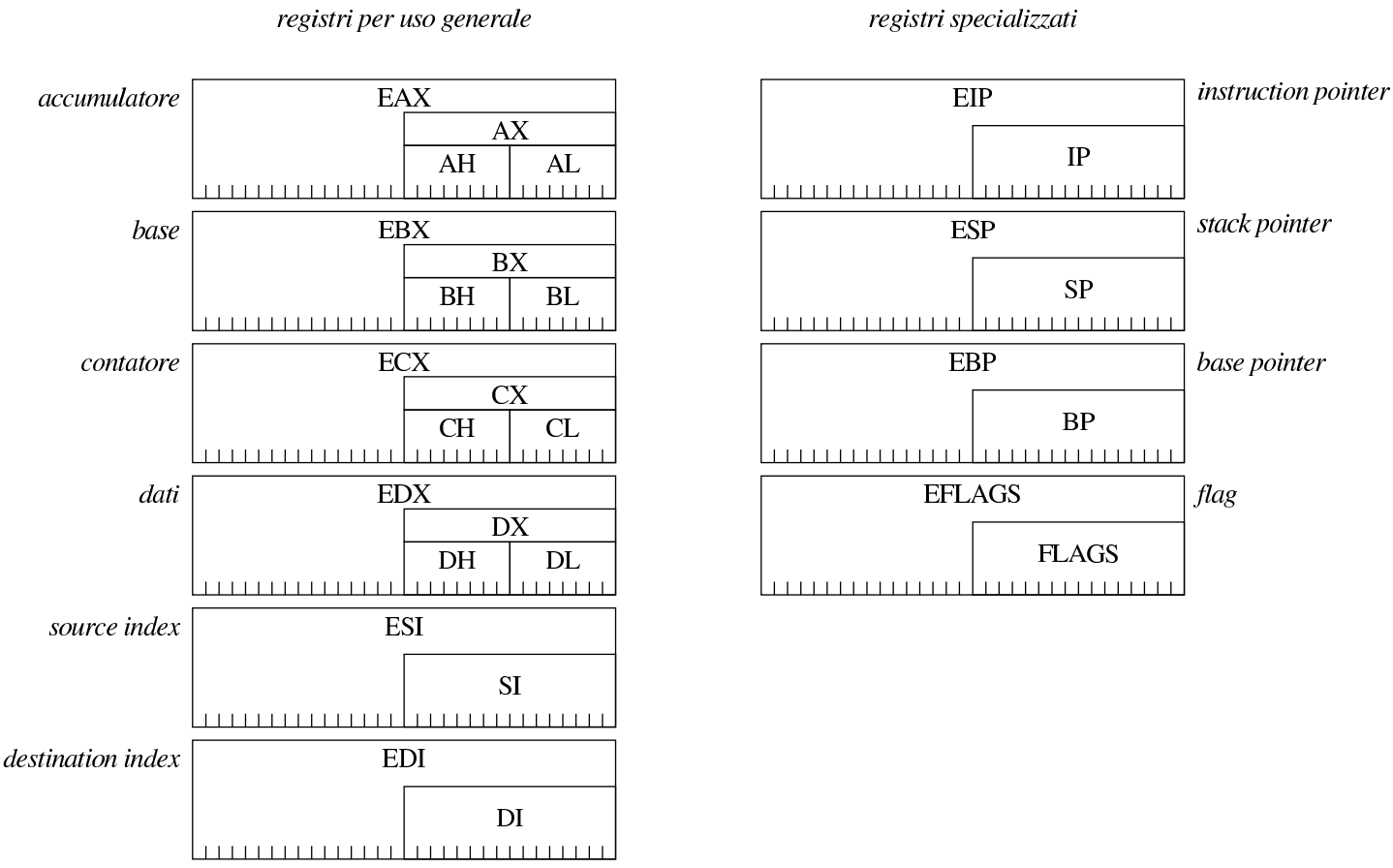


Tabella 64.3. Gli indicatori principali, contenuti nel registro ***FLAGS***.

Indicatore (*flag*)	Descrizione
C *carry*	È l'indicatore del riporto per le operazioni con valori senza segno. In particolare si attiva dopo una somma che genera un riporto e dopo una sottrazione che richiede il prestito di una cifra (in tal caso si chiama anche *borrow*).
O *overflow*	È l'indicatore di traboccamento per le operazioni che riguardano valori con segno.
Z *zero*	Viene impostato dopo un'operazione che dà come risultato il valore zero.

Indicatore (*flag*)	Descrizione
S *sign*	Riproduce il bit più significativo di un valore, dopo un'operazione. Se il valore è da intendersi con segno, l'indicatore serve a riprodurre il segno stesso.
P *parity*	Si attiva quando l'ultima operazione produce un risultato i cui otto bit meno significativi contengono una quantità pari di cifre a uno.

64.3 Sintesi delle istruzioni principali

«

Nelle tabelle successive vengono annotate le istruzioni più semplici che possono essere utilizzate con i microprocessori x86, raggruppate secondo il contesto a cui appartengono. In modo particolare sono assenti le istruzioni per i calcoli in virgola mobile e quelle per la gestione delle stringhe.

L'ordine in cui sono specificati gli operandi è quello «Intel», ovvero appare prima la destinazione e poi l'origine. Le sigle usate per definire i tipi di operandi sono: ***reg*** per «registro»; ***mem*** per «memoria»; ***imm*** per «immediato» (costante numerica).

Nella colonna degli indicatori appare: il simbolo «#» per annotare che l'indicatore relativo può essere modificato dall'istruzione; il simbolo «t» per annotare che lo stato precedente dell'indicatore viene considerato dall'istruzione; zero o uno se l'indicatore viene impostato in un certo modo; il simbolo «?» se l'effetto dell'istruzione sull'indicatore è indefinito.

Tabella 64.4. Assegnamenti, scambi, conversioni e istruzione nulla.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`NOP`		Istruzione nulla.	`c p z s o` `. . . . .`
`MOV`	***reg***, ***reg*** ***reg***, ***mem*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	Copia il valore dell'origine nella destinazione. Origine e destinazione devono avere la stessa quantità di bit. ***dst*** := ***org***	`c p z s o` `. . . . .`
`LEA`	***reg32***, ***mem***	*load effective address* Mette nel registro l'indirizzo della memoria. ***dst*** := indirizzo(***org***)	`c p z s o` `. . . . .`
`MOVSX`	***reg16***, ***reg8*** ***reg16***, ***mem8*** ***reg32***, ***reg8*** ***reg32***, ***mem8*** ***reg32***, ***reg16*** ***reg32***, ***mem16***	Tratta il valore nell'origine come un numero con segno e lo estende in modo da occupare tutto lo spazio della destinazione.	`c p z s o` `. . . . .`
`MOVZX`	***reg16***, ***reg8*** ***reg16***, ***mem8*** ***reg32***, ***reg8*** ***reg32***, ***mem8*** ***reg32***, ***reg16*** ***reg32***, ***mem16***	Tratta il valore nell'origine come un numero senza segno e lo estende in modo da occupare tutto lo spazio della destinazione.	`c p z s o` `. . . . .`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`XCHG`	***reg***, ***reg*** ***reg***, ***mem*** ***mem***, ***reg***	Scambia i valori. ***dst*** :==: ***org***	`c p z s o` `. . . . .`
`CBW`		Converte un intero con segno, della dimensione di 8 bit, contenuto in ***AL***, in modo da occupare tutto ***AX*** (da 8 bit a 16 bit). L'espansione tiene conto del segno. ***AX*** := ***AL***	`c p z s o` `. . . . .`
`CWDE`		Converte un intero con segno, della dimensione di 16 bit, contenuto in ***AX***, in modo da occupare ***EAX*** (da 16 bit a 32 bit). L'espansione tiene conto del segno. ***EAX*** := ***AX***	`c p z s o` `. . . . .`
`CDQ`		Converte un intero con segno, della dimensione di 32 bit, contenuto in ***EAX***, in modo da occupare la somma di ***EDX***:***EAX*** (da 32 bit a 64 bit). L'espansione tiene conto del segno. ***EDX***:***EAX*** := ***EAX***	`c p z s o` `. . . . .`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`BSWAP`	***reg32***	Inverte l'ordine dei byte contenuti nel registro: quello meno significativo diventa il più significativo; la coppia interna si scambia.	`c p z s o` `. . . . .`

Tabella 64.5. Operazioni aritmetiche.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`NEG`	***reg*** ***mem***	Inverte il segno di un numero, attraverso il complemento a due. ***dst*** := **–*dst***	`c p z s o` `# # # # #`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`ADD`	***reg***, ***reg*** ***reg***, ***mem*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	Somma di interi, con o senza segno, ignorando il riporto precedente. Se i valori si intendono con segno, è importante l'esito dell'indicatore di traboccamento (*overflow*), se invece i valori sono da intendersi senza segno, è importante l'esito dell'indicatore di riporto (*carry*). ***dst*** := ***org*** + ***dst***	`c p z s o` `# # # # #`
`SUB`	***reg***, ***reg*** ***reg***, ***mem*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	Sottrazione di interi con o senza segno, ignorando il riporto precedente. ***dst*** := ***org*** − ***dst***	`c p z s o` `# # # # #`
`ADC`	***reg***, ***reg*** ***reg***, ***mem*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	Somma di interi, con o senza segno, aggiungendo anche il riporto precedente (l'indicatore *carry*). ***dst*** := ***org*** + ***dst*** + ***c***	`c p z s o` `t . . . .` `# # # # #`
`SBB`	***reg***, ***reg*** ***reg***, ***mem*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	Sottrazione di interi, con o senza segno, tenendo conto del «prestito» precedente (l'indicatore *carry*). ***dst*** := ***org*** + ***dst*** − ***c***	`c p z s o` `t . . . .` `# # # # #`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`INC`	***reg*** ***mem***	Incrementa di una unità un intero. ***dst++***	`c p z s o` `· # # # #`
`DEC`	***reg*** ***mem***	Decrementa di una unità un valore intero. ***dst−−***	`c p z s o` `· # # # #`
`MUL`	***reg*** ***mem***	Moltiplicazione intera senza segno. L'operando è il moltiplicatore, mentre il moltiplicando è costituito da registri prestabiliti. ***AX*** := ***AL*src*** ***DX:AX*** := ***AX*src*** ***EDX:EAX*** := ***EAX*src***	`c p z s o` `# ? ? ? #`
`DIV`	***reg*** ***mem***	Divisione intera senza segno. L'operando è il divisore, mentre il dividendo è costituito da registri prestabiliti. ***AL*** := ***AX/src*** ***AH*** := ***resto*** ***AX*** := ***DX:AX/src*** ***DX*** := ***resto*** ***EAX*** := ***EDX:EAX/src*** ***EDX*** := ***resto***	`c p z s o` `? ? ? ? ?`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`IMUL`	***reg*** ***mem***	Moltiplicazione intera con segno. In questo caso l'operando è il moltiplicatore, mentre il moltiplicando è costituito da registri prestabiliti. ***AX*** := ***AL*src*** ***DX:AX*** := ***AX*src*** ***EDX:EAX*** := ***EAX*src***	`c p z s o` `# ? ? ? #`
`IDIV`	***reg*** ***mem***	Divisione intera con segno. L'operando è il divisore, mentre il dividendo è costituito da registri prestabiliti. ***AL*** := ***AX/src*** ***AH*** := ***resto*** ***AX*** := ***DX:AX/src*** ***DX*** := ***resto*** ***EAX*** := ***EDX:EAX/src*** ***EDX*** := ***resto***	`c p z s o` `? ? ? ? ?`

Tabella 64.6. Operazioni logiche.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`NOT`	***reg*** ***mem***	NOT di tutti i bit dell'operando. ***dst*** := NOT ***dst***	`c p z s o` `. . . . .`
`AND` `OR` `XOR`	***reg***, ***reg*** ***reg***, ***mem*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	AND, OR, o XOR, tra tutti i bit dei due operandi. ***dst*** := ***org*** AND ***dst*** ***dst*** := ***org*** OR ***dst*** ***dst*** := ***org*** XOR ***dst***	`c p z s o` `0 # # # 0`

Tabella 64.7. Scorrimenti e rotazioni.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`SHL` `SHR`	***reg***, 1 ***mem***, 1 ***reg*** ***mem***	Fa scorrere i bit, rispettivamente verso sinistra o verso destra (l'ultima cifra perduta finisce nell'indicatore del riporto). Se appare un solo operando, la rotazione viene eseguita ***CL*** volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	`c p z s o` `# · · · #`
`SAL`	***reg***, 1 ***mem***, 1 ***reg*** ***mem***	Funziona esattamente come '**`SHL`**', ma esiste in quanto è la controparte di '**`SAR`**', riferendosi a uno scorrimento aritmetico.	`c p z s o` `# · · · #`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`SAR`	***reg***, 1 ***mem***, 1 ***reg*** ***mem***	Fa scorrere i bit verso destra (l'ultima cifra perduta finisce nell'indicatore del riporto), mantenendo il segno originale. Se appare un solo operando, la rotazione viene eseguita ***CL*** volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	`c p z s o` `# · · · ·`
`RCL` `RCR`	***reg***, 1 ***mem***, 1 ***reg*** ***mem***	Ruota i bit, rispettivamente verso sinistra o verso destra, utilizzando anche l'indicatore di riporto (*carry*). Se appare un solo operando, la rotazione viene eseguita ***CL*** volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	`c p z s o` `t · · · ·` `# · · · #`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`ROL` `ROR`	***reg***, 1 ***mem***, 1 ***reg*** ***mem***	Ruota i bit, rispettivamente verso sinistra o verso destra. Se appare un solo operando, la rotazione viene eseguita ***CL*** volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	`c p z s o` `# . . . #`

Tabella 64.8. Chiamate e gestione della pila.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`INT`	***imm8***	Esegue una chiamata attraverso un'interruzione.	`c p z s o` `. . . . .`
`CALL`	***reg*** ***mem*** ***imm***	Inserisce nella pila l'indirizzo dell'istruzione successiva e salta all'indirizzo indicato.	`c p z s o` `. . . . .`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`RET`		Estrae dalla pila l'indirizzo dell'istruzione da raggiungere e salta a quella (serve a concludere una chiamata eseguita con '**`CALL`**').	`c p z s o` `. . . . .`
`PUSH`	***reg*** ***mem***	Inserisce nella pila il valore (della dimensione di un registro comune).	`c p z s o` `. . . . .`
`POP`	***reg*** ***mem***	Estrae dalla pila l'ultimo valore inserito (della dimensione di un registro comune).	`c p z s o` `. . . . .`
`PUSHF`		Inserisce nella pila l'insieme del registro degli indicatori (***FLAGS*** o ***EFLAGS***).	`c p z s o` `. . . . .`
`POPF`		Estrae dalla pila l'insieme del registro degli indicatori (***FLAGS*** o ***EFLAGS***), aggiornando di conseguenza il registro stesso.	`c p z s o` `. . . . .`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`PUSHA` `PUSHAD`		Inserisce nella pila i registri principali: '**`PUSHA`**' inserisce nella pila i registri da 16 bit, mentre '**`PUSHAD`**' li inserisce a 32 bit. In sequenza vengono inseriti: ***AX***, ***CX***, ***DX***, ***BX***, ***SP***, ***BP***, ***SI***, ***DI***; ovvero: ***EAX***, ***ECX***, ***EDX***, ***EBX***, ***ESP***, ***EBP***, ***ESI***, ***EDI***. Si osservi che di solito si usa solo '**`PUSHA`**', lasciando al compilatore la responsabilità di scegliere quale istruzione è appropriata per il contesto.	`c p z s o` `. . . . .`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`POPA` `POPAD`		Ripristina i registri principali, estraendo i contenuti dalla pila: '**`POPA`**' ripristina i registri da 16 bit, mentre '**`POPAD`**' riguarda quelli da 32 bit. In sequenza vengono estratti: ***DI***, ***SI***, ***BP***, ***SP*** viene eliminato senza aggiornare il registro, ***BX***, ***DX***, ***CX***, ***AX***; ovvero: ***EDI***, ***ESI***, ***EBP***, ***ESP*** viene eliminato senza aggiornare il registro, ***EBX***, ***EDX***, ***ECX***, ***EAX***. Come si vede, anche se '**`PUSHA`**' e '**`PUSHAD`**' salvano l'indice della pila, in pratica questo indice non viene ripristinato. Si osservi che di solito si usa solo '**`POPA`**', lasciando al compilatore la responsabilità di scegliere quale istruzione è appropriata per il contesto.	`c p z s o` `. . . . .`

Tabella 64.9. Indicatori e confronti.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`CLC`		Azzera l'indicatore del riporto (*carry*), senza intervenire negli altri indicatori.	`c p z s o` `0 . . . .`
`CMC`		Inverte il valore dell'indicatore del riporto (*carry*).	`c p z s o` `# . . . .`
`CMP`	***reg***, ***reg*** ***reg***, ***mem*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	Confronta due valori interi. La comparazione avviene simulando la sottrazione dell'origine dalla destinazione, senza però modificare gli operandi, ma aggiornando gli indicatori, come se fosse avvenuta una sottrazione vera e propria. ***dst*** - ***org***	`c p z s o` `# # # # #`
`TEST`	***reg***, ***reg*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	AND dei due valori senza conservare il risultato. Serve solo a ottenere l'aggiornamento degli indicatori. ***dst*** AND ***org***	`c p z s o` `0 # # # 0`

Tabella 64.10. Salti.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`JMP`	***reg*** ***mem*** ***imm***	Salto incondizionato all'indirizzo indicato.	`c p z s o` `. . . . .`
`JA` `JNBE`	***imm***	Dopo un confronto di valori senza segno, salta se la destinazione era maggiore dell'origine. CMP ***dst***, ***org*** IF ***dst > org*** THEN go to ***imm***	`c p z s o` `t . t . .`
`JAE` `JNB`	***imm***	Dopo un confronto di valori senza segno, salta se la destinazione era maggiore o uguale all'origine. CMP ***dst***, ***org*** IF ***dst >= org*** THEN go to ***imm***	`c p z s o` `t . t . .`
`JB` `JNAE`	***imm***	Dopo un confronto di valori senza segno, salta se la destinazione era minore dell'origine. CMP ***dst***, ***org*** IF ***dst < org*** THEN go to ***imm***	`c p z s o` `t . t . .`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`JBE` `JNA`	***imm***	Dopo un confronto di valori senza segno, salta se la destinazione era minore o uguale all'origine. CMP ***dst***, ***org*** IF ***dst <= org*** THEN go to ***imm***	`c p z s o` `t · t · ·`
`JE`	***imm***	Dopo un confronto, indipendentemente dal segno, salta se la destinazione era uguale all'origine. CMP ***dst***, ***org*** IF ***dst == org*** THEN go to ***imm***	`c p z s o` `· · t · ·`
`JNE`	***imm***	Dopo un confronto, indipendentemente dal segno, salta se la destinazione era diversa dall'origine. CMP ***dst***, ***org*** IF ***dst =/= org*** THEN go to ***imm***	`c p z s o` `· · t · ·`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`JG` `JNLE`	***imm***	Dopo un confronto con segno, salta se la destinazione era maggiore dell'origine. CMP ***dst***, ***org*** IF ***dst > org*** THEN go to ***imm***	`c p z s o` `· · t t t`
`JGE` `JNL`	***imm***	Dopo un confronto con segno, salta se la destinazione era maggiore o uguale all'origine. CMP ***dst***, ***org*** IF ***dst >= org*** THEN go to ***imm***	`c p z s o` `· · t t t`
`JL` `JNGE`	***imm***	Dopo un confronto con segno, salta se la destinazione era minore dell'origine. CMP ***dst***, ***org*** IF ***dst < org*** THEN go to ***imm***	`c p z s o` `· · t t t`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`JLE` `JNG`	***imm***	Dopo un confronto con segno, salta se la destinazione era minore o uguale all'origine. CMP ***dst***, ***org*** IF ***dst <= org*** THEN go to ***imm***	`c p z s o` `· · t t t`
`JC` `JNC`	***imm***	Salta se l'indicatore del riporto (*carry*), rispettivamente, è attivo, oppure non è attivo.	`c p z s o` `t · · · ·`
`JO` `JNO`	***imm***	Salta se l'indicatore di traboccamento (*overflow*), rispettivamente, è attivo, oppure non è attivo.	`c p z s o` `· · · · t`
`JS` `JNS`	***imm***	Salta se l'indicatore di segno (*sign*), rispettivamente, è attivo, oppure non è attivo.	`c p z s o` `· · · t ·`
`JZ` `JNZ`	***imm***	Salta se l'indicatore di zero, rispettivamente, è attivo, oppure non è attivo.	`c p z s o` `· · t · ·`
`JP` `JNP`	***imm***	Salta se l'indicatore di parità, rispettivamente, è attivo, oppure non è attivo.	`c p z s o` `· t · · ·`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`JCXZ`	***imm***	Salta se il valore contenuto nel registro ***CX*** è pari a zero.	`c p z s o` `. . . . .`

Tabella 64.11. Impostazione del valore in base all'esito di un confronto.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`SETA` `SETNBE`	***reg8*** ***mem8***	Dopo un confronto di valori senza segno, imposta a uno il registro o la memoria indicati, se la destinazione era maggiore dell'origine. CMP ***dst***, ***org*** IF ***dst > org*** THEN ***x***:=1 ELSE ***x***:=0	`c p z s o` `. . . . .`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`SETAE` `SETNB`	***reg8*** ***mem8***	Dopo un confronto di valori senza segno, imposta a uno il registro o la memoria indicati, se la destinazione era maggiore o uguale all'origine. CMP ***dst***, ***org*** IF ***dst >= org*** THEN ***x***:=1 ELSE ***x***:=0	`c p z s o` `. . . . .`
`SETB` `SETNAE`	***reg8*** ***mem8***	Dopo un confronto di valori senza segno, imposta a uno il registro o la memoria indicati, se la destinazione era minore dell'origine. CMP ***dst***, ***org*** IF ***dst < org*** THEN ***x***:=1 ELSE ***x***:=0	`c p z s o` `. . . . .`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`SETBE` `SETNA`	***reg8*** ***mem8***	Dopo un confronto di valori senza segno, imposta a uno il registro o la memoria indicati, se la destinazione era minore o uguale all'origine. CMP ***dst***, ***org*** IF ***dst*** <= ***org*** THEN ***x***:=1 ELSE ***x***:=0	`c p z s o` `. . . . .`
`SETE`	***reg8*** ***mem8***	Dopo un confronto, indipendentemente dal segno, imposta a uno il registro o la memoria indicati, se la destinazione era uguale all'origine. CMP ***dst***, ***org*** IF ***dst*** == ***org*** THEN ***x***:=1 ELSE ***x***:=0	`c p z s o` `. . . . .`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`SETNE`	***reg8*** ***mem8***	Dopo un confronto, indipendentemente dal segno, imposta a uno il registro o la memoria indicati, se la destinazione era diversa dall'origine. CMP ***dst***, ***org*** IF ***dst =/= org*** THEN ***x***:=1 ELSE ***x***:=0	`c p z s o` `. . . . .`
`SETG` `SETNLE`	***reg8*** ***mem8***	Dopo un confronto con segno, imposta a uno il registro o la memoria indicati, se la destinazione era maggiore dell'origine. CMP ***dst***, ***org*** IF ***dst > org*** THEN ***x***:=1 ELSE ***x***:=0	`c p z s o` `. . . . .`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`SETGE` `SETNL`	***reg8*** ***mem8***	Dopo un confronto con segno, imposta a uno il registro o la memoria indicati, se la destinazione era maggiore o uguale all'origine. CMP ***dst***, ***org*** IF ***dst*** >= ***org*** THEN ***x***:=1 ELSE ***x***:=0	`c p z s o` `. . . . .`
`SETL` `SETNGE`	***reg8*** ***mem8***	Dopo un confronto con segno, imposta a uno il registro o la memoria indicati, se la destinazione era minore dell'origine. CMP ***dst***, ***org*** IF ***dst*** < ***org*** THEN ***x***:=1 ELSE ***x***:=0	`c p z s o` `. . . . .`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`SETLE` `SETNG`	***reg8*** ***mem8***	Dopo un confronto con segno, imposta a uno il registro o la memoria indicati, se la destinazione era minore o uguale all'origine. CMP ***dst***, ***org*** IF ***dst*** <= ***org*** THEN ***x***:=1 ELSE ***x***:=0	`c p z s o` `. . . . .`
`SETC` `SETNC`	***reg8*** ***mem8***	Imposta a uno il registro o la memoria indicati, se l'indicatore del riporto (*carry*), rispettivamente, è attivo, oppure non è attivo.	`c p z s o` `t . . . .`
`SETO` `SETNO`	***reg8*** ***mem8***	Imposta a uno il registro o la memoria indicati, se l'indicatore di traboccamento (*overflow*), rispettivamente, è attivo, oppure non è attivo.	`c p z s o` `. . . . t`
`SETS` `SETNS`	***reg8*** ***mem8***	Imposta a uno il registro o la memoria indicati, se l'indicatore di segno (*sign*), rispettivamente, è attivo, oppure non è attivo.	`c p z s o` `. . . t .`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`SETZ` `SETNZ`	***reg8*** ***mem8***	Imposta a uno il registro o la memoria indicati, se l'indicatore di zero, rispettivamente, è attivo, oppure non è attivo.	`c p z s o` `· · t · ·`

Tabella 64.12. Iterazioni

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`LOOP`	***imm8***	Senza alterare gli indicatori, decrementa di una unità il registro '**`ECX`**' (o solo '**`CX`**', se viene richiesta una dimensione più piccola), quindi, se il registro è ancora diverso da zero, salta all'indirizzo cui fa riferimento l'operando.	`c p z s o` `· · · · ·`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`LOOPE` `LOOPZ`	***imm8***	Senza alterare gli indicatori, decrementa di una unità il registro '**`ECX`**' (o solo '**`CX`**', se viene richiesta una dimensione più piccola), quindi, se il registro è ancora diverso da zero e l'indicatore «zero» è attivo, salta all'indirizzo cui fa riferimento l'operando.	`c p z s o` `. . t . .`
`LOOPNE` `LOOPNZ`	***imm8***	Senza alterare gli indicatori, decrementa di una unità il registro '**`ECX`**' (o solo '**`CX`**', se viene richiesta una dimensione più piccola), quindi, se il registro è ancora diverso da zero e l'indicatore «zero» non è attivo, salta all'indirizzo cui fa riferimento l'operando.	`c p z s o` `. . t . .`

64.4 Primo approccio al linguaggio assemblatore per x86

Per scrivere programmi con il linguaggio assemblatore in un sistema GNU/Linux, è necessario disporre del compilatore: serve GNU AS[1] (GAS) per la sintassi AT&T oppure NASM[2] per quella Intel.

64.4.1 Il primo programma

Viene mostrato il listato di un programma molto semplice, il cui scopo è unicamente quello di concludere il proprio funzionamento restituendo un valore, attraverso una funzione del sistema operativo. Ne vengono preparate due versioni: una adatta a GNU AS e l'altra per NASM.

Listato 64.13. File `fine-gas.s`, adatto per GNU AS.

```
1   #
2   # fine-gas.s
3   #
4   .section .data
5   #
6   .section .text
7   .globl _start
8   #
9   _start:
10      mov  $1, %eax
11      mov  $7, %ebx
12      int  $0x80
```

Listato 64.14. File ‘`fine-nasm.s`’, adatto per NASM.

```
1   ;
2   ; fine-nasm.s
3   ;
4   section .data
5   ;
6   section .text
7   global _start
8   ;
9   _start:
10      mov eax, 1
11      mov ebx, 7
12      int 0x80
```

Per compilare il file si genera prima un file oggetto, quindi si passa per il *linker* (il «collegatore»), ovvero il programma che collega i file oggetto che devono comporre il file eseguibile finale. Si comincia con il sorgente per GNU AS:

```
$ as -o fine-gas.o fine-gas.s[Invio]

$ ld -o fine-gas fine-gas.o[Invio]
```

Per quanto riguarda il sorgente per NASM:

```
$ nasm -f elf -o fine-nasm.o fine-nasm.s[Invio]

$ ld -o fine-nasm fine-nasm.o[Invio]
```

In entrambi gli esempi, viene generato un file oggetto in formato ELF (*Executable and linkable format*), cosa che deve essere richiesta esplicitamente nel secondo caso, mentre nel primo è implicita. Pertanto, come si vede, il programma ‘**`ld`**’ viene usato sempre nello stesso modo.

Il programma generato (‘**fine-gas**’ o ‘**fine-nasm**’) si limita a chiamare una funzione del sistema operativo, con la quale conclude il suo lavoro restituendo il valore numerico sette. Lo si può verificare ispezionando il parametro ‘**$?**’ della shell:

```
$ ./fine-gas [Invio]

$ echo $? [Invio]

  7
```

I due programmi sono perfettamente uguali nel modo di disporsi nelle righe del file sorgente, pertanto vengono descritti senza fare distinzioni.

Le prime tre righe sono commenti, ignorati dal compilatore; la quarta riga contiene una direttiva per il compilatore che lo avverte dell'inizio della zona usata per descrivere l'uso della memoria (che in questo caso non viene usata affatto); la quinta riga è un altro commento; la sesta riga avverte il compilatore dell'inizio del codice del programma.

La settima riga serve ad avvisare il compilatore che il simbolo rappresentato dall'etichetta denominata ‘**_start**’ individua l'indirizzo dell'istruzione iniziale da rendere pubblica, pertanto deve rimanere rintracciabile nel file oggetto generato dalla compilazione. L'ottava riga è un altro commento e la nona riga dichiara il simbolo ‘**_start**’ già nominato.

Per il compilatore, l'etichetta '**_start**' indica convenzionalmente il punto di inizio del programma e il simbolo corrispondente deve essere reso pubblico, perché '**ld**' deve sapere da che parte si comincia (soprattutto quando più file oggetto si collegano assieme in un solo eseguibile).

Nella decima riga si assegna il valore uno al registro ***EAX*** e nell'undicesima si assegna il valore sette al registro ***EBX***; infine, nell'ultima riga, si esegue un'interruzione all'indirizzo 80_{16}. In pratica, le ultime tre righe servono a eseguire la chiamata di una funzione del sistema operativo. La funzione è individuata dal numero uno che deve essere collocato nel registro ***EAX*** e il parametro, rappresentato dal valore di uscita, da collocare nel registro ***EBX***. La chiamata effettiva avviene con l'interruzione all'indirizzo 80_{16}.

Dopo aver compilato il programma e ottenuto il file eseguibile, si può dare un'occhiata al suo contenuto con l'aiuto di Objdump:[3]

```
$ objdump --disassemble fine-gas[Invio]

 fine-gas:     file format elf32-i386

 Disassembly of section .text:

 08048074 <_start>:
  8048074:       b8 01 00 00 00          mov    $0x1,%eax
  8048079:       bb 07 00 00 00          mov    $0x7,%ebx
  804807e:       cd 80                   int    $0x80
```

Usato in questo modo, Objdump disassembla il programma e mostra anche le istruzioni nel linguaggio macchina vero e proprio, con gli indirizzi di memoria virtuale che verrebbero utilizzati durante il

funzionamento. Eventualmente si può richiedere espressamente di disassemblare utilizzando una notazione Intel:

```
$ objdump --disassemble -M intel fine-gas[Invio]

 fine-gas:     file format elf32-i386

 Disassembly of section .text:

 08048074 <_start>:
  8048074:       b8 01 00 00 00          mov    eax,0x1
  8048079:       bb 07 00 00 00          mov    ebx,0x7
  804807e:       cd 80                   int    0x80
```

64.4.2 Utilizzo di GDB

Per poter iniziare lo studio di un linguaggio assemblatore è praticamente indispensabile saper utilizzare un *debugger*, ovvero uno strumento che permetta di eseguire passo per passo il proprio programma, consentendo di verificare lo stato dei registri ed eventualmente della memoria. Infatti, con un linguaggio assemblatore, operazioni «semplici» come l'emissione di informazioni attraverso lo schermo diventano invece molto complicate.

Nei sistemi GNU è disponibile GDB (GNU debugger)[4]. Per capire come utilizzarlo, si prenda nuovamente l'esempio di programma introduttivo, aggiungendo qualche piccola modifica:

```
1   #
2   # fine-gas.s
3   #
4   .section .data
5   #
6   .section .text
7   .globl _start
8   #
9   _start:
10      mov  $1, %eax
11  bp1:
12      mov  $7, %ebx
13  bp2:
14      int  $0x80
```

```
1   ;
2   ; fine-nasm.s
3   ;
4   segment .data
5   ;
6   segment .text
7   global _start
8   ;
9   _start:
10      mov eax, 1
11  bp1:
12      mov ebx, 7
13  bp2:
14      int 0x80
```

Rispetto al file originale sono state aggiunge due etichette, '**bp1**' e '**bp2**' (il cui nome è stato scelto arbitrariamente, ma in questo caso ricorda il termine *breakpoint*), collocate tra le istruzioni che si traducono in codici del linguaggio macchina. Naturalmente, il file

va ricompilato nel modo già descritto; poi, una volta ottenuto il file eseguibile, lo si avvia all'interno di GDB:

```
$ gdb fine-gas [Invio]

  GNU gdb 6.5-debian
  Copyright (C) 2006 Free Software Foundation, Inc.
  GDB is free software, covered by the GNU General Public
  License, and you are welcome to change it and/or distribute
  copies of it under certain conditions. Type "show copying"
  to see the conditions.
  There is absolutely no warranty for GDB.  Type "show
  warranty" for details. This GDB was configured as
  "i486-linux-gnu"...(no debugging symbols found)
  Using host libthread_db library
  "/lib/tls/libthread_db.so.1".
```

A questo punto GDB presenta un invito, dal quale è possibile inserire dei comandi in modo interattivo.

```
(gdb)
```

Di solito, la prima cosa da fare consiste nel definire degli «stop» (*breakpoint*), dove il programma deve essere fermato automaticamente. È per questa ragione che sono state aggiunte delle etichette nel sorgente: per poter associare a quei simboli dei punti di sospensione dell'esecuzione.

```
(gdb) break bp1 [Invio]

  Breakpoint 1 at 0x8048079

(gdb) break bp2 [Invio]

  Breakpoint 2 at 0x804807e
```

Una volta fissati gli stop, si può «avviare» il programma, che viene così sospeso in corrispondenza del primo di questi punti:

```
(gdb) run [Invio]

  Starting program: /home/tizio/fine-gas
  (no debugging symbols found)

  Breakpoint 1, 0x08048079 in bp1 ()
```

Si ispezionano i registri:

```
(gdb) info registers [Invio]

  eax            0x1      1
  ecx            0x0      0
  edx            0x0      0
  ebx            0x0      0
  esp            0xbff50080       0xbff50080
  ebp            0x0      0x0
  esi            0x0      0
  edi            0x0      0
  eip            0x8048079        0x8048079 <bp1>
  eflags         0x292    [ AF SF IF ]
  cs             0x73     115
  ss             0x7b     123
  ds             0x7b     123
  es             0x7b     123
  fs             0x0      0
  gs             0x0      0
```

Si può verificare che il registro ***EAX*** contiene il valore uno, come dovrebbe effettivamente in questa posizione. Per far proseguire il programma fino al prossimo stop si usa il comando '**continue**':

```
(gdb) continue [Invio]
```

```
Breakpoint 2, 0x0804807e in bp2 ()
```

Si ispezionano nuovamente i registri:

```
(gdb) info registers[Invio]

eax            0x1      1
ecx            0x0      0
edx            0x0      0
ebx            0x7      7
esp            0xbfcefe20       0xbfcefe20
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x804807e        0x804807e <bp2>
eflags         0x292    [ AF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x0      0
```

Si può vedere che a questo punto il registro ***EBX*** risulta impostato con il valore previsto. Si lascia concludere il programma e si termina l'attività con GDB:

```
(gdb) continue[Invio]

Continuing.

Program exited with code 07.
(gdb) quit[Invio]
```

Il procedimento descritto vale per il programma compilato nel modo

«normale», sia attraverso GNU AS, sia attraverso NASM. Ma per avere una visione più chiara di ciò che si fa, occorre abbinare il sorgente originale. Questo può essere fatto con GNU AS, utilizzando l'opzione '**--gstabs**', oppure con NASM mettendo l'opzione '**-g**'. Qui si mostra solo il caso di GNU AS:

```
$ as --gstabs -o fine-gas.o fine-gas.s[Invio]

$ ld -o fine-gas fine-gas.o[Invio]

$ gdb fine-gas[Invio]

  GNU gdb 6.5-debian
  Copyright (C) 2006 Free Software Foundation, Inc.
  GDB is free software, covered by the GNU General Public
  License, and you are welcome to change it and/or distribute
  copies of it under certain conditions. Type "show copying"
  to see the conditions.
  There is absolutely no warranty for GDB.  Type "show
  warranty" for details. This GDB was configured as
  "i486-linux-gnu"...(no debugging symbols found)
  Using host libthread_db library
  "/lib/tls/libthread_db.so.1".

(gdb) break bp1[Invio]

  Breakpoint 1 at 0x8048079: file fine-gas.s, line 12.

(gdb) break bp2[Invio]

  Breakpoint 2 at 0x804807e: file fine-gas.s, line 14.

(gdb) run[Invio]
```

```
Starting program: /home/tizio/fine-gas

Breakpoint 1, bp1 () at fine-gas.s:12
12          mov  $7, %ebx
Current language:  auto; currently asm
```

Come si vede dall'esempio, si ottengono più informazioni collegate al sorgente originale; in particolare si può sapere qual è la prossima istruzione che verrebbe eseguita. Questa volta si decide di procedere eseguendo un'istruzione alla volta, con il comando '**`stepi`**':

```
(gdb) stepi [Invio]

bp2 () at fine-gas.s:14
14          int  $0x80
```

Dato che il programma è molto breve, con la prossima istruzione si va a concluderne il funzionamento:

```
(gdb) stepi [Invio]

Program exited with code 07.
(gdb) quit [Invio]
```

Tabella 64.34. Alcuni comandi interattivi per GDB.

Comando	Descrizione
`set args` ***argomento*** …	Definisce gli argomenti della riga di comando del programma.
`r` [***argomento***] … `run` [***argomento***] …	Avvia l'esecuzione del programma, eventualmente con gli argomenti indicati.

Comando	Descrizione
`b` ***funzione*** `break` ***funzione***	Definisce uno stop all'esecuzione del programma in corrispondenza del simbolo indicata. Questo simbolo viene definito precisamente «funzione» in quanto riguarda la zona che descrive le istruzioni del programma e non quelle dei dati.
`b` ***riga*** `break` ***riga***	Definisce uno stop all'esecuzione del programma in corrispondenza della riga indicata, riferita al file sorgente. Per poter usare questo comando occorre compilare il programma con i riferimenti al file sorgente.
`cl` [***funzione***] `clear` [***funzione***]	Elimina uno stop associato a un certo simbolo, oppure, se c'è, elimina quello della posizione corrente.
`cl` ***riga*** `clear` ***riga***	Elimina uno stop associato alla riga indicata.
`d` [***n***] … `disable` [***n***] …	Disabilita gli stop indicati per numero, o tutti se non ne viene indicato alcuno.
`enable` [***n***] …	Riabilita gli stop indicati per numero, o tutti se non ne viene indicato alcuno.

Comando	Descrizione
`d` [*n*] … `delete` [*n*] …	Elimina gli stop indicati per numero, o tutti se non ne viene indicato alcuno.
`si` `stepi`	Esegue la prossima istruzione-macchina e poi si ferma nuovamente. Se l'istruzione è una chiamata di funzione, passa all'inizio della stessa.
`ni` `nexti`	Esegue la prossima istruzione-macchina e poi si ferma nuovamente. **Se l'istruzione è una chiamata di funzione, questa viene saltata.**
`c` `continue`	Riprende l'esecuzione di un programma dopo uno stop (che può poi fermarsi nuovamente allo stop successivo, se c'è).
`kill`	Interrompe definitivamente l'esecuzione del programma.
`i r` `info registers`	Mostra lo stato dei registri.
`bt` `backtrace`	Mostra lo stato della pila, precisamente mostra gli elementi che compongono lo *stack frame*.

Comando	Descrizione
`p` [*/f*] [(*tipo*)] ***variabile*** `print` [*/f*] [(*tipo*)] ***variabile***	Mostra il valore contenuto in memoria, in corrispondenza del simbolo specificato (***variabile***). Se non si tratta di una variabile scalare di dimensione «standard», occorre specificarne il formato, tra parentesi tonde, esattamente prima del nome; se si vuole visualizzare il valore contenuto nella variabile in modo diverso da quello predefinito, occorre aggiungere l'opzione '***/f***', dove la lettera ***f*** specifica il tipo di rappresentazione.
`p` */f* (***tipo*** [***n***]) ***variabile*** `print` */f* (***tipo*** [***n***]) ***variabile***	Se nella definizione del formato si mette un valore numerico tra parentesi quadre, si intende visualizzare ***n*** valori di quel tipo a partire dalla posizione indicata dal nome della variabile. Ciò consente di visualizzare il contenuto di un array.

64.4.3 Modalità «TUI»

«

GDB, se è stato compilato per includere tale funzionalità, può essere usato con l'opzione '**`-tui`**', con la quale si ottiene una suddivisione dello schermo in finestre:

```
$ gdb -tui fine-gas[Invio]
```

```
   +--fine-gas.s-----------------------------------------+
   |10          mov  $1, %eax                           |
   |11      bp1:                                        |
   |12          mov  $7, %ebx                           |
   |13      bp2:                                        |
   |14          int  $0x80                              |
   |15                                                  |
   |16                                                  |
   |17                                                  |
   |18                                                  |
   |19                                                  |
   |20                                                  |
   |21                                                  |
   |22                                                  |
   |23                                                  |
   +--------------------------------------------------+
exec No process In:                          Line: ??   PC: 0x0
GDB is free software, covered by the GNU General Public
License, and you are welcome to change it and/or distribute
copies of it under certain conditions. Type "show copying"
to see the conditions.
There is absolutely no warranty for GDB.  Type "show
warranty" for details. This GDB was configured as
"i486-linux-gnu"...Using host libthread_db library
"/lib/tls/libthread_db.so.1".
```

(gdb) **break bp1** [*Invio*]

(gdb) **break bp2** [*Invio*]

```
   +--fine-gas.s-----------------------------------------+
   |10          mov  $1, %eax                           |
   |11      bp1:                                        |
b+ |12          mov  $7, %ebx                           |
   |13      bp2:                                        |
b+ |14          int  $0x80                              |
   |15                                                  |
   |16                                                  |
   |17                                                  |
   |18                                                  |
   |19                                                  |
   |20                                                  |
   |21                                                  |
   |22                                                  |
   |23                                                  |
   +----------------------------------------------------+
exec No process In:                          Line: ??   PC: 0x0
This GDB was configured as "i486-linux-gnu"...Using host
libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) break bp1
Breakpoint 1 at 0x8048079: file fine-gas.s, line 12.
(gdb) break bp2
Breakpoint 2 at 0x804807e: file fine-gas.s, line 14.
```

(gdb) **run** [*Invio*]

```
   +--fine-gas.s-----------------------------------------+
   |10          mov  $1, %eax                           |
   |11      bp1:                                        |
B+>|12          mov  $7, %ebx                           |
   |13      bp2:                                        |
b+ |14          int  $0x80                              |
   |15                                                  |
   |16                                                  |
   |17                                                  |
   |18                                                  |
   |19                                                  |
   |20                                                  |
   |21                                                  |
   |22                                                  |
   |23                                                  |
   +----------------------------------------------------+
child process 9699 In: bp1          Line: 12   PC: 0x8048079
(gdb) break bp2
Breakpoint 2 at 0x804807e: file fine-gas.s, line 14.
(gdb) run
Starting program: /home/tizio/fine-gas

Breakpoint 1, bp1 () at fine-gas.s:12
Current language:  auto; currently asm
```

Durante il funzionamento si può anche attivare una finestra con lo stato dei registri, attraverso il comando '**`layout reg`**':

`(gdb)` **`layout reg`**[*Invio*]

```
+--Register group: general--------------------------------+
|eax            0x1      1                                |
|ecx            0x0      0                                |
|edx            0x0      0                                |
|ebx            0x0      0                                |
|esp            0xbfa77b90       0xbfa77b90               |
|ebp            0x0      0x0                              |
+-------------------------------------------------------+
   |10          mov  $1, %eax                            |
   |11      bp1:                                         |
B+>|12          mov  $7, %ebx                            |
   |13      bp2:                                         |
b+ |14          int  $0x80                               |
   |15                                                   |
   |16                                                   |
   +-----------------------------------------------------+
child process 9699 In: bp1          Line: 12   PC: 0x8048079
Breakpoint 2 at 0x804807e: file fine-gas.s, line 14.
(gdb) run
Starting program: /home/tizio/fine-gas

Breakpoint 1, bp1 () at fine-gas.s:12
Current language:  auto; currently asm
(gdb) layout reg
```

(gdb) **quit** [*Invio*]

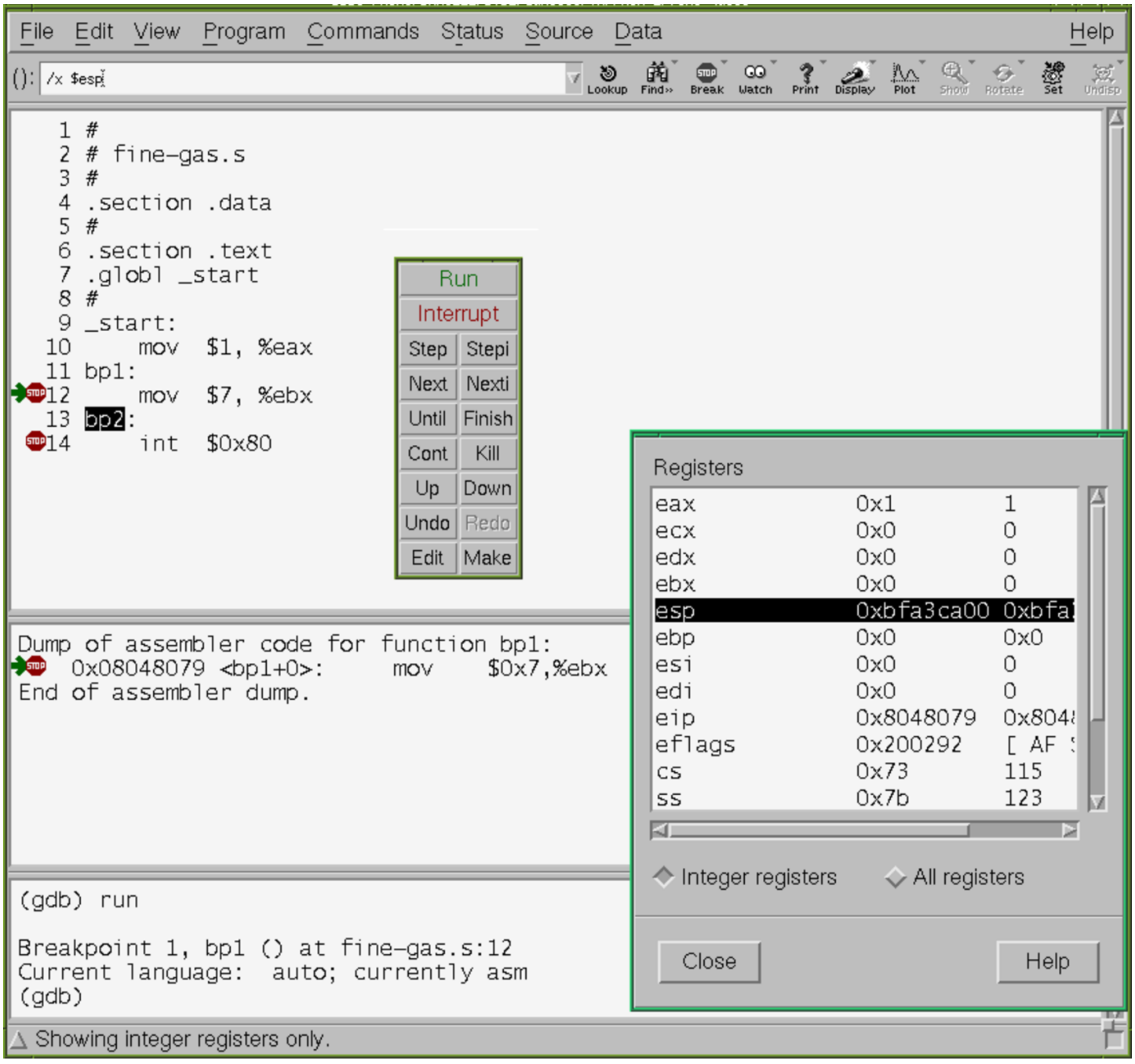
Durante il funzionamento normale di GDB, se è prevista tale modalità di funzionamento, questa può essere attivata anche con la combinazione di tasti [*Ctrl x*][*a*].

64.4.4 Utilizzo di DDD

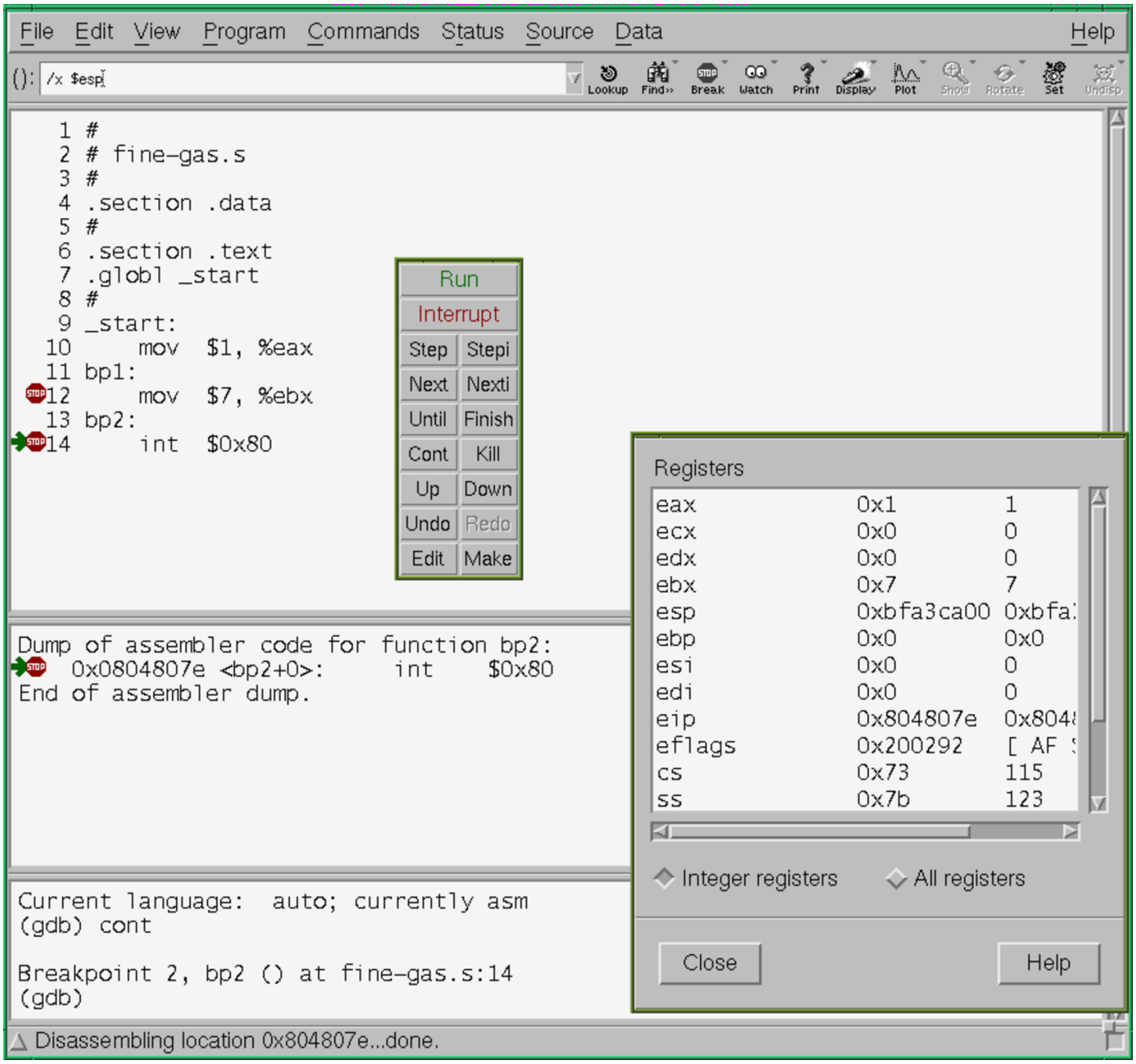
GDB è lo strumento fondamentale per il controllo del funzionamento di un programma, ma al suo fianco se ne possono aggiungere altri che consentono di avere una visione più «semplice». Per esempio, DDD,[5] ovvero *Data display debugger* è un programma frontale che si avvale di GDB, ma lo fa attraverso un'interfaccia grafica che consente di tenere sotto controllo più cose, simultaneamente.

```
$ ddd fine-gas [Invio]
```

Prendendo lo stesso esempio di programma già usato nella sezione precedente, compilato in modo da avere i riferimenti al sorgente, ecco come si può presentare DDD dopo che sono stati fissati gli stop e dopo che il programma è stato avviato, in modo che si arresti al primo di questi:



Ecco la situazione al livello del secondo stop:



64.4.5 Alcune istruzioni comuni

Le istruzioni del linguaggio assemblatore che si traducono direttamente in istruzioni del linguaggio macchina hanno una forma uniforme: un nome mnemonico seguito dagli operandi.

mnemonico ***operando*** [, ***operando***] ...

Per esempio, l'istruzione seguente sottrae il valore contenuto nel registro ***EAX*** da quello di ***EBX***, mettendo il risultato nel registro ***EBX*** (***EBX***=***EBX***-***EAX***):

GNU AS **sub %eax, %ebx**

NASM **sub ebx, eax**

In questo caso il nome mnemonico è '**SUB**', mentre i nomi dei registri sono gli operandi (eventualmente, il codice corrispondente in linguaggio macchina sarebbe $29C3_{16}$). Nelle sezioni successive si descrivono alcune istruzioni comuni.

64.4.5.1 MOV

L'istruzione '**MOV**' serve a copiare il contenuto dell'origine nella destinazione. Gli operandi possono essere registri, aree di memoria e costanti numeriche, tenendo conto che le costanti numeriche possono figurare solo nell'origine e che si può fare riferimento ad aree di memoria in una sola posizione (nell'origine o nella destinazione).

GNU AS **mov** ***origine*** **,** ***destinazione***

NASM **mov** ***destinazione*** **,** ***origine***

L'esempio seguente copia il contenuto del registro ***EAX*** all'interno di ***EBX***:

GNU AS **mov %eax, %ebx**

NASM **mov ebx, eax**

64.4.5.2 ADD

L'istruzione '**ADD**' serve a eseguire la somma di valori interi, con o senza segno. Gli operandi possono essere registri, aree di memoria e costanti numeriche, tenendo conto che le costanti numeriche possono figurare solo nell'origine e che si può fare riferimento ad aree di memoria in una sola posizione: nell'origine o nella destinazione.

GNU AS **add** ***origine*** **,** ***destinazione***

NASM **add** ***destinazione*** **,** ***origine***

L'esempio seguente aggiunge al registro ***EAX*** una unità:

GNU AS
```
add $1, %eax
```

NASM
```
add eax, 1
```

64.4.5.3 SUB

L'istruzione '**SUB**' serve a eseguire la sottrazione di valori interi, con o senza segno. Gli operandi possono essere registri, aree di memoria e costanti numeriche, tenendo conto che le costanti numeriche possono figurare solo nell'origine e che si può fare riferimento ad aree di memoria in una sola posizione: nell'origine o nella destinazione.

GNU AS **sub** ***origine*** **,** ***destinazione***

NASM **sub** ***destinazione*** **,** ***origine***

L'esempio seguente sottrae il valore del registro ***EAX*** da quello di ***EBX***, mettendo il risultato in ***EBX***:

GNU AS
```
sub %eax, %ebx
```

NASM
```
sub ebx, eax
```

64.4.5.4 INC e DEC

«

Le istruzioni '**INC**' e '**DEC**' servono, rispettivamente, a incrementare e a decrementare di una unità il valore dell'operando, che può essere un registro o un'area di memoria. Dal momento che c'è un solo operando, non c'è differenza tra la sintassi di GNU AS e di NASM per quanto riguarda l'ordine degli stessi:

inc ***destinazione***

dec ***destinazione***

I due esempi seguenti, rispettivamente, incrementano e decrementano di una unità il valore del registro ***EAX***:

- GNU AS **inc %eax**

  NASM **inc eax**

- GNU AS **dec %eax**

  NASM **dec eax**

64.4.6 Dimensione dei dati nelle istruzioni

«

Di norma, nelle istruzioni che elaborano dati, come quelle descritte nelle sezioni precedenti, se un registro si trova a essere origine o destinazione di qualcosa, implicitamente si intende che i dati debbano essere della sua dimensione. Per esempio, scrivendo '**mov $1, %eax**' si intende che la costante numerica sia precisamente $0000\ 0001_{16}$ (ovvero $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2$), perché la destinazione è da 32 bit.

Quando il contesto non è sufficiente a stabilire di quanti bit deve essere fatto il valore che si elabora, l'istruzione va precisata. Nel caso di GNU AS si aggiunge un suffisso al nome mnemonico che distingue l'operazione, suffisso che è composto da una sola lettera. Per esempio, '**mov**' diventa '**movl**' per chiarire che si tratta di dati da 32 bit. Nel caso di NASM si aggiunge una parola chiave dopo il nome mnemonico.

Tabella 64.41. Suffissi per precisare la quantità di bit coinvolti nelle istruzioni, secondo la sintassi di GNU AS. Negli esempi, '**mem**' è il nome di un simbolo (un'etichetta) che rappresenta un indirizzo di memoria.

Suffisso	Significato	Dimensione in bit	Esempi
`b`	byte	8 bit	**`mov $1 %al`** **`movb $1 mem`** **`pushb mem`**
`w`	*word*	16 bit	**`mov $1 %ax`** **`movw $1 mem`** **`pushw mem`**
`l`	*long*	32 bit	**`mov $1 %eax`** **`movl $1 mem`** **`pushl mem`**

Tabella 64.42. Parole chiave per precisare la quantità di bit coinvolti nelle istruzioni, secondo la sintassi di NASM. Negli esempi, '**mem**' è un simbolo che rappresenta un indirizzo di memoria.

Parola chiave	Dimensione in bit	Esempi
`byte`	8 bit	`mov al, 1` `mov byte [mem], 1` `push byte [mem]`
`word`	16 bit	`mov ax, 1` `mov word ax, 1` `mov word [mem], 1` `push word [mem]`
`long`	32 bit	`mov eax, 1` `mov long [mem], 1` `push long [mem]`

64.4.7 Direttive per il compilatore

Nel sorgente in linguaggio assemblatore è possibile inserire delle direttive (o pseudoistruzioni) che il compilatore può interpretare per la costruzione corretta del file oggetto. Si usano queste direttive in particolare per definire delle costanti e delle aree di memoria.

64.4.7.1 Commenti

Le righe vuote e quelle bianche vengono ignorate dal compilatore; inoltre, è possibile inserire dei commenti preceduti da un carattere che viene riconosciuto come prefisso, con il quale si annulla il significato di ciò che appare da quel punto fino alla fine della riga:

GNU AS **#*commento***

NASM **; *commento***

64.4.7.2 Direttiva «equ»

La direttiva '**equ**' viene usata per definire delle costanti simboliche (attraverso un nome, ovvero un'etichetta) da utilizzare poi nelle istruzioni, al posto del dato corrispondente. In linea di principio non dovrebbe essere possibile ridefinire le costanti.

GNU AS `.equ` *nome*`,` *valore*

NASM *nome* `equ` *valore*

Nell'esempio seguente si associa al simbolo '**ADDRESS**' il numero otto:

GNU AS
```
.equ ADDRESS, 8
```

NASM
```
ADDRESS equ 8
```

64.4.7.3 Direttive di dichiarazione dei dati

Attraverso delle direttive del compilatore si definiscono delle aree di memoria, a cui si fa riferimento nelle istruzioni attraverso dei nomi simbolici (etichette). In questo modo il compilatore attribuisce il loro indirizzo e lo sostituisce nella fase di assemblaggio. Si distingue tra dati che devono essere inizializzati preventivamente con un certo valore e dati il cui valore iniziale è indifferente. Quando si tratta di dati privi di valore iniziale, le informazioni necessarie consistono solo nel nome e nella dimensione dell'area di memoria:

GNU AS `.lcomm` *nome*`,` *dimensione_in_byte*

NASM *nome* `resb` *dimensione_in_byte*

NASM *nome* `resw` *dimensione_in_multipli_di_16_bit*

NASM ***nome*** **resd** ***dimensione_in_multipli_di_32_bit***

Segue la descrizione di alcuni esempi.

- GNU AS **.lcomm BUFFER, 500**

  NASM **BUFFER resb 500**

  Dichiara localmente un'area di memoria da 500 byte, nominata provvisoriamente '**BUFFER**'.

- GNU AS **.lcomm NUMERO, 4**

  NASM **NUMERO resd 1**

  Dichiara localmente un'area di memoria da 4 byte (32 bit), nominata provvisoriamente '**NUMERO**'.

Quando si tratta di dati da inizializzare, le informazioni necessarie alla dichiarazione consistono nel nome e nel contenuto, mentre la dimensione deriva dalla pseudoistruzione scelta:

GNU AS ***nome*** **:** **.ascii** ***stringa*** **[**, ***stringa*** **]** ...

GNU AS ***nome*** **:** **.byte** ***byte*** **[**, ***byte*** **]** ...

GNU AS ***nome*** **:** **.int** ***intero*** **[**, ***intero*** **]** ...

NASM ***nome*** **db** ***byte*** **[**, ***byte*** **]**

NASM ***nome*** **dd** ***intero*** **[**, ***intero*** **]**

Segue la descrizione di alcuni esempi.

- GNU AS **X1: .byte 65**

  NASM **X1 db 65**

Inizializza l'area di memoria identificata temporaneamente con il nome '**X1**', con il valore 65_{10}.

- GNU AS `X2: .byte 'A'`

  NASM `X2 db 'A'`

  Inizializza l'area di memoria identificata temporaneamente con il nome '**X2**', inserendo il codice corrispondente alla lettera «A».

- GNU AS `X3: .int 12345`

  NASM `X3 dd 12345`

  Inizializza l'area di memoria identificata temporaneamente con il nome '**X3**', inserendo il valore 12345.

- GNU AS `X4: .ascii 'Ciao!', 0`

  NASM `X4 db 'Ciao', 0`

  Inizializza l'area di memoria a partire dall'indirizzo identificato temporaneamente con il nome '**X3**', inserendo una stringa, terminata con il valore zero.

64.4.8 Sezioni del sorgente

Il sorgente si suddivide in sezioni, le quali descrivono la struttura del file-oggetto che si deve andare a produrre. Per quanto riguarda il formato ELF di un sistema GNU/Linux, si distingue l'area del codice, da quella dei dati; inoltre, nell'ambito dei dati si distingue la parte di quelli preinizializzati e di quelli che non lo sono.[6]

```
.section .data
    dichiarazione_dati_inizializzati
.section .bss
    dichiarazione_dati_non_definiti
.section .text
    istruzioni_del_programma
```

```
section .data
    dichiarazione_dati_inizializzati
section .bss
    dichiarazione_dati_non_definiti
section .text
    istruzioni_del_programma
```

I due modelli sintattici si riferiscono rispettivamente a GNU AS e a NASM.

64.4.9 Usare GNU AS con la notazione Intel

«

È possibile chiedere al compilatore GNU AS di interpretare il sorgente secondo la sintassi «Intel». Per questo si usa la direttiva '**`.intel_syntax noprefix`**' (l'opzione '**`noprefix`**' serve a consentire di annotare i nomi dei registri senza il prefisso '`%`'). L'esempio visto nella sezione 64.4.1 potrebbe essere modificato nel modo seguente:

```
#
.section .data
#
.section .text
.globl _start
#
.intel_syntax noprefix
_start:
    mov  eax, 1
    mov  ebx, 7
    int  0x80
```

La direttiva fa sì che il sorgente venga interpretato secondo la sintassi Intel a partire dal punto in cui si trova. Dal momento che l'effetto riguarda solo l'interpretazione delle istruzioni di codice che si traduce in linguaggio macchina (pertanto il modo di dare le altre direttive continua a essere quello normale di GNU AS), è conveniente piazzare la direttiva '**`.intel_syntax noprefix`**' all'inizio della sezione '**`.text`**', come si vede nell'esempio.

Naturalmente è possibile tornare alla sintassi AT&T con una direttiva analoga: '**`.att_syntax`**'. Lo si può vedere nell'esempio successivo:

```
#
.section .data
#
.section .text
.globl _start
#
.intel_syntax noprefix
_start:
    mov  eax, 1
.att_syntax
    mov  $7, %ebx
    int  $0x80
```

64.5 Esempi con le «quattro operazioni»

«

Vengono mostrati esempi di programmi estremamente banali, per dimostrare il funzionamento delle istruzioni con cui si eseguono le «quattro operazioni» su valori interi, attraverso l'aiuto di GDB. Per la compilazione, se si utilizza GNU AS è bene ricordare di inserire l'opzione '**--gstabs**', mentre con NASM è bene aggiungere l'opzione '**-g**', in modo da poter gestire più facilmente GDB, disponendo dei riferimenti al sorgente:

```
$ as --gstabs -o nome.o nome.s [Invio]
```

```
$ nasm -g -f elf -o nome.o nome.s [Invio]
```

Tabella 64.5. Operazioni aritmetiche.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`NEG`	***reg*** ***mem***	Inverte il segno di un numero, attraverso il complemento a due. ***dst*** := **−*dst***	`c p z s o` `# # # # #`
`ADD`	***reg***, ***reg*** ***reg***, ***mem*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	Somma di interi, con o senza segno, ignorando il riporto precedente. Se i valori si intendono con segno, è importante l'esito dell'indicatore di traboccamento (*overflow*), se invece i valori sono da intendersi senza segno, è importante l'esito dell'indicatore di riporto (*carry*). ***dst*** := ***org*** + ***dst***	`c p z s o` `# # # # #`
`SUB`	***reg***, ***reg*** ***reg***, ***mem*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	Sottrazione di interi con o senza segno, ignorando il riporto precedente. ***dst*** := ***org*** − ***dst***	`c p z s o` `# # # # #`
`ADC`	***reg***, ***reg*** ***reg***, ***mem*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	Somma di interi, con o senza segno, aggiungendo anche il riporto precedente (l'indicatore *carry*). ***dst*** := ***org*** + ***dst*** + ***c***	`c p z s o` `t . . . .` `# # # # #`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`SBB`	***reg***, ***reg*** ***reg***, ***mem*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	Sottrazione di interi, con o senza segno, tenendo conto del «prestito» precedente (l'indicatore *carry*). ***dst* := *org* + *dst* − *c***	`c p z s o` `t . . . .` `# # # # #`
`INC`	***reg*** ***mem***	Incrementa di una unità un intero. ***dst*++**	`c p z s o` `· # # # #`
`DEC`	***reg*** ***mem***	Decrementa di una unità un valore intero. ***dst*−−**	`c p z s o` `· # # # #`
`MUL`	***reg*** ***mem***	Moltiplicazione intera senza segno. L'operando è il moltiplicatore, mentre il moltiplicando è costituito da registri prestabiliti. ***AX* := *AL**src*** ***DX:AX* := *AX**src*** ***EDX:EAX* := *EAX**src***	`c p z s o` `# ? ? ? #`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`DIV`	***reg*** ***mem***	Divisione intera senza segno. L'operando è il divisore, mentre il dividendo è costituito da registri prestabiliti. ***AL*** := ***AX/src*** ***AH*** := ***resto*** ***AX*** := ***DX:AX/src*** ***DX*** := ***resto*** ***EAX*** := ***EDX:EAX/src*** ***EDX*** := ***resto***	`c p z s o` `? ? ? ? ?`
`IMUL`	***reg*** ***mem***	Moltiplicazione intera con segno. In questo caso l'operando è il moltiplicatore, mentre il moltiplicando è costituito da registri prestabiliti. ***AX*** := ***AL**src*** ***DX:AX*** := ***AX**src*** ***EDX:EAX*** := ***EAX**src***	`c p z s o` `# ? ? ? #`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`IDIV`	***reg*** ***mem***	Divisione intera con segno. L'operando è il divisore, mentre il dividendo è costituito da registri prestabiliti. ***AL*** := ***AX/src*** ***AH*** := ***resto*** ***AX*** := ***DX:AX/src*** ***DX*** := ***resto*** ***EAX*** := ***EDX:EAX/src*** ***EDX*** := ***resto***	`c p z s o` `? ? ? ? ?`

64.5.1 Somma

«

Viene proposto un programma che si limita a sommare due numeri (interi positivi) definiti in memoria e a restituire il risultato (ammesso che non sia troppo grande) attraverso il valore di uscita. Il programma viene mostrato sia nella forma adatta a GNU AS, sia in quella conforme a NASM. Le righe dei due listati coincidono.

```
1  # op1 + op2
2  #
3  .section .data
4  op1:    .int    15
5  op2:    .int     5
6  #
7  .section .text
8  .globl _start
9  #
```

```
10  _start:
11      mov op1, %edx   # Accumula il primo addendo in EDX.
12      add op2, %edx   # Somma il secondo addendo in EDX.
13  bp1:
14      mov $1,   %eax  # Restituisce il valore contenuto
15      mov %edx, %ebx  # in EDX come valore di uscita,
16      int $0x80       # attraverso la chiamata di sistema
17                      # 1 (exit).
```

```
1   ; op1 + op2
2   ;
3   section .data
4   op1:    dd 15
5   op2:    dd  5
6   ;
7   section .text
8   global _start
9   ;
10  _start:
11      mov edx, [op1]  ; Accumula il primo addendo in EDX.
12      add edx, [op2]  ; Somma il secondo addendo in EDX.
13  bp1:
14      mov eax, 1      ; Restituisce il valore contenuto
15      mov ebx, edx    ; in EDX come valore di uscita,
16      int 80h         ; attraverso la chiamata di sistema
17                      ; 1 (exit).
```

Nelle righe 4 e 5 vengono dichiarate due aree di memoria della dimensione di un registro (32 bit), associando rispettivamente i nomi '**`op1`**' e '**`op2`**', a indicare il primo e il secondo operando della somma. Nella riga 11 il contenuto della memoria che rappresenta il primo operando della somma, viene inserito nel registro ***EDX***, mentre nella riga 12 si somma quanto è rappresentato dal secondo operando,

nello stesso registro.

Si osservi che per indicare l'indirizzo di memoria è stata usata la modalità di indirizzamento diretta. In pratica, il compilatore sostituisce i nomi '**op1**' e '**op2**' con l'indirizzo di memoria a cui fanno riferimento.

Al termine, nelle righe da 14 a 16, si prepara la chiamata di sistema '**exit**', passando il risultato in modo che venga usato come valore di uscita. Se il risultato della somma è inferiore o uguale a 255, può essere letto.

I programmi sono uguali, a parte qualche piccola differenza nell'allocazione della memoria. Si può controllare con Objdump. Si suppone che il programma sia stato compilato con il nome 'add':

```
$ objdump --disassemble add[Invio]

 add:     file format elf32-i386

 Disassembly of section .text:

 08048074 <_start>:
  8048074:       8b 15 8c 90 04 08       mov    0x804908c,%edx
  804807a:       03 15 90 90 04 08       add    0x8049090,%edx

 08048080 <bp1>:
  8048080:       b8 01 00 00 00          mov    $0x1,%eax
  8048085:       89 d3                   mov    %edx,%ebx
  8048087:       cd 80                   int    $0x80
```

Si può controllare il funzionamento del programma, avviandolo e

verificando poi il valore di uscita:

```
$ ./add ; echo $?[Invio]

  20
```

Si analizza il funzionamento del programma con GDB:

```
$ gdb add[Invio]

(gdb) break bp1[Invio]

  Breakpoint 1 at 0x8048080: file add.s, line 14.
(gdb) run[Invio]

  Starting program: /home/tizio/add

  Breakpoint 1, bp1 () at add.s:14
  14          mov $1,   %eax  # Restituisce il valore contenuto in EDX
  Current language:  auto; currently asm


(gdb) info registers[Invio]

  eax            0x0      0
  ecx            0x0      0
  edx            0x14     20
  ebx            0x0      0
  esp            0xbf9dcb10       0xbf9dcb10
  ebp            0x0      0x0
  esi            0x0      0
  edi            0x0      0
  eip            0x8048080        0x8048080 <bp1>
  eflags         0x216    [ PF AF IF ]
  cs             0x73     115
  ss             0x7b     123
  ds             0x7b     123
  es             0x7b     123
```

```
 fs             0x0      0
 gs             0x0      0
```

(gdb) **stepi**[*Invio*]

```
 bp1 () at add.s:15
 15          mov %edx, %ebx  # come valore di uscita, attraverso
```

(gdb) **stepi**[*Invio*]

```
 bp1 () at add.s:16
 16          int $0x80  # chiamata di sistema 1 (exit).
```

(gdb) **info registers**[*Invio*]

```
 eax            0x1      1
 ecx            0x0      0
 edx            0x14     20
 ebx            0x14     20
 esp            0xbfba64d0       0xbfba64d0
 ebp            0x0      0x0
 esi            0x0      0
 edi            0x0      0
 eip            0x8048087        0x8048087 <bp1+7>
 eflags         0x216    [ PF AF IF ]
 cs             0x73     115
 ss             0x7b     123
 ds             0x7b     123
 es             0x7b     123
 fs             0x0      0
 gs             0x0      0
```

(gdb) **stepi**[*Invio*][7]

```
 Program exited with code 024.
```

```
(gdb) quit [Invio]
```

64.5.1.1 Traboccamento

Si può modificare leggermente il programma proposto, allo scopo di causare un traboccamento, in modo da vedere cosa accade nei registri:

```
3  .section .data
4  op1: .int 0x7FFFFFFF # 0b01111111111111111111111111111111
5  op2: .int 0x00000001 # 0b00000000000000000000000000000001
```

```
3  section .data
4  op1: dd 0x7FFFFFFF ; 01111111111111111111111111111111b
5  op2: dd 0x00000001 ; 00000000000000000000000000000001b
```

Compilando il programma ed eseguendolo con l'ausilio di GDB si può verificare che la somma di quei due valori trasforma il risultato in un valore apparentemente negativo, cosa che è indice di un traboccamento:

```
eax            0x0        0
ecx            0x0        0
edx            0x80000000       -2147483648
ebx            0x0        0
esp            0xbfa6f390       0xbfa6f390
ebp            0x0        0x0
esi            0x0        0
edi            0x0        0
eip            0x8048080        0x8048080 <bp1>
eflags         0xa96      [ PF AF SF IF OF ]
cs             0x73       115
ss             0x7b       123
ds             0x7b       123
es             0x7b       123
```

```
fs             0x0        0
gs             0x0        0
```

A ogni modo il fatto viene sottolineato dall'indicatore di traboccamento (*overflow*). In questo caso non interviene l'indicatore di riporto (*carry*), perché se il risultato fosse da intendersi senza segno, sarebbe ancora corretto.

64.5.1.2 Riporto

Si può modificare leggermente il programma proposto, allo scopo di causare un riporto, in modo da vedere cosa accade nei registri:

```
3   .section .data
4   op1: .int 0xFFFFFFFE # 0b11111111111111111111111111111110
5   op2: .int 0x00000002 # 0b00000000000000000000000000000010
```

```
3   section .data
4   op1: dd 0xFFFFFFFE ; 11111111111111111111111111111110b
5   op2: dd 0x00000002 ; 00000000000000000000000000000001b
```

In questo caso, la somma dei due valori supera proprio di una unità la capienza del registro, che alla fine risulta a zero, ma l'indicatore di riporto (*carry*) segnala l'accaduto:

```
eax            0x0        0
ecx            0x0        0
edx            0x0        0
ebx            0x0        0
esp            0xbfbb44e0         0xbfbb44e0
ebp            0x0        0x0
esi            0x0        0
edi            0x0        0
eip            0x8048080          0x8048080 <bp1>
```

```
eflags          0x257    [ CF PF AF ZF IF ]
cs              0x73     115
ss              0x7b     123
ds              0x7b     123
es              0x7b     123
fs              0x0      0
gs              0x0      0
```

Si può osservare che è assente l'indicatore di traboccamento (*overflow*), perché se la somma fosse avvenuta tra due numeri con segno, il risultato sarebbe corretto.

64.5.1.3 Somma di valori molti grandi

Viene mostrato un altro esempio, dove la somma riguarda valori molto grandi, divisi tra due registri. I due listati sono equivalenti e compatibili, riga per riga:

```
1   # op1 + op2
2   #
3   .section .data
4   op1:    .quad   0x00FFFFFFFFFFFFFF
5   op2:    .quad   0x0000000000000001
6   #
7   .section .text
8   .globl _start
9   #
10  _start:
11      mov op1  , %eax # Accumula metà del primo addendo
12                      # in EAX.
13      mov op1+4, %edx # Accumula il resto del primo
14                      # addendo in EDX.
15      add op2,   %eax # Somma metà del secondo addendo
16                      # in EAX.
17  bp1:
```

```
18      adc op2+4, %edx # Somma il resto del secondo addendo
19                      # in EDX. Il risultato atteso in
20                      # EDX:EAX è: 0x01000000:0x00000000
21  bp2:
22      mov $1,    %eax # Conclude il funzionamento del
23      mov $0,    %ebx # programma restituendo zero in
24      int $0x80       # ogni caso.
```

```
1   ; op1 + op2
2   ;
3   section .data
4   op1:    dd  0xFFFFFFFF, 0x00FFFFFF ; 0x00FFFFFFFFFFFFFF
5   op2:    dd  0x00000001, 0x00000000 ; 0x0000000000000001
6   ;
7   section .text
8   global _start
9   ;
10  _start:
11      mov eax, [op1]   ; Accumula metà del primo addendo
12                       ; in EAX.
13      mov edx, [op1+4] ; Accumula il resto del primo
14                       ; addendo in EDX.
15      add eax, [op2]   ; Somma metà del secondo addendo
16                       ; in EAX.
17  bp1:
18      adc edx, [op2+4] ; Somma il resto del secondo
19                       ; addendo in EDX. Il risultato
20                       ; atteso in EDX:EAX è:
21                       ; 0x01000000:0x00000000
22  bp2:
23      mov eax, 1       ; Conclude il funzionamento
24      mov edx, 0       ; del programma restituendo
25      int 80h          ; zero in ogni caso.
```

In questo programma ci sono delle complicazioni che vanno descritte, cominciando preferibilmente dalla versione per GNU AS (il primo listato). Nelle righe 4 e 5 vengono dichiarati due numeri molto grandi, da 64 bit. Successivamente, nelle righe da 11 a 15, si fa riferimento a questi due numeri, prendendoli a pezzi. Per la precisione, nella riga 11 si copiano i primi 32 bit a partire dall'indirizzo a cui fa riferimento l'etichetta '**`op1`**' (sono solo 32 bit perché l'istruzione copia il valore in un registro di tale dimensione); nella riga 12 si copiano gli altri 32 bit, indicando che dall'indirizzo dell'etichetta '**`op1`**' occorre spostarsi in avanti di quattro byte. Lo stesso ragionamento si fa per il secondo operando.

L'ordine in cui sono prelevati i dati è importante e occorre riflettere su questo fatto. Il registro ***EAX*** viene caricato con la porzione meno significativa del numero, che in memoria si trova nei «primi» 32 bit. Ciò avviene perché si intende che il microprocessore operi ordinando i byte in memoria secondo la modalità *little endian*.

Nel secondo listato, quello per NASM, non essendo possibile indicare un numero completo da 64 bit, si è reso necessario spezzarlo in due. In questo caso, per mantenere la stessa struttura dell'altro listato, i due tronconi sono stati messi in fila, apparentemente in ordine inverso, per riprodurre la stessa sequenza *little endian* complessiva.

Una volta compreso il modo in cui i dati sono prelevati dalla memoria, ci si può soffermare sulle istruzioni di somma: il troncone meno significativo viene sommato con l'istruzione '**`ADD`**', mentre per quello più significativo si usa '**`ADC`**' che aggiunge anche il riporto, se c'è.

Alla fine, una volta compilato il programma, con GDB è possibile

eseguirlo fino al simbolo evidenziato dall'etichetta '**bp1**' per verificare l'effetto della prima addizione, quindi lo si può fare proseguire fino al simbolo '**bp2**', per verificare che la coppia di registri ***EDX:EAX*** contenga il risultato corretto:

```
(gdb) break bp1 [Invio]

(gdb) break bp2 [Invio]

(gdb) run [Invio]

(gdb) info registers [Invio]

eax            0x0        0
ecx            0x0        0
edx            0xffffff   16777215
ebx            0x0        0
esp            0xbfa89bb0         0xbfa89bb0
ebp            0x0        0x0
esi            0x0        0
edi            0x0        0
eip            0x8048085          0x8048085 <bp1>
eflags         0x257      [ CF PF AF ZF IF ]
cs             0x73       115
ss             0x7b       123
ds             0x7b       123
es             0x7b       123
fs             0x0        0
gs             0x0        0
```

La prima somma ha prodotto uno zero nel registro ***EAX***, con riporto.

```
(gdb) continue [Invio]

(gdb) info registers [Invio]
```

```
eax            0x0        0
ecx            0x0        0
edx            0x1000000          16777216
ebx            0x0        0
esp            0xbfa89bb0         0xbfa89bb0
ebp            0x0        0x0
esi            0x0        0
edi            0x0        0
eip            0x804808b          0x804808b <bp2>
eflags         0x216      [ PF AF IF ]
cs             0x73       115
ss             0x7b       123
ds             0x7b       123
es             0x7b       123
fs             0x0        0
gs             0x0        0
```

(gdb) **kill** [*Invio*]

(gdb) **quit** [*Invio*]

64.5.2 Sottrazione

Viene proposto un programma che esegue una sottrazione, emettendo il risultato attraverso il valore di uscita. Il programma viene mostrato sia nella forma adatta a GNU AS, sia in quella conforme a NASM. Le righe dei due listati coincidono.

```
1  # op1 - op2
2  #
3  .section .data
4  op1: .int 0x00000001 # 0b00000000000000000000000000000001
5  op2: .int 0x00000002 # 0b00000000000000000000000000000010
6  #
7  .section .text
```

```
8   .globl _start
9   #
10  _start:
11      mov op1, %edx  # Accumula il minuendo in EDX.
12      sub op2, %edx  # Riduce EDX del sottraendo.
13  bp:
14      mov $1,   %eax # Restituisce il valore contenuto in
15      mov %edx, %ebx # EDX come valore di uscita,
16      int $0x80      # attraverso la chiamata di sistema
17                     # 1 (exit).
```

```
1   ; op1 - op2
2   ;
3   section .data
4   op1: dd 0x00000001 ; 00000000000000000000000000000001b
5   op2: dd 0x00000002 ; 00000000000000000000000000000010b
6   ;
7   section .text
8   global _start
9   ;
10  _start:
11      mov edx, [op1] ; Accumula il minuendo in EDX.
12      sub edx, [op2] ; Riduce EDX del sottraendo.
13  bp1:
14      mov eax, 1     ; Restituisce il valore contenuto in EDX
15      mov ebx, edx   ; come valore di uscita, attraverso la
16      int 80h        ; chiamata di sistema 1 (exit).
```

Rispetto agli esempi che riguardano la somma, qui, nella riga 12 si utilizza l'istruzione '**SUB**'. Come si può comprendere, dal momento che si sottrae una grandezza maggiore di quella contenuta nel minuendo, si ottiene un valore negativo, oppure, se i valori sono da intendersi senza segno, si ottiene un «riporto», come richiesta del prestito di una cifra oltre quella più significativa. Con GDB, in cor-

rispondenza del simbolo '**bp1**' si possono vedere i registri e si può verificare che è scattato il riporto (*carry*) e il segno:

```
…
edx            0xffffffff       -1
…
eflags         0x297    [ CF PF AF SF IF ]
…
```

Se si lascia concludere il programma, il valore di uscita che si ottiene è 255_{10}, pari a FF_{16}, ovvero 377_8, ovvero 11111111_2, dal momento che si possono ottenere solo otto bit.

64.5.2.1 Inversione del segno

Si può vedere cosa accade se, invece di usare l'istruzione '**SUB**', si cambia il segno al sottraendo e lo si somma semplicemente all'altro operando:

```
1   # op1 - op2
2   #
3   .section .data
4   op1:    .int    0x00000001   # 0b00000000000000000000000000000001
5   op2:    .int    0x00000002   # 0b00000000000000000000000000000010
6   #
7   .section .text
8   .globl _start
9   #
10  _start:
11      mov op2, %edx  # Accumula il sottraendo in EDX.
12      neg %edx       # Ne inverte il segno con il complemento a due.
13  bp1:
14      add op1, %edx  # Somma il minuendo a EDX.
15  bp2:
16      mov $1,   %eax # Restituisce il valore contenuto in EDX
17      mov %edx, %ebx # come valore di uscita, attraverso la
18      int $0x80      # chiamata di sistema 1 (exit).
```

```
1   ; op1 - op2
2   ;
3   section .data
4   op1:    dd      0x00000001   ; 00000000000000000000000000000001b
5   op2:    dd      0x00000002   ; 00000000000000000000000000000010b
6   ;
7   section .text
8   global _start
9   ;
10  _start:
11      mov edx, [op2] ; Accumula il sottraendo in EDX.
12      neg edx        ; Ne inverte il segno con il complemento a due.
13  bp1:
14      add edx, [op1] ; Somma il minuendo a EDX.
15  bp2:
16      mov eax, 1     ; Restituisce il valore contenuto in EDX
17      mov ebx, edx   ; come valore di uscita, attraverso la
18      int 80h        ; chiamata di sistema 1 (exit).
```

Rispetto a quanto fatto nel caso precedente, qui, nella riga 11 viene sommato il valore del sottraendo nel registro ***EDX***, di cui viene invertito il segno nella riga 12. Successivamente, nella riga 14 viene sommato il valore del minuendo. Il risultato è lo stesso, ma gli indicatori si comportano in modo differente durante il procedimento. In corrispondenza del simbolo '**`bp1`**' si può vedere quanto segue:

```
…
edx             0xfffffffe       -2
…
eflags          0x293    [ CF AF SF IF ]
…
```

Si può osservare che l'inversione del segno ha prodotto un riporto, oltre alla segnalazione del segno, ammesso che questo vada considerato.

In corrispondenza del simbolo '**bp2**' è appena stata eseguita la somma del minuendo. Si può osservare che questa volta non si ottiene alcun riporto:

```
…
edx             0xffffffff       -1
…
eflags          0x293    [ PF SF IF ]
…
```

64.5.2.2 Sottrazione per fasi successive

Viene proposto un esempio di sottrazione da svolgere in due fasi, perché il valore non è contenibile in un solo registro. Si vuole eseguire: 1000000000000000_{16} – $0FFFFFFFFFFFFFFF_{16}$.

```
1   # op1 + op2
2   #
3   .section .data
4   op1:    .quad   0x1000000000000000
5   op2:    .quad   0x0FFFFFFFFFFFFFFF
6   #
7   .section .text
8   .globl _start
9   #
10  _start:
11      mov op1  , %eax # Accumula metà del primo valore
12                      # in EAX.
13      mov op1+4, %edx # Accumula il resto del primo
14                      # valore in EDX.
15  bp1:
16      sub op2,   %eax # Sottrae metà del secondo valore
17                      # in EAX.
18  bp2:
19      sbb op2+4, %edx # Sottrae il resto del secondo
```

```
20                          # valore in EDX.
21  bp3:
22      mov $1,     %eax # Conclude il funzionamento del
23      mov $0,     %ebx # programma restituendo zero
24      int $0x80        # in ogni caso.
```

```
1   ; op1 + op2
2   ;
3   section .data
4   op1:    dd  0x00000000, 0x10000000 ; 0x1000000000000000
5   op2:    dd  0xFFFFFFFF, 0x0FFFFFFF ; 0x0FFFFFFFFFFFFFFF
6   ;
7   section .text
8   global _start
9   ;
10  _start:
11      mov eax, [op1]   ; Accumula metà del primo valore
12                       ; in EAX.
13      mov edx, [op1+4] ; Accumula il resto del primo
14                       ; valore in EDX.
15  bp1:
16      sub eax, [op2]   ; Sottrae metà del secondo valore
17                       ; in EAX.
18  bp2:
19      sbb edx, [op2+4] ; Sottrae il resto del secondo
20                       ; valore in EDX.
21  bp3:
22      mov eax, 1       ; Conclude il funzionamento del
23      mov edx, 0       ; programma restituendo zero
24      int 0x80         ; in ogni caso.
```

Il valore del minuendo viene copiato, in due pezzi, nei registri ***EDX:EAX***; quindi si sottraggono i 32 bit inferiori del sottraendo al registro ***EAX*** e infine si sottraggono i 32 bit più significativi del

sottraendo dal registro ***EDX***, tenendo conto del riporto precedente.

In corrispondenza del simbolo '**bp1**', il minuendo è stato copiato nei registri ***EDX:EAX***:

```
…
eax             0x0       0
…
edx             0x10000000        268435456
…
eflags          0x292     [ AF SF IF ]
…
```

Al punto di '**bp2**' è stata eseguita la sottrazione dei 32 bit inferiori, causando un riporto, da intendersi come richiesta di un prestito:

```
…
eax             0x1       1
…
edx             0x10000000        268435456
…
eflags          0x213     [ CF AF IF ]
…
```

Al punto di '**bp3**' è stata completata la sottrazione:

```
…
eax             0x1       1
…
edx             0x0       0
…
eflags          0x256     [ PF AF ZF IF ]
…
```

64.5.3 Moltiplicazione senza segno

«

Nella moltiplicazione si distingue il fatto che si consideri il segno o meno. Quando si esegue una moltiplicazione senza segno si usa l'istruzione '**MUL**' con l'indicazione di un solo operando, perché gli altri sono impliciti. Nella moltiplicazione il contenitore del risultato deve essere più capiente di ciò che è stato usato per produrlo. Si distinguono questi casi:

$$AX := AL * src$$

$$DX{:}AX := AX * src$$

$$EDX{:}EAX := EAX * src$$

In pratica, l'origine deve essere di pari dimensioni del moltiplicando, costituito, rispettivamente da: ***AL***, ***AX*** o ***EAX***.

```
1   # op1 * op2
2   #
3   .section .data
4   op1:    .short  0x8008
5   op2:    .short  0x2002
6   #
7   .section .bss
8   .lcomm prodotto, 4
9   #
10  .section .text
11  .globl _start
12  #
13  _start:
```

```
14      mov  op1, %ax          # Accumula il moltiplicando
15                             # in AX.
16      mulw op2               # Moltiplica il secondo valore
17                             # per AX (implicito).
18  bp1:
19      mov %ax, prodotto      # Copia in memoria la prima
20                             # parte del risultato.
21      mov %dx, prodotto+2    # Copia in memoria la seconda
22                             # parte del risultato.
23      mov prodotto, %eax     # Copia il risultato dalla
24                             # memoria a EAX.
25  bp2:
26      mov $1,   %eax         # Restituisce il valore
27                             # contenuto in EAX
28      mov %eax, %ebx         # come valore di uscita,
29                             # attraverso la
30      int $0x80              # chiamata di sistema 1 (exit).
```

```
1   ; op1 * op2
2   ;
3   section .data
4   op1:    dw      0x8008
5   op2:    dw      0x2002
6   ;
7   section .bss
8   prodotto resb 4
9   ;
10  section .text
11  global _start
12  ;
13  _start:
14      mov  ax, [op1]         ; Accumula il moltiplicando
15                             ; in AX.
16      mul word [op2]         ; Moltiplica il secondo
17                             ; valore per AX (implicito).
```

```
18  bp1:
19      mov  [prodotto], ax   ; Copia in memoria la prima
20                            ; parte del risultato.
21      mov  [prodotto+2], dx ; Copia in memoria la
22                            ; seconda parte del
23                            ; risultato.
24      mov  eax, [prodotto]  ; Copia il risultato
25                            ; dalla memoria a EAX.
26  bp2:
27      mov  eax, 1           ; Restituisce il valore
28                            ; contenuto in EAX
29      mov  ebx, eax         ; come valore di uscita,
30                            ; attraverso la
31      int  0x80             ; chiamata di sistema
32                            ; 1 (exit).
```

In questo esempio i valori da moltiplicare sono della dimensione di 16 bit e sono, rispettivamente, 8008_{16} e 2002_{16}. Moltiplicando questi due valori si deve ottenere 10020010_{16}. In pratica si deve eseguire una moltiplicazione del tipo ***DX:AX*** := ***AX*src***.

Rispetto a esempi già visti nelle sezioni precedenti, in questo si dichiara un'area di memoria non inizializzata, nella riga numero 8, per contenere almeno quattro byte (32 bit), con il nome simbolico '**`prodotto`**'. All'interno di questa area di memoria si vuole ricostruire il risultato della moltiplicazione in modo che occupi un gruppo continuo di 32 bit.

Nella riga 15 si esegue la moltiplicazione, utilizzando come operando direttamente la memoria. Tuttavia, per farlo, occorre specificare la dimensione di questo operando, altrimenti verrebbe presa in considerazione un'area più grande del voluto.

In corrispondenza del punto '**bp1**' si può vedere il risultato della moltiplicazione diviso tra ***DX*** e ***AX***:

```
…
eax             0x10     16
…
edx             0x1002   4098
…
eflags          0xa93    [ CF AF SF IF OF ]
…
```

Nelle righe 17 e 18 viene copiato il risultato in memoria, ricomponendolo nell'ordine corretto, osservando che si usa una rappresentazione dei valori numerici in modalità *little endian*, quindi la parte meno significativa viene copiata prima. In corrispondenza del punto '**bp2**' il risultato della moltiplicazione è tutto contenuto nel registro ***EAX***:

```
…
eax             0x10020010       268566544
…
edx             0x1002   4098
…
eflags          0xa93    [ CF AF SF IF OF ]
…
```

64.5.4 Moltiplicazione con segno

La moltiplicazione con segno si ottiene con un'istruzione differente, '**IMUL**', che però va usata con la stessa modalità di quella senza segno. Sarebbe possibile usare più di un operando con questa istruzione, senza bisogno di operandi impliciti, ma in generale non è conveniente perché c'è il rischio di creare confusione sulla dimensione di questi operandi.

```
1  # op1 * op2
2  #
3  .section .data
4  op1:    .short   0x0007
5  op2:    .short  -0x0001
6  #
7  .section .bss
8  .lcomm prodotto, 4
9  #
10 .section .text
11 .globl _start
12 #
13 _start:
14     mov   op1, %ax       # Accumula il moltiplicando
15                          # in AX.
16     imulw op2            # Moltiplica il secondo valore
17                          # per AX (implicito).
18 bp1:
19     mov %ax, prodotto    # Copia in memoria la prima
20                          # parte del risultato.
21     mov %dx, prodotto+2  # Copia in memoria la seconda
22                          # parte del risultato.
23     mov prodotto, %eax   # Copia il risultato dalla
24                          # memoria a EAX.
25 bp2:
26     mov $1,   %eax       # Restituisce il valore
27                          # contenuto in EAX
28     mov %eax, %ebx       # come valore di uscita,
29                          # attraverso la
30     int $0x80            # chiamata di sistema 1 (exit).
```

```
1  ; op1 * op2
2  ;
3  section .data
4  op1:    dw       0x0007
```

```
5   op2:    dw      -0x0001
6   ;
7   section .bss
8   prodotto resb 4
9   ;
10  section .text
11  global _start
12  ;
13  _start:
14      mov       ax, [op1]         ; Accumula il
15                                  ; moltiplicando in AX.
16      imul word [op2]             ; Moltiplica il secondo
17                                  ; valore per AX
18                                  ; (implicito).
19  bp1:
20      mov       [prodotto], ax    ; Copia in memoria la
21                                  ; prima parte del
22                                  ; risultato.
23      mov       [prodotto+2], dx  ; Copia in memoria la
24                                  ; seconda parte del
25                                  ; risultato.
26      mov       eax, [prodotto]   ; Copia il risultato
27                                  ; dalla memoria a EAX.
28  bp2:
29      mov       eax, 1            ; Restituisce il valore
30                                  ; contenuto in EAX
31      mov       ebx, eax          ; come valore di uscita,
32                                  ; attraverso la
33      int       0x80              ; chiamata di sistema
34                                  ; 1 (exit).
```

Rispetto a esempi già visti, questo utilizza una costante numerica negativa (riga 5); in pratica è il compilatore che la trasforma nel complemento a due, in modo automatico. Per il resto, tutto pro-

cede come nell'esempio della moltiplicazione intera, a parte l'uso dell'istruzione '**IMUL**'.

In corrispondenza del punto '**bp1**' si può vedere il risultato della moltiplicazione, distribuito tra ***DX:AX***:

```
…
eax             0xfff9    65529
…
edx             0xffff    65535
…
eflags          0x292     [ AF SF IF ]
…
```

In '**bp2**' il risultato è completo nel registro ***EAX***:

```
…
eax             0xfffffff9        -7
…
eflags          0x292     [ AF SF IF ]
…
```

64.5.5 Divisione

Per la divisione si usa un meccanismo simile a quello della moltiplicazione, ma opposto:

AL := ***AX*** / ***src*** ***AH*** := ***resto***

AX := ***DX:AX*** / ***src*** ***DX*** := ***resto***

EAX := ***EDX:EAX*** / ***src*** ***EDX*** := ***resto***

In questo esempio si parte da valori che occupano 32 bit, azzerando inizialmente ***EDX*** perché il dividendo non è così grande da richiederne l'utilizzo.

```
1   # op1 / op2
2   #
3   .section .data
4   op1:    .int    0x00010001
5   op2:    .int    0x00000002
6   #
7   .section .text
8   .globl _start
9   #
10  _start:
11      mov  op1, %eax # Accumula il dividendo in EAX.
12      mov  $0,  %edx # Azzera EDX.
13      divl op2       # Divide EDX:EAX per il divisore.
14  bp1:
15      mov $1,   %eax # Restituisce il valore contenuto
16      mov %edx, %ebx # in EDX come valore di uscita,
17      int $0x80      # attraverso la chiamata di sistema
18                     # 1 (exit).
```

```
1   ; op1 * op2
2   ;
3   section .data
4   op1:    dd      0x00010001
5   op2:    dd      0x00000002
6   ;
7   section .text
8   global _start
9   ;
10  _start:
11      mov  eax, [op1] ; Accumula il dividendo in EAX.
12      mov  edx, 0     ; Azzera EDX.
```

```
13      div long [op2]  ; Divide EDX:EAX per il divisore.
14  bp1:
15      mov  eax, 1     ; Restituisce il valore contenuto
16      mov  edx, eax   ; in EDX come valore di uscita,
17      int  0x80       ; attraverso la chiamata di sistema
18                      ; 1 (exit).
```

In corrispondenza del punto '**bp1**' si può leggere il risultato della divisione in ***EAX*** e il resto in ***EDX***:

```
…
eax             0x8000   32768
…
edx             0x1      1
…
eflags          0x212    [ AF IF ]
…
```

Per quanto riguarda la divisione con segno, tutto procede nello stesso modo, a parte il fatto che si utilizza l'istruzione '**IDIV**':

```
10  _start:
11      mov   op1, %eax # Accumula il dividendo in EAX.
12      mov   $0,  %edx # Azzera EDX.
13      idivl op2       # Divide EDX:EAX per il divisore.
```

```
10  _start:
11      mov       eax, [op1] ; Accumula il dividendo in EAX.
12      mov       edx, 0     ; Azzera EDX.
13      idiv long [op2]      ; Divide EDX:EAX per il divisore.
```

64.6 Esempi con gli «spostamenti»

Vengono mostrati esempi di programmi estremamente banali, per dimostrare il funzionamento delle istruzioni con cui si eseguono gli spostamenti di bit (scorrimenti e rotazioni), attraverso l'aiuto di GDB. Per la compilazione, se si utilizza GNU AS è bene ricordare di inserire l'opzione '**`--gstabs`**', mentre con NASM è bene aggiungere l'opzione '**`-g`**', in modo da poter gestire più facilmente GDB, disponendo dei riferimenti al sorgente:

`$` **`as --gstabs -o`** ***`nome`*****`.o`** ***`nome`*****`.s`** [*Invio*]

`$` **`nasm -g -f elf -o`** ***`nome`*****`.o`** ***`nome`*****`.s`** [*Invio*]

Tabella 64.7. Scorrimenti e rotazioni.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`SHL` `SHR`	***reg***, 1 ***mem***, 1 ***reg*** ***mem***	Fa scorrere i bit, rispettivamente verso sinistra o verso destra (l'ultima cifra perduta finisce nell'indicatore del riporto). Se appare un solo operando, la rotazione viene eseguita ***CL*** volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	`c p z s o` `# · · · #`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`SAL`	***reg***, 1 ***mem***, 1 ***reg*** ***mem***	Funziona esattamente come '**`SHL`**', ma esiste in quanto è la controparte di '**`SAR`**', riferendosi a uno scorrimento aritmetico.	`c p z s o` `# · · · #`
`SAR`	***reg***, 1 ***mem***, 1 ***reg*** ***mem***	Fa scorrere i bit verso destra (l'ultima cifra perduta finisce nell'indicatore del riporto), mantenendo il segno originale. Se appare un solo operando, la rotazione viene eseguita ***CL*** volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	`c p z s o` `# · · · ·`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`RCL` `RCR`	***reg***, 1 ***mem***, 1 ***reg*** ***mem***	Ruota i bit, rispettivamente verso sinistra o verso destra, utilizzando anche l'indicatore di riporto (*carry*). Se appare un solo operando, la rotazione viene eseguita ***CL*** volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	`c p z s o` `t . . . .` `# . . . #`
`ROL` `ROR`	***reg***, 1 ***mem***, 1 ***reg*** ***mem***	Ruota i bit, rispettivamente verso sinistra o verso destra. Se appare un solo operando, la rotazione viene eseguita ***CL*** volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	`c p z s o` `# . . . #`

64.6.1 Scorrimento logico

Viene proposto un esempio in cui si vede l'uso delle istruzioni '**`SHL`**' e '**`SHR`**'. Si parte da un valore che viene fatto scorrere a sinistra, poi viene ripristinato e quindi viene fatto scorrere a destra.

```
# Scorrimento logico (logic shift)
#
.section .data
op1:    .byte   0b01001100      # 0x4C
#
.section .text
.globl _start
#
_start:
    mov  op1, %al  # Inizializza AL con il valore di
                   # partenza.
    shl  $1,  %al  # Sposta a sinistra le cifre di una
                   # posizione.
bp1:               # AL = 0b10011000 0x98 overflow
    shl  $1,  %al  # Sposta a sinistra le cifre di una
                   # posizione.
bp2:               # AL = 0b00110000 0x30 carry, overflow
    mov  op1, %al  # Inizializza AL con il valore di
                   # partenza.
    shr  $1,  %al  # Sposta a destra le cifre di una
                   # posizione.
bp3:               # AL = 0b00100110 0x26
    shr  $2,  %al  # Sposta a destra le cifre di due
                   # posizioni.
bp4:               # AL = 0b00001001 0x09 carry
    mov $0,   %ebx # Restituisce il valore contenuto in AL
    mov %al,  %bl  # copiandolo nella parte inferiore del
    mov $1,   %eax # registro EBX ed eseguendo la chiamata
    int $0x80      # di sistema 1 (exit).
```

```
; Scorrimento logico (logic shift)
;
section .data
op1:    dd      01001100b       ; 0x4C
;
```

```
section .text
global _start
;
_start:
    mov  al, [op1] ; Inizializza AL con il valore di
                   ; partenza.
    shl  al, 1     ; Sposta a sinistra le cifre di una
                   ; posizione.
bp1:               ; AL = 10011000b 0x98 overflow
    shl  al,  1    ; Sposta a sinistra le cifre di una
                   ; posizione.
bp2:               ; AL = 00110000b 0x30 carry, overflow
    mov  al, [op1] ; Inizializza AL con il valore di
                   ; partenza.
    shr  al, 1     ; Sposta a destra le cifre di una
                   ; posizione.
bp3:               ; AL = 00100110b 0x26
    shr  al, 2     ; Sposta a destra le cifre di due
                   ; posizioni.
bp4:               ; AL = 00001001b 0x09 carry
    mov ebx, 0     ; Restituisce il valore contenuto in AL
    mov bl, al     ; copiandolo nella parte inferiore del
    mov eax, 1     ; registro EBX ed eseguendo la chiamata
    int 0x80       ; di sistema 1 (exit).
```

Viene mostrato lo stato del registro ***EAX*** e degli indicatori, nei vari punti di sospensione previsti:

bp1

```
eax             0x98     152
eflags          0xa92    [ AF SF IF OF ]
```

Bisogna tenere presente che l'ultima istruzione eseguita riguarda solo la porzione ***AL*** del registro ***EAX***; pertanto, dal momento

che il bit più significativo è a uno, si attiva l'indicatore del segno e anche quello del traboccamento, dato che, se lo scorrimento riguardasse un valore con segno, il valore ottenuto non sarebbe più valido (perché a questo punto si sarebbe trasformato in un numero negativo).

bp2

```
eax             0x30     48
eflags          0xa17    [ CF PF AF IF OF ]
```

Sempre tenendo conto che le istruzioni eseguite riguardano solo ***AL***, si può osservare che la perdita della cifra più significativa a uno si traduce nell'attivazione del riporto; inoltre, se si trattasse di un valore con segno, anche questa volta si sarebbe verificata un'inversione (da negativo a positivo) e per questo si attiva ancora l'indicatore di traboccamento.

bp3

```
eax             0x26     38
eflags          0x212    [ AF IF ]
```

Questo scorrimento a destra fa perdere solo delle cifre a zero, pertanto il segno non cambia e non c'è alcun riporto (resto).

bp4

```
eax             0x9      9
eflags          0x217    [ CF PF AF IF ]
```

Questo ulteriore scorrimento, di due posizioni, comporta la fuoriuscita di una cifra a uno che implica l'attivazione dell'indicatore del riporto (resto).

64.6.2 Scorrimento aritmetico

Viene proposto un esempio analogo a quello della sezione precedente, in cui si vede l'uso delle istruzioni '**SAL**' e '**SAR**'. Si parte da un valore negativo che viene fatto scorrere a sinistra, poi viene ripristinato e quindi viene fatto scorrere a destra.

```
# Scorrimento aritmetico
#
.section .data
op1:    .byte   0b11001100      # 0xCC
#
.section .text
.globl _start
#
_start:
    mov  op1, %al  # Inizializza AL con il valore di
                   # partenza.
    sal  $1,  %al  # Sposta a sinistra le cifre di una
                   # posizione.
bp1:               # AL = 0b10011000 0x98 carry
    sal  $1,  %al  # Sposta a sinistra le cifre di una
                   # posizione.
bp2:               # AL = 0b00110000 0x30 carry, overflow
    mov  op1, %al  # Inizializza AL con il valore di
                   # partenza.
    sar  $1,  %al  # Sposta a destra le cifre di una
                   # posizione.
bp3:               # AL = 0b11100110 0xE6
    sar  $2,  %al  # Sposta a destra le cifre di due
                   # posizioni.
bp4:               # AL = 0b11111001 0xF9 carry
    mov $0,   %ebx # Restituisce il valore contenuto in AL
    mov %al,  %bl  # copiandolo nella parte inferiore del
    mov $1,   %eax # registro EBX ed eseguendo la chiamata
```

```
    int $0x80       # di sistema 1 (exit).
```

```
; Scorrimento aritmetico
;
section .data
op1:    dd      11001100b       ; 0xCC
;
section .text
global _start
;
_start:
    mov  al, [op1] ; Inizializza AL con il valore di
                   ; partenza.
    sal  al, 1     ; Sposta a sinistra le cifre di una
                   ; posizione.
bp1:               ; AL = 10011000b 0x98 carry
    sal  al,  1    ; Sposta a sinistra le cifre di una
                   ; posizione.
bp2:               ; AL = 00110000b 0x30 carry, overflow
    mov  al, [op1] ; Inizializza AL con il valore di
                   ; partenza.
    sar  al, 1     ; Sposta a destra le cifre di una
                   ; posizione.
bp3:               ; AL = 11100110b 0xE6
    sar  al, 2     ; Sposta a destra le cifre di due
                   ; posizioni.
bp4:               ; AL = 11111001b 0xF9 carry
    mov ebx, 0     ; Restituisce il valore contenuto in AL
    mov bl, al     ; copiandolo nella parte inferiore del
    mov eax, 1     ; registro EBX ed eseguendo la chiamata
    int 0x80       ; di sistema 1 (exit).
```

Viene mostrato lo stato del registro ***EAX*** e degli indicatori, nei vari punti di sospensione previsti:

bp1

```
eax             0x98     152
eflags          0x293    [ CF AF SF IF ]
```

Bisogna tenere presente che l'ultima istruzione eseguita riguarda solo la porzione ***AL*** del registro ***EAX***; pertanto, dal momento che il bit più significativo è a uno, è attivo l'indicatore del segno e anche quello del riporto, avendo perduto una cifra a uno. Il numero non ha cambiato di segno e quindi l'indicatore di traboccamento non è attivo.

bp2

```
eax             0x30     48
eflags          0xa17    [ CF PF AF IF OF ]
```

Sempre tenendo conto che le istruzioni eseguite riguardano solo ***AL***, si può osservare che anche questa volta è stata persa una cifra più significativa a uno, essendo attivo l'indicatore del riporto, ma la cosa più importante è che adesso il valore ha cambiato di segno e così si vede attivo anche l'indicatore di traboccamento.

bp3

```
eax             0xe6     230
eflags          0x292    [ AF SF IF ]
```

Questo scorrimento a destra fa perdere solo delle cifre a zero, pertanto non c'è alcun riporto (resto).

bp4

```
eax             0xf9     249
eflags          0x297    [ CF PF AF SF IF ]
```

Questo ulteriore scorrimento, di due posizioni, comporta la fuoriuscita di una cifra a uno che implica l'attivazione dell'indicatore del riporto (resto).

64.6.3 Rotazione

«

Viene proposto un esempio analogo a quello delle sezioni precedenti, in cui si vede l'uso delle istruzioni '**ROL**' e '**ROR**'. Si parte da un valore che viene fatto ruotare a sinistra, poi viene ripristinato e quindi viene fatto ruotare a destra.

```
# Rotazione
#
.section .data
op1:    .byte   0b11001100      # 0xCC
#
.section .text
.globl _start
#
_start:
    mov  op1, %al  # Inizializza AL con il valore di
                   # partenza.
    rol  $1,  %al  # Ruota a sinistra le cifre di una
                   # posizione.
bp1:               # AL = 0b10011001 0x99 carry
    rol  $1,  %al  # Ruota a sinistra le cifre di una
                   # posizione.
bp2:               # AL = 0b00110011 0x33 carry, overflow
    mov  op1, %al  # Inizializza AL con il valore di
                   # partenza.
    ror  $1,  %al  # Ruota a destra le cifre di una
                   # posizione.
bp3:               # AL = 0b01100110 0x66 overflow
    ror  $2,  %al  # Ruota a destra le cifre di due
```

```
                    # posizioni.
bp4:                # AL = 0b10011001 0x99 carry, overflow
    mov $0,   %ebx # Restituisce il valore contenuto in AL
    mov %al,  %bl  # copiandolo nella parte inferiore del
    mov $1,   %eax # registro EBX ed eseguendo la chiamata
    int $0x80      # di sistema 1 (exit).
```

```
; Rotazione
;
section .data
op1:    dd      11001100b       ; 0xCC
;
section .text
global _start
;
_start:
    mov  al, [op1] ; Inizializza AL con il valore di
                   ; partenza.
    rol  al, 1     ; Ruota a sinistra le cifre di una
                   ; posizione.
bp1:               ; AL = 10011001b 0x99 carry
    rol  al,  1    ; Ruota a sinistra le cifre di una
                   ; posizione.
bp2:               ; AL = 00110011b 0x33 carry, overflow
    mov  al, [op1] ; Inizializza AL con il valore di
                   ; partenza.
    ror  al, 1     ; Ruota a destra le cifre di una
                   ; posizione.
bp3:               ; AL = 01100110b 0x66 overflow
    ror  al, 2     ; Ruota a destra le cifre di due
                   ; posizioni.
bp4:               ; AL = 10011001b 0x99 carry, overflow
    mov ebx, 0     ; Restituisce il valore contenuto in AL
    mov bl, al     ; copiandolo nella parte inferiore del
    mov eax, 1     ; registro EBX ed eseguendo la chiamata
```

```
    int 0x80        ; di sistema 1 (exit).
```

Viene mostrato lo stato del registro ***EAX*** e degli indicatori, nei vari punti di sospensione previsti:

bp1

```
eax             0x99     153
eflags          0x293    [ CF AF SF IF ]
```

Si deve tenere conto che l'ultima istruzione eseguita riguarda solo la porzione ***AL*** del registro ***EAX***. La rotazione ha fatto uscire a sinistra una cifra a uno che rientra nella parte destra. La cifra spostata la si ritrova nell'indicatore di riporto.

bp2

```
eax             0x33     51
eflags          0xa93    [ CF AF SF IF OF ]
```

Questa ulteriore rotazione a sinistra fa uscire un'altra cifra a uno, che viene segnalata nel riporto, ma il bit più significativo passa a zero e quindi l'indicatore di traboccamento viene attivato.

bp3

```
eax             0x66     102
eflags          0xa92    [ AF SF IF OF ]
```

Questa rotazione a destra fa perdere solo una cifra a zero, pertanto non c'è alcun riporto (resto), ma dal momento che rientra a sinistra, l'indicatore di traboccamento manifesta l'ipotesi di cambiamento del segno.

bp4

```
eax             0x99     153
eflags          0xa93    [ CF AF SF IF OF ]
```

Questa ulteriore rotazione, di due posizioni, comporta la fuoriuscita di una cifra a uno che implica l'attivazione dell'indicatore del riporto (resto). Dal momento che la cifra rientra a sinistra, l'indicatore di traboccamento segnala l'ipotesi di cambiamento del segno.

64.6.4 Rotazione con riporto

Viene proposto un esempio analogo a quello della sezione precedente, in cui si vede l'uso delle istruzioni '**RCL**' e '**RCR**'. Si parte da un valore che viene fatto ruotare a sinistra, poi viene ripristinato e quindi viene fatto ruotare a destra. Qui viene usata anche l'istruzione '**CLC**' per azzerare il riporto.

```
# Rotazione con riporto
#
.section .data
op1:    .byte   0b11001101      # 0xCD
#
.section .text
.globl _start
#
_start:
    clc             # Azzera il riporto.
    mov  op1, %al   # Inizializza AL con il valore di
                    # partenza.
    rcl  $1,  %al   # Ruota a sinistra le cifre di una
                    # posizione.
bp1:                # AL = 0b10011010 0x9A carry
    rcl  $1,  %al   # Ruota a sinistra le cifre di una
```

```
                    # posizione.
bp2:                # AL = 0b00110101 0x35 carry, overflow
    clc             # Azzera il riporto.
    mov  op1, %al   # Inizializza AL con il valore di
                    # partenza.
    rcr  $1,  %al   # Ruota a destra le cifre di una
                    # posizione.
bp3:                # AL = 0b01100110 0x66 carry, overflow
    rcr  $2,  %al   # Ruota a destra le cifre di due
                    # posizioni.
bp4:                # AL = 0b01011001 0x59 carry, overflow
    mov $0,   %ebx # Restituisce il valore contenuto in AL
    mov %al,  %bl  # copiandolo nella parte inferiore del
    mov $1,   %eax # registro EBX ed eseguendo la chiamata
    int $0x80      # di sistema 1 (exit).
```

```
; Rotazione con riporto
;
section .data
op1:    dd      11001101b       ; 0xCD
;
section .text
global _start
;
_start:
    clc             ; Azzera il riporto.
    mov  al, [op1] ; Inizializza AL con il valore di
                    ; partenza.
    rcl  al, 1      ; Ruota a sinistra le cifre di una
                    ; posizione.
bp1:                ; AL = 10011010b 0x9A carry
    rcl  al,  1     ; Ruota a sinistra le cifre di una
                    ; posizione.
bp2:                ; AL = 00110101b 0x35 carry, overflow
    clc             ; Azzera il riporto.
```

```
    mov  al, [op1] ; Inizializza AL con il valore di
                   ; partenza.
    rcr  al, 1     ; Ruota a destra le cifre di una
                   ; posizione.
bp3:               ; AL = 01100110b 0x66 carry, overflow
    rcr  al, 2     ; Ruota a destra le cifre di due
                   ; posizioni.
bp4:               ; AL = 01011001b 0x59 carry, overflow
    mov ebx, 0     ; Restituisce il valore contenuto in AL
    mov bl, al     ; copiandolo nella parte inferiore del
    mov eax, 1     ; registro EBX ed eseguendo la chiamata
    int 0x80       ; di sistema 1 (exit).
```

Viene mostrato lo stato del registro ***EAX*** e degli indicatori, nei vari punti di sospensione previsti:

bp1

```
eax             0x9a     154
eflags          0x293    [ CF AF SF IF ]
```

Si deve tenere conto che l'ultima istruzione eseguita riguarda solo la porzione ***AL*** del registro ***EAX***. La rotazione ha fatto uscire a sinistra una cifra a uno che passa nel riporto, mentre a destra entra il valore precedente del riporto (zero).

bp2

```
eax             0x35     53
eflags          0xa93    [ CF AF SF IF OF ]
```

Questa ulteriore rotazione a sinistra fa uscire un'altra cifra a uno che passa nel, mentre entra a destra la cifra a uno del riporto precedente. Dopo la rotazione il bit più significativo passa a zero e quindi l'indicatore di traboccamento viene attivato.

bp3

```
eax             0x66     102
eflags          0xa93    [ CF AF SF IF OF ]
```

Questa rotazione a destra fa perdere una cifra a uno che si trasferisce del riporto; inoltre, dato che il riporto precedente era zero, a sinistra si inserisce una cifra a zero, la quale fa scattare l'indicatore di traboccamento (nell'ipotesi di un valore con segno).

bp4

```
eax             0x59     89
eflags          0xa93    [ CF AF SF IF OF ]
```

Questa ulteriore rotazione, di due posizioni, comporta la fuoriuscita di una cifra a zero e poi a uno che implica l'attivazione dell'indicatore del riporto (resto). In precedenza c'era stato un riporto che attualmente risulta inserito subito dopo la cifra più significativa. Dal momento l'ultimo scorrimento (dei due eseguiti qui) fa cambiare la cifra più significativa, si attiva l'indicatore di traboccamento.

64.7 Esempi con i confronti

«

Viene mostrato un esempio di programma, con l'unico scopo di dimostrare il funzionamento dell'istruzione di confronto, attraverso l'aiuto di GDB. Per la compilazione, se si utilizza GNU AS è bene ricordare di inserire l'opzione '**`--gstabs`**', mentre con NASM è bene aggiungere l'opzione '**`-g`**', in modo da poter gestire più facilmente GDB, disponendo dei riferimenti al sorgente:

```
$ as --gstabs -o nome.o nome.s [Invio]
```

$ **nasm -g -f elf -o *nome*.o *nome*.s** [*Invio*]

```
# Confronto
#
.section .text
.globl _start
#
_start:
    mov $0b01000000, %al        # A =  64       A =   64
    mov $0b10000000, %bl        # B = 128       B = -128
    cmp %bl, %al                # carry, segno, overflow
bp1:
    mov $0b01000000, %al        # A =  64       A =   64
    mov $0b11000000, %bl        # B = 192       B =  -64
    cmp %bl, %al                # carry, segno, overflow
bp2:
    mov $0b01000000, %al        # A =  64       A =   64
    mov $0b00000000, %bl        # B =   0       B =    0
    cmp %bl, %al                #
bp3:
    mov $0b01000000, %al        # A =  64       A =   64
    mov $0b01000000, %bl        # B =  64       B =   64
    cmp %bl, %al                # zero
bp4:
    mov $0b00000000, %al        # A =   0       A =    0
    mov $0b10000000, %bl        # B = 128       B = -128
    cmp %bl, %al                # carry, segno, overflow
bp5:
    mov $0b00000000, %al        # A =   0       A =    0
    mov $0b11000000, %bl        # B = 192       B =  -64
    cmp %bl, %al                # carry
bp6:
    mov $0b00000000, %al        # A =   0       A =    0
    mov $0b00000000, %bl        # B =   0       B =    0
    cmp %bl, %al                # zero
bp7:
```

```
    mov $0b00000000, %al        # A =   0       A =    0
    mov $0b01000000, %bl        # B =  64       B =   64
    cmp %bl, %al                # carry, segno
bp8:
    mov $0b11000000, %al        # A = 192       A =  -64
    mov $0b10000000, %bl        # B = 128       B = -128
    cmp %bl, %al                #
bp9:
    mov $0b11000000, %al        # A = 192       A =  -64
    mov $0b11000000, %bl        # B = 192       B =  -64
    cmp %bl, %al                # zero
bp10:
    mov $0b11000000, %al        # A = 192       A =  -64
    mov $0b00000000, %bl        # B =   0       B =    0
    cmp %bl, %al                # segno
bp11:
    mov $0b11000000, %al        # A = 192       A =  -64
    mov $0b01000000, %bl        # B =  64       B =   64
    cmp %bl, %al                # segno
bp12:
    mov $0b10000000, %al        # A = 128       A = -128
    mov $0b10000000, %bl        # B = 128       B = -128
    cmp %bl, %al                # zero
bp13:
    mov $0b10000000, %al        # A = 128       A = -128
    mov $0b11000000, %bl        # B = 192       B =  -64
    cmp %bl, %al                # carry, segno
bp14:
    mov $0b10000000, %al        # A = 128       A = -128
    mov $0b00000000, %bl        # B =   0       B =    0
    cmp %bl, %al                # segno
bp15:
    mov $0b10000000, %al        # A = 128       A = -128
    mov $0b01000000, %bl        # B =  64       B =   64
```

```
    cmp %bl, %al                # overflow
bp16:
    mov $0,   %ebx      # Conclude il funzionamento con la
    mov $1,   %eax      # chiamata di sistema 1 (exit).
    int $0x80           #
```

```
; Confronto
;
section .text
global _start
;
_start:
    mov al, 01000000b           ; A =  64       A =   64
    mov bl, 10000000b           ; B = 128       B = -128
    cmp al, bl                  ; carry, segno, overflow
bp1:
    mov al, 01000000b           ; A =  64       A =   64
    mov bl, 11000000b           ; B = 192       B =  -64
    cmp al, bl                  ; carry, segno, overflow
bp2:
    mov al, 01000000b           ; A =  64       A =   64
    mov bl, 00000000b           ; B =   0       B =    0
    cmp al, bl                  ;
bp3:
    mov al, 01000000b           ; A =  64       A =   64
    mov bl, 01000000b           ; B =  64       B =   64
    cmp al, bl                  ; zero
bp4:
    mov al, 00000000b           ; A =   0       A =    0
    mov bl, 10000000b           ; B = 128       B = -128
    cmp al, bl                  ; carry, segno, overflow
bp5:
    mov al, 00000000b           ; A =   0       A =    0
    mov bl, 11000000b           ; B = 192       B =  -64
    cmp al, bl                  ; carry
```

```
bp6:
    mov al, 00000000b           ; A =   0       A =    0
    mov bl, 00000000b           ; B =   0       B =    0
    cmp al, bl                  ; zero
bp7:
    mov al, 00000000b           ; A =   0       A =    0
    mov bl, 01000000b           ; B =  64       B =   64
    cmp al, bl                  ; carry, segno
bp8:
    mov al, 11000000b           ; A = 192       A =  -64
    mov bl, 10000000b           ; B = 128       B = -128
    cmp al, bl                  ;
bp9:
    mov al, 11000000b           ; A = 192       A =  -64
    mov bl, 11000000b           ; B = 192       B =  -64
    cmp al, bl                  ; zero
bp10:
    mov al, 11000000b           ; A = 192       A =  -64
    mov bl, 00000000b           ; B =   0       B =    0
    cmp al, bl                  ; segno
bp11:
    mov al, 11000000b           ; A = 192       A =  -64
    mov bl, 01000000b           ; B =  64       B =   64
    cmp al, bl                  ; segno
bp12:
    mov al, 10000000b           ; A = 128       A = -128
    mov bl, 10000000b           ; B = 128       B = -128
    cmp al, bl                  ; zero
bp13:
    mov al, 10000000b           ; A = 128       A = -128
    mov bl, 11000000b           ; B = 192       B =  -64
    cmp al, bl                  ; carry, segno
bp14:
    mov al, 10000000b           ; A = 128       A = -128
```

```
    mov bl, 00000000b           ; B =   0       B =    0
    cmp al, bl                  ; segno
bp15:
    mov al, 10000000b           ; A = 128       A = -128
    mov bl, 01000000b           ; B =  64       B =   64
    cmp al, bl                  ; overflow
bp16:
    mov ebx, 0  ; Conclude il funzionamento con la
    mov eax, 1  ; chiamata di sistema 1 (exit).
    int 0x80    ;
```

Tra i commenti si possono osservare i valori confrontati, interpretandoli sia con segno, sia senza segno, assieme all'effetto atteso sugli indicatori. Viene mostrato lo stato degli indicatori nei vari punti di sospensione previsti, con l'ausilio di GDB:

bp1

```
eax             0x40    64
ebx             0x80    128
eflags          0xa87   [ CF PF SF IF OF ]
```

bp2

```
eax             0x40    64
ebx             0xc0    192
eflags          0xa83   [ CF SF IF OF ]
```

bp3

```
eax             0x40    64
ebx             0x0     0
eflags          0x202   [ IF ]
```

bp4

```
eax             0x40      64
ebx             0x40      64
eflags          0x246     [ PF ZF IF ]
```

bp5

```
eax             0x0       0
ebx             0x80      128
eflags          0xa83     [ CF SF IF OF ]
```

bp6

```
eax             0x0       0
ebx             0xc0      192
eflags          0x203     [ CF IF ]
```

bp7

```
eax             0x0       0
ebx             0x0       0
eflags          0x246     [ PF ZF IF ]
```

bp8

```
eax             0x0       0
ebx             0x40      64
eflags          0x287     [ CF PF SF IF ]
```

bp9

```
eax             0xc0      192
ebx             0x80      128
eflags          0x202     [ IF ]
```

bp10

```
eax             0xc0      192
ebx             0xc0      192
eflags          0x246     [ PF ZF IF ]
```

bp11

```
eax             0xc0     192
ebx             0x0      0
eflags          0x286    [ PF SF IF ]
```

bp12

```
eax             0xc0     192
ebx             0x40     64
eflags          0x282    [ SF IF ]
```

bp13

```
eax             0x80     128
ebx             0x80     128
eflags          0x246    [ PF ZF IF ]
```

bp14

```
eax             0x80     128
ebx             0xc0     192
eflags          0x287    [ CF PF SF IF ]
```

bp15

```
eax             0x80     128
ebx             0x0      0
eflags          0x282    [ SF IF ]
```

bp16

```
eax             0x80     128
ebx             0x40     64
eflags          0xa02    [ IF OF ]
```

64.8 Le istruzioni di salto

«

Con il linguaggio macchina, le strutture di controllo si realizzano solo attraverso le istruzioni di salto. Una di queste istruzioni è incondizionata, mentre le altre sono sottoposte al verificarsi di una condizione. A loro volta, le istruzioni di salto condizionato si dividono in due gruppi: uno riferito al controllo dello stato di un certo indicatore, l'altro riferito virtualmente a un confronto di valori.

64.8.1 Portata del salto

«

In generale, le istruzioni di salto hanno un solo operando, costituito dal riferimento all'indirizzo di memoria da raggiungere. Il compilatore traduce il riferimento all'indirizzo di memoria in un «dislocamento», ovvero nella quantità di byte da percorrere, in avanti o indietro. A questo proposito, ciò che rappresenta il riferimento alla memoria può avere dimensioni diverse e questo significa che l'istruzione può richiedere di precisare la dimensione del numero che rappresenta tale dislocamento. In modo predefinito, la dimensione del numero usato per indicare il dislocamento è quella di un registro comune.

Si osservi che la possibilità di dichiarare esplicitamente l'entità del dislocamento dipende dal compilatore; d'altro canto, sarebbe compito del compilatore determinarlo automaticamente.

64.8.2 Salto incondizionato

Il salto incondizionato si ottiene con l'istruzione '**JMP**':

```
JMP imm
```

Con un linguaggio assemblatore, il valore immediato richiesto come operando si ottiene con un simbolo, ovvero indicando il nome di un'etichetta già dichiarata altrove nel sorgente:

```
# Salto incondizionato
#
.section .text
.globl _start
#
_start:
    mov $1, %ebx
bp1:
    jmp bp3
bp2:
    mov $2, %ebx      # Questa istruzione non viene eseguita.
bp3:
    mov $1, %eax      # Conclude il funzionamento con la
    int $0x80         # chiamata di sistema 1 (exit).
```

```
; Salto incondizionato
;
section .text
global _start
;
_start:
    mov ebx, 1
bp1:
    jmp bp3
bp2:
```

```
    mov ebx, 2          ; Questa istruzione non viene eseguita.
bp3:
    mov eax, 1          ; Conclude il funzionamento con la
    int 0x80            ; chiamata di sistema 1 (exit).
```

In questo esempio si vede che l'istruzione contenuta tra i punti '**bp2**' e '**bp3**' non viene mai eseguita.

64.8.3 Salto condizionato dallo stato di un indicatore

«

Un gruppo di istruzioni di salto condizionato dipende dallo stato di un certo indicatore. Queste istruzioni hanno la forma seguente, dove la lettera ***x*** identifica l'indicatore da controllare:

```
Jx  imm
```

```
JNx  imm
```

Per esempio, l'istruzione '**JZ**' salta se l'indicatore zero è attivo, mentre '**JNZ**' salta se l'indicatore zero non è attivo.

Tabella 64.135. Salti condizionati dallo stato di un indicatore particolare.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione
JC JNC	***imm***	Salta se l'indicatore del riporto (*carry*), rispettivamente, è attivo, oppure non è attivo.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione
`JO` `JNO`	***imm***	Salta se l'indicatore di traboccamento (*overflow*), rispettivamente, è attivo, oppure non è attivo.
`JS` `JNS`	***imm***	Salta se l'indicatore di segno (*sign*), rispettivamente, è attivo, oppure non è attivo.
`JZ` `JNZ`	***imm***	Salta se l'indicatore di zero, rispettivamente, è attivo, oppure non è attivo.
`JP` `JNP`	***imm***	Salta se l'indicatore di parità, rispettivamente, è attivo, oppure non è attivo.

64.8.4 Salto condizionato da un confronto

Il gruppo più importante di istruzioni di salto condizionato dipende da un confronto, che di solito si realizza con l'istruzione '**`CMP`**'. In pratica, l'istruzione '**`CMP`**' simula una sottrazione, aggiornando gli indicatori come se si trattasse di una sottrazione vera e propria; successivamente, l'istruzione di salto condizionato verifica gli indicatori e si comporta di conseguenza.

Da quanto descritto si deve comprendere che, anche se le istruzioni di questo tipo richiamano l'idea del confronto tra due valori, in pratica dipendono da indicatori che possono essere stati modificati da istruzioni di tipo differente.

Dal momento che, ai fini del confronto tra due valori, la valutazione degli indicatori va fatta diversamente a seconda che si tratti di interi senza segno o con segno, queste istruzioni sono suddivise in sottogruppi.

Tabella 64.136. Salti condizionati dall'esito di un confronto con valori interi senza segno.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione
`JA` `JNBE`	***imm***	Dopo un confronto di valori senza segno, salta se la destinazione era maggiore dell'origine. CMP ***dst***, ***org*** IF ***dst > org*** THEN go to ***imm***
`JAE` `JNB`	***imm***	Dopo un confronto di valori senza segno, salta se la destinazione era maggiore o uguale all'origine. CMP ***dst***, ***org*** IF ***dst >= org*** THEN go to ***imm***
`JB` `JNAE`	***imm***	Dopo un confronto di valori senza segno, salta se la destinazione era minore dell'origine. CMP ***dst***, ***org*** IF ***dst < org*** THEN go to ***imm***

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione
`JBE` `JNA`	***imm***	Dopo un confronto di valori senza segno, salta se la destinazione era minore o uguale all'origine. CMP ***dst***, ***org*** IF ***dst <= org*** THEN go to ***imm***

Tabella 64.137. Salti condizionati dall'esito di un confronto con valori interi, indipendentemente dal segno.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione
`JE`	***imm***	Dopo un confronto, indipendentemente dal segno, salta se la destinazione era uguale all'origine. CMP ***dst***, ***org*** IF ***dst == org*** THEN go to ***imm***
`JNE`	***imm***	Dopo un confronto, indipendentemente dal segno, salta se la destinazione era diversa dall'origine. CMP ***dst***, ***org*** IF ***dst =/= org*** THEN go to ***imm***

Tabella 64.138. Salti condizionati dall'esito di un confronto con valori interi con segno.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione
`JG` `JNLE`	***imm***	Dopo un confronto con segno, salta se la destinazione era maggiore dell'origine. CMP ***dst***, ***org*** IF ***dst > org*** THEN go to ***imm***
`JGE` `JNL`	***imm***	Dopo un confronto con segno, salta se la destinazione era maggiore o uguale all'origine. CMP ***dst***, ***org*** IF ***dst >= org*** THEN go to ***imm***
`JL` `JNGE`	***imm***	Dopo un confronto con segno, salta se la destinazione era minore dell'origine. CMP ***dst***, ***org*** IF ***dst < org*** THEN go to ***imm***
`JLE` `JNG`	***imm***	Dopo un confronto con segno, salta se la destinazione era minore o uguale all'origine. CMP ***dst***, ***org*** IF ***dst <= org*** THEN go to ***imm***

64.8.5 Cicli

«

È possibile realizzare dei cicli enumerativi attraverso delle istruzioni simili a quelle di salto condizionato, dove in pratica, dopo il blocco di istruzioni da reiterare, viene verificata la condizione e quindi, eventualmente, si ripete il ciclo. Si distinguono tre casi:

```
LOOP imm
```

```
LOOPE imm | LOOPZ imm
```

```
LOOPNE imm | LOOPNZ imm
```

In tutte le situazioni, il valore immediato che viene fornito come operando si riferisce al dislocamento dell'indirizzo di memoria da raggiungere (cosa che viene tradotta dal compilatore, sostituendo il simbolo con il numero appropriato). Il dislocamento consentito è breve ($\pm$128 byte), quindi non si possono realizzare cicli contenenti tante istruzioni.

In tutte le situazioni, l'istruzione decrementa prima il registro ***ECX*** (oppure solo ***CX***, se viene dichiarato l'uso di una dimensione «breve», ovvero a soli 16 bit[8]) senza alterare gli indicatori, quindi verifica se tale registro è diverso da zero. Nel caso dell'istruzione '**LOOP**', il fatto che il registro, dopo il decremento di una unità, contenga ancora un valore diverso da zero, è sufficiente per far scattare la ripetizione del ciclo; nel caso di '**LOOPE**' o di '**LOOPZ**', è necessario anche che l'indicatore zero sia attivo; per converso, con '**LOOPNE**' o '**LOOPNZ**', è necessario anche che l'indicatore zero non sia attivo.

64.9 Esempi di programmi con strutture di controllo

Vengono mostrati esempi di programmi estremamente banali, per dimostrare il funzionamento delle strutture di controllo, basate sostanzialmente su istruzioni di salto condizionato, attraverso l'aiuto

di GDB. Per la compilazione, se si utilizza GNU AS è bene ricordare di inserire l'opzione '**--gstabs**', mentre con NASM è bene aggiungere l'opzione '**-g**', in modo da poter gestire più facilmente GDB, disponendo dei riferimenti al sorgente:

```
$ as --gstabs -o nome.o nome.s [Invio]

$ nasm -g -f elf -o nome.o nome.s [Invio]
```

64.9.1 Somma attraverso l'incremento unitario

«

Viene mostrato un programma che esegue la somma di due valori interi senza segno, incrementando progressivamente il primo addendo, di una unità, corrispondentemente alla riduzione di una unità del secondo. Quando il secondo addendo è stato ridotto a zero, il primo contiene il risultato della somma. Il ciclo con cui si incrementa il primo addendo è controllato dall'istruzione '**LOOP**':

```
# op1 + op2
#
.section .data
op1:    .int    0x00000007      # Intero senza segno.
op2:    .int    0x00000002      # Intero senza segno.
#
.section .text
.globl _start
#
_start:
    mov op1, %eax   # Primo addendo.
    mov op2, %ecx   # Secondo addendo.
bp1:
    cmp $0, %ecx    # Se il secondo addendo è zero, non
    je end_do_somma # esegue il ciclo di incrementi.
do_somma:
```

```
    inc %eax          # Aggiunge una unità a EAX.
    loop do_somma     # Ripete il ciclo se nel frattempo
                      # ECX non si è azzerato.
end_do_somma:
    mov %eax, %ebx    # Restituisce il valore
    mov $1,   %eax    # ottenuto dalla somma.
    int $0x80         #
```

```
; op1 + op2
;
section .data
op1:    dd      0x00000007      ; Intero senza segno.
op2:    dd      0x00000002      ; Intero senza segno.
;
section .text
global _start
;
_start:
    mov eax, [op1]  ; Primo addendo.
    mov ecx, [op2]  ; Secondo addendo.
bp1:
    cmp ecx, 0      ; Se il secondo addendo è zero, non
    je end_do_somma ; esegue il ciclo di incrementi.
do_somma:
    inc eax         ; Aggiunge una unità a EAX.
    loop do_somma   ; Ripete il ciclo se nel frattempo
                    ; ECX non si è azzerato.
end_do_somma:
    mov ebx, eax    ; Restituisce il valore ottenuto
    mov eax, 1      ; dalla somma.
    int 0x80        ;
```

Come si può vedere, il ciclo di incremento è racchiuso tra i simboli '**do_somma**' e '**end_do_somma**'; inoltre, prima di entrare nel ciclo

di somma, viene verificato che il secondo addendo sia diverso da zero, perché se è pari a zero, viene saltato il ciclo di somma, dal momento che ***EAX*** contiene già il valore corretto.

Viene mostrata una seconda versione del ciclo, dove si sostituisce l'istruzione '**LOOP**' con altre istruzioni di salto condizionato:

```
    cmp $0, %ecx     # Se il secondo addendo è zero, non
    je end_do_somma # esegue il ciclo di incrementi.
do_somma:
    inc %eax         # Aggiunge una unità a EAX.
    dec %ecx         # Riduce ECX di una unità.
    cmp $0, %ecx     # Se il secondo addendo è ancora diverso
    jnz do_somma     # da zero, allora ripete il ciclo.
end_do_somma:
```

```
    cmp ecx, 0       ; Se il secondo addendo è zero, non
    je end_do_somma ; esegue il ciclo di incrementi.
do_somma:
    inc eax          ; Aggiunge una unità a EAX.
    dec ecx          ; Riduce ECX di una unità.
    cmp ecx, 0       ; Se il secondo addendo è ancora diverso
    jnz do_somma     ; da zero, allora ripete il ciclo.
end_do_somma:
```

Viene mostrata una terza versione del ciclo, dove il controllo di uscita avviene solo all'inizio:

```
do_somma:
    cmp $0, %ecx     # Se il secondo addendo è zero, esce dal
    je end_do_somma # ciclo di incrementi.
    dec %ecx         # Riduce di una unità ECX.
    inc %eax         # Aggiunge una unità a EAX.
    jmp do_somma     # Ritorna all'inizio del ciclo.
end_do_somma:
```

```
do_somma:
    cmp ecx, 0       ; Se il secondo addendo è zero, esce dal
    je end_do_somma  ; ciclo di incrementi.
    dec ecx          ; Riduce di una unità ECX.
    inc eax          ; Aggiunge una unità a EAX.
    jmp do_somma     ; Ritorna all'inizio del ciclo.
end_do_somma:
```

64.9.2 Moltiplicazione attraverso la somma

Viene mostrato un programma che esegue la moltiplicazione di due valori interi senza segno, sommando progressivamente il moltiplicando a un registro che inizialmente è pari a zero, corrispondentemente alla riduzione di una unità del moltiplicatore. Quando il moltiplicatore è stato ridotto a zero, il registro che viene incrementato contiene il risultato della moltiplicazione. Il ciclo è controllato dall'istruzione '**LOOP**':

```
# op1 * op2
#
.section .data
op1:    .int    0x00000007      # Intero senza segno.
op2:    .int    0x00000003      # Intero senza segno.
#
.section .text
.globl _start
#
_start:
    mov op1, %eax           # Moltiplicando.
    mov op2, %ecx           # Moltiplicatore.
    mov $0, %ebx            # Risultato.
bp1:
    cmp $0, %ecx            # Se il moltiplicatore è zero,
    je end_do_moltiplica # non esegue il ciclo di somme.
```

```
do_moltiplica:
    add %eax, %ebx        # Aggiunge il moltiplicando al
                          # risultato.
    loop do_moltiplica    # Ripete il ciclo se nel frattempo
                          # ECX non si è azzerato.
end_do_moltiplica:
    mov $1,   %eax        # Restituisce il valore ottenuto
    int $0x80             # dalla moltiplicazione.
```

```
; op1 * op2
;
section .data
op1:    dd      0x00000007      ; Intero senza segno.
op2:    dd      0x00000003      ; Intero senza segno.
;
section .text
global _start
;
_start:
    mov eax, [op1]        ; Moltiplicando.
    mov ecx, [op2]        ; Moltiplicatore.
    mov ebx, 0            ; Risultato.
bp1:
    cmp ecx, 0            ; Se il moltiplicatore è zero,
    je end_do_moltiplica ; non esegue il ciclo di somme.
do_moltiplica:
    add ebx, eax          ; Aggiunge il moltiplicando al
                          ; risultato.
    loop do_moltiplica    ; Ripete il ciclo se nel frattempo
                          ; ECX non si è azzerato.
end_do_moltiplica:
    mov eax, 1            ; Restituisce il valore ottenuto
    int 0x80              ; dalla moltiplicazione.
```

Viene mostrata una seconda versione del ciclo, dove si sostituisce l'istruzione '**LOOP**' con altre istruzioni di salto condizionato:

```
    cmp $0, %ecx          # Se il moltiplicatore è zero,
    je end_do_moltiplica  # non esegue il ciclo di somme.
do_moltiplica:
    add %eax, %ebx        # Aggiunge il moltiplicando al
                          # risultato.
    cmp $0, %ecx          # Se il moltiplicatore è ancora
    jnz do_moltiplica     # diverso da zero, allora ripete
                          # il ciclo.
end_do_moltiplica:
```

```
    cmp ecx, 0            ; Se il moltiplicatore è zero,
    je end_do_moltiplica  ; non esegue il ciclo di somme.
do_moltiplica:
    add ebx, eax          ; Aggiunge il moltiplicando al
                          ; risultato.
    cmp ecx, 0            ; Se il moltiplicatore è ancora
    jnz do_moltiplica     ; diverso da zero, allora ripete
                          ; il ciclo.
end_do_moltiplica:
```

Viene mostrata una terza versione del ciclo, dove il controllo di uscita avviene solo all'inizio:

```
do_moltiplica:
    cmp $0, %ecx          # Se il moltiplicatore è zero,
    je end_do_moltiplica  # esce dal ciclo di somme.
    dec %ecx              # Riduce di una unità il
                          # moltiplicatore.
    add %eax, %ebx        # Aggiunge il moltiplicando al
                          # risultato.
    jmp do_moltiplica     # Ritorna all'inizio del ciclo.
end_do_moltiplica:
```

```
do_moltiplica:
    cmp ecx, 0            ; Se il moltiplicatore è zero,
    je end_do_moltiplica  ; esce dal ciclo di somme.
    dec ecx               ; Riduce di una unità il
                          ; moltiplicatore.
    add ebx, eax          ; Aggiunge il moltiplicando al
                          ; risultato.
    jmp do_moltiplica     ; Ritorna all'inizio del ciclo.
end_do_moltiplica:
```

64.9.3 Divisione attraverso la sottrazione

«

Viene mostrato un programma che esegue la divisione di due valori interi senza segno, sottraendo progressivamente il divisore a un registro che inizialmente è pari al valore del dividendo, corrispondentemente all'incremento di una unità del risultato (a partire da zero). Quando il registro che viene ridotto, di volta in volta, del valore del divisore, diventa minore del divisore, la divisione termina. Viene proposta una sola versione, con un ciclo controllato da una condizione iniziale:

```
# op1 / op2
#
.section .data
op1:    .int    33      # Intero senza segno.
op2:    .int    12      # Intero senza segno.
#
.section .text
.globl _start
#
_start:
    mov op1,  %eax       # Dividendo.
    mov op2,  %ecx       # Divisore.
```

```
    mov $0,   %ebx        # Risultato.
    mov %eax, %edx        # Resto.
do_dividi:
    cmp %ecx, %edx        # Se il resto è minore del
    jb end_do_dividi      # divisore, conclude il ciclo di
                          # sottrazioni.
    sub %ecx, %edx        # Sottrae al resto il divisore.
    inc %ebx              # Incrementa il risultato di una
                          # unità.
    jmp do_dividi         # Torna all'inizio del ciclo.
end_do_dividi:
    mov $1,   %eax        # Restituisce il valore ottenuto
    int $0x80             # dalla moltiplicazione.
```

```
; op1 / op2
;
section .data
op1:    dd      33      ; Intero senza segno.
op2:    dd      12      ; Intero senza segno.
;
section .text
global _start
;
_start:
    mov eax, [op1]        ; Dividendo.
    mov ecx, [op2]        ; Divisore.
    mov ebx, 0            ; Risultato.
    mov edx, eax          ; Resto.
do_dividi:
    cmp edx, ecx          ; Se il resto è minore del
    jb end_do_dividi      ; divisore, conclude il ciclo di
                          ; sottrazioni.
    sub edx, ecx          ; Sottrae al resto il divisore.
    inc ebx               ; Incrementa il risultato di una
                          ; unità.
```

```
    jmp do_dividi          ; Torna all'inizio del ciclo.
end_do_dividi:
    mov eax, 1             ; Restituisce il valore ottenuto
    int 0x80               ; dalla moltiplicazione.
```

64.9.4 Elevamento a potenza

«

Viene mostrato un programma che calcola la potenza di due numeri interi senza segno, moltiplicando progressivamente la base, corrispondentemente al decremento di un contatore che parte dal valore dell'esponente. Quando il contatore raggiunge lo zero, il ciclo di moltiplicazioni termina e il risultato della potenza viene emesso come valore di uscita, ammesso che sia abbastanza piccolo da poter essere rappresentato:

```
# op1 / op2
#
.section .data
op1:    .int    5      # Intero senza segno.
op2:    .int    3      # Intero senza segno.
#
.section .text
.globl _start
#
_start:
    mov op1,  %esi     # Base.
    mov op2,  %edi     # Esponente.
    mov $0,   %edx     # Risultato: EDX:EAX
    mov $1,   %eax     #
    mov %edi, %ecx     # Contatore.
do_potenza:
    cmp $0, %ecx       # Se il contatore ha raggiunto lo
    jz end_do_potenza  # zero, conclude il ciclo di
```

```
                        # moltiplicazioni.
    mul %esi            # EDX:EAX := EAX*ESI.
    dec %ecx            # Riduce di una unità il contatore.
    jmp do_potenza      # Torna all'inizio del ciclo.
end_do_potenza:
    mov %eax, %ebx      # Restituisce il valore della potenza,
    mov $1,   %eax      # ammesso che sia abbastanza piccolo
    int $0x80           # da poter essere rappresentato come
                        # valore di uscita.
```

```
; op1 / op2
;
section .data
op1:    dd      5       ; Intero senza segno.
op2:    dd      3       ; Intero senza segno.
;
section .text
global _start
;
_start:
    mov esi, [op1]      ; Base.
    mov edi, [op2]      ; Esponente.
    mov edx, 0          ; Risultato: EDX:EAX
    mov eax, 1          ;
    mov ecx, edi        ; Contatore.
do_potenza:
    cmp ecx, 0          ; Se il contatore ha raggiunto lo
    jz end_do_potenza   ; zero, conclude il ciclo di
                        ; moltiplicazioni.
    mul esi             ; EDX:EAX := EAX*ESI.
    dec ecx             ; Riduce di una unità il contatore.
    jmp do_potenza      ; Torna all'inizio del ciclo.
end_do_potenza:
    mov ebx, eax        ; Restituisce il valore della potenza,
    mov eax, 1          ; ammesso che sia abbastanza piccolo
```

```
    int 0x80            ; da poter essere rappresentato come
                        ; valore di uscita.
```

64.9.5 Moltiplicazione attraverso lo scorrimento e la somma

«

Viene mostrato un programma che calcola il prodotto di due numeri interi senza segno, sommando progressivamente il moltiplicando che viene fatto scorrere verso sinistra. Per comprendere il procedimento occorre fare mente locale al modo in cui la moltiplicazione verrebbe eseguita a mano: il moltiplicando viene sommato al risultato (che inizialmente è pari a zero) se la cifra meno significativa del moltiplicatore è pari a uno; successivamente il moltiplicando viene fatto scorrere verso sinistra di una posizione e lo si somma nuovamente al risultato se la cifra successiva del moltiplicatore è pari a uno; quindi si continua fino a che si esauriscono le cifre del moltiplicatore.

Nel programma mostrato, durante il ciclo di somme, il moltiplicatore viene fatto scorrere verso destra, in modo da poter verificare il valore della cifra espulsa attraverso l'indicatore di riporto (*carry*). Così facendo, quando il moltiplicatore è pari a zero, il ciclo di somme può terminare.

```
# op1 * op2
#
.section .data
op1:    .byte   0x07, 0x00 # little endian = 0x0007
                           # intero senza segno.
op2:    .byte   0x03, 0x00 # little endian = 0x0003
                           # intero senza segno.
#
.section .text
```

```
.globl _start
#
_start:
    movzx op1,  %edx        # Moltiplicando.
    movzx op2,  %ecx        # Moltiplicatore.
    mov   $0,   %eax        # Risultato.
do_mol:
    cmp   $0,   %ecx        # Se il moltiplicatore è pari a
    jz    end_do_mol        # zero, il ciclo deve terminare.
    shr   $1,   %ecx        # Fa scorrere a destra il
    jnc   end_do_mol_somma  # moltiplicatore e se
                            # l'indicatore di riporto non è
                            # attivo, salta la somma.
do_mol_somma:
    add   %edx, %eax        # Aggiunge il moltiplicando al
                            # risultato.
end_do_mol_somma:
    shl   $1,   %edx        # Fa scorrere il moltiplicando
                            # a sinistra.
    jmp   do_mol            # Torna all'inizio del ciclo.
end_do_mol:
    mov   %eax, %ebx        # Restituisce il valore del
    mov   $1,   %eax        # prodotto, ammesso che sia
    int   $0x80             # abbastanza piccolo da poter
                            # essere rappresentato come
                            # valore di uscita.
```

```
; op1 * op2
;
section .data
op1:    dw      0x0007      ; Intero senza segno.
op2:    dw      0x0003      ; Intero senza segno.
;
section .text
global _start
```

```
;
_start:
    movzx edx, word [op1]  ; Moltiplicando.
    movzx ecx, word [op2]  ; Moltiplicatore.
    mov   eax,      0      ; Risultato.
do_mol:
    cmp   ecx,      0      ; Se il moltiplicatore è pari a
    jz    end_do_mol       ; zero, il ciclo deve terminare.
    shr   ecx,      1      ; Fa scorrere a destra il
    jnc   end_do_mol_somma ; moltiplicatore e se
                           ; l'indicatore di riporto non è
                           ; attivo, salta la somma.
do_mol_somma:
    add   eax,      edx    ; Aggiunge il moltiplicando al
                           ; risultato.
end_do_mol_somma:
    shl   edx,      1      ; Fa scorrere il moltiplicando a
                           ; sinistra.
    jmp   do_mol           ; Torna all'inizio del ciclo.
end_do_mol:
    mov   ebx,      eax    ; Restituisce il valore del
    mov   eax,      1      ; prodotto, ammesso che sia
    int   0x80             ; abbastanza piccolo da poter
                           ; essere rappresentato come
                           ; valore di uscita.
```

Come si può vedere, moltiplicando e moltiplicatore sono variabili da 16 bit, in modo da avere la certezza che il risultato del prodotto sia contenibile in un registro. Nel caso del primo listato, fatto per GNU AS, lo spazio in memoria viene dichiarato come sequenza di due byte, perché manca la possibilità di definire esplicitamente un intero «corto».

64.9.6 Conteggio dei bit a uno

Viene proposto un esempio di programma che conta quanti bit a uno compongono un certo valore numerico. Per farlo, si usa lo scorrimento (in questo caso è a destra, ma non farebbe differenza) e quindi viene contato il riporto (l'indicatore di riporto è attivo se fuoriesce una cifra a uno).

```
#
.section .data
op1:       .int 44
#
.section .text
.globl _start
#
_start:
    mov   op1, %eax     # EAX contiene il numero di cui
                        # contare i bit a 1.
    mov   $0, %ecx      # ECX è il contatore dei bit a uno.
do_conta:
    cmp   $0, %eax      # Se EAX è a zero, il conteggio dei
    jz    end_do_conta  # bit si conclude.
    shr   $1, %eax      # Fa scorrere a destra EAX.
    adc   $0, %ecx      # ECX = ECX + 0 + carry.
    jmp   do_conta      # Riprende il ciclo.
end_do_conta:
    mov   %ecx, %ebx    # Restituisce la quantità di bit.
    mov   $1, %eax      #
    int   $0x80         #
```

```
;
section .data
op1:       dd 44
;
section .text
```

```
global _start
;
_start:
    mov   eax, [op1]    ; EAX contiene il numero di cui
                        ; contare i bit a 1.
    mov   ecx, 0        ; ECX è il contatore dei bit a uno.
do_conta:
    cmp   eax, 0        ; Se EAX è a zero, il conteggio dei
    jz    end_do_conta  ; bit si conclude.
    shr   eax, 1        ; Fa scorrere a destra EAX.
    adc   ecx, 0        ; ECX = ECX + 0 + carry.
    jmp   do_conta      ; Riprende il ciclo.
end_do_conta:
    mov   ebx, ecx      ; Restituisce la quantità di bit.
    mov   eax, 1        ;
    int   0x80          ;
```

Viene proposto un metodo alternativo che utilizza la sottrazione e l'abbinamento con l'operatore logico AND (è descritto nella sezione 63.5.10):

```
#
.section .data
op1:       .int 44
#
.section .text
.globl _start
#
_start:
    mov   op1, %eax     # EAX contiene il numero di cui
                        # contare i bit a 1.
    mov   $0, %ecx      # ECX è il contatore dei bit a uno.
do_conta:
    cmp   $0, %eax      # Se EAX è a zero, il conteggio
    jz    end_do_conta  # dei bit si conclude.
```

```
    mov   %eax, %edx    # Fa una copia in EDX.
    dec   %eax          # Decrementa EAX di una unità.
    and   %edx, %eax    # EAX := EAX AND EDX.
    inc   %ecx          # ECX++
    jmp   do_conta      # Riprende il ciclo.
end_do_conta:
    mov   %ecx, %ebx    # Restituisce la quantità di bit.
    mov   $1, %eax      #
    int   $0x80         #
```

```
;
section .data
op1:      dd 44
;
section .text
global _start
;
_start:
    mov   eax, [op1]    ; EAX contiene il numero di cui
                        ; contare i bit a 1.
    mov   ecx, 0        ; ECX è il contatore dei bit a uno.
do_conta:
    cmp   eax, 0        ; Se EAX è a zero, il conteggio dei
    jz    end_do_conta  ; bit si conclude.
    mov   edx, eax      ; Fa una copia in EDX.
    dec   eax           ; Decrementa EAX di una unità.
    and   eax, edx      ; EAX := EAX AND EDX.
    inc   ecx           ; ECX++
    jmp   do_conta      ; Riprende il ciclo.
end_do_conta:
    mov   ebx, ecx      ; Restituisce la quantità di bit.
    mov   eax, 1        ;
    int   0x80          ;
```

64.10 Funzioni

«

Attraverso l'uso delle istruzioni '**CALL**' e '**RET**' è possibile realizzare delle subroutine, ovvero qualcosa che assomigli alle funzioni dei linguaggi di programmazione più evoluti.

```
CALL indirizzo
```

```
RET
```

L'istruzione '**CALL**', prima di passare il controllo all'indirizzo di memoria indicato,[9] salva l'indirizzo dell'istruzione successiva alla chiamata nella pila dei dati (*stack*). Per converso, l'istruzione '**RET**' recupera dalla pila l'ultimo elemento e passa il controllo all'istruzione che si trova all'indirizzo contenuto in tale elemento estratto.

64.10.1 Esempio banale di chiamata

«

Per comprendere il meccanismo con cui si realizzano le subroutine, si prende in considerazione un esempio, già descritto, che calcola il prodotto tra due numeri attraverso lo scorrimento dei valori. Come sempre, vengono mostrati due listati equivalenti, fatti rispettivamente per GNU AS e NASM.

```
1  # op1 * op2
2  #
3  .section .data
4  op1:    .byte   0x07, 0x00    # little endian = 0x0005
5                                # intero senza segno.
6  op2:    .byte   0x03, 0x00    # little endian = 0x0003
7                                # intero senza segno.
```

```
8   #
9   .section .text
10  .globl _start
11  #-----------------------------------------------------
12  _start:
13      movzx op1,  %edx        # Moltiplicando.
14      movzx op2,  %ecx        # Moltiplicatore.
15      mov   $0,   %eax        # Risultato.
16  bp1:
17      call  f_mol             # Esegue la moltiplicazione:
18                              # EAX = EDX * ECX
19  bp2:
20      mov   %eax, %ebx        # Restituisce il valore del
21      mov   $1,   %eax        # prodotto, ammesso che sia
22      int   $0x80             # abbastanza piccolo da poter
23                              # essere rappresentato come
24                              # valore di uscita.
25  #-----------------------------------------------------
26  # Moltiplicazione di due numeri interi.
27  # EAX = EDX * ECX
28  # I registri EDX e ECX vengono alterati durante il
29  # procedimento.
30  #
31  f_mol:
32      cmp   $0,   %ecx        # Se il moltiplicatore è
33      jz    f_end_mol         # pari a zero, il ciclo
34                              # deve terminare.
35      shr   $1,   %ecx        # Fa scorrere a destra il
36      jnc   end_do_somma      # moltiplicatore e se
37                              # l'indicatore di riporto
38                              # non è attivo, salta
39                              # la somma.
40  do_somma:
41      add   %edx, %eax        # Aggiunge il moltiplicando
```

```
42                                  # al risultato.
43  end_do_somma:
44      shl   $1,   %edx            # Fa scorrere il
45                                  # moltiplicando a sinistra.
46      jmp   f_mol                 # Torna all'inizio della
47                                  # funzione.
48  f_end_mol:
49      ret                         # Torna all'istruzione
50                                  # successiva alla chiamata.
```

```
1   ; op1 * op2
2   ;
3   section .data
4   op1:    dw      0x0007    ; Intero senza segno.
5   op2:    dw      0x0003    ; Intero senza segno.
6   ;
7   section .text
8   global _start
9   ;-------------------------------------------------
10  _start:
11      movzx edx, word [op1]  ; Moltiplicando.
12      movzx ecx, word [op2]  ; Moltiplicatore.
13      mov   eax,      0      ; Risultato.
14  bp1:
15      call  f_mol            ; Esegue la moltiplicazione:
16                             ; EAX = EDX * ECX
17  bp2:
18      mov   ebx,      eax    ; Restituisce il valore
19      mov   eax,      1      ; del prodotto, ammesso che
20      int   0x80             ; sia abbastanza piccolo da
21                             ; poter essere rappresentato
22                             : come valore di uscita.
23  ;-------------------------------------------------
24  ; Moltiplicazione di due numeri interi.
```

```
25  ; EAX = EDX * ECX
26  ; I registri EDX e ECX vengono alterati durante il
27  ; procedimento.
28  ;
29  f_mol:
30      cmp   ecx,      0       ; Se il moltiplicatore è
31      jz    f_end_mol         ; pari a zero, il ciclo deve
32                              ; terminare.
33      shr   ecx,      1       ; Fa scorrere a destra il
34      jnc   end_do_somma      ; moltiplicatore e se
35                              ; l'indicatore di riporto
36                              ; non è attivo, salta
37                              ; la somma.
38  do_somma:
39      add   eax,      edx     ; Aggiunge il moltiplicando
40                              ; al risultato.
41  end_do_somma:
42      shl   edx,      1       ; Fa scorrere il
43                              ; moltiplicando a sinistra.
44      jmp   f_mol             ; Torna all'inizio della
45                              ; funzione.
46  f_end_mol:
47      ret                     ; Torna all'istruzione
48                              ; successiva alla chiamata.
```

Si osservi che il programma viene eseguito a partire dalla riga 11 e che si conclude alla riga 19. La subroutine, ovvero la funzione che esegue la moltiplicazione, si trova in un gruppo di istruzioni successive, il cui inizio è segnalato dal simbolo '**f_mol**'.

La funzione riceve il moltiplicando e il moltiplicatore attraverso due registri, utilizzando a sua volta un altro registro per restituire il risultato.

64.10.2 Salvataggio dei registri prima della chiamata

«

Ogni funzione ha la necessità di elaborare dati senza interferire con il resto del programma; in pratica, ogni funzione deve poter utilizzare i registri con una certa libertà. Nell'esempio precedente vengono utilizzati dei registri per passare alla funzione i valori da moltiplicare, ma all'interno della funzione il contenuto dei registri viene modificato. Per lo scopo dell'esempio, il fatto che i registri ***ECX*** e ***EDX*** vengano modificati, non produce effetti collaterali, ma in un programma più complesso potrebbe essere il caso di salvaguardare il contenuto originale dei registri prima della chiamata di una funzione. L'esempio successivo mostra una variante del codice contenuto tra i simboli '**bp1**' e '**bp2**', allo scopo di conservare una copia dei registri che contengono il moltiplicando e il moltiplicatore, per ripristinarla dopo la chiamata:

```
bp1:
    push  %ecx  # Salva il moltiplicatore nella pila.
    push  %edx  # Salva il moltiplicando nella pila.
    call  f_mol # Esegue la moltiplicazione: EAX = EDX * ECX
    pop   %edx  # Recupera il moltiplicando dalla pila.
    pop   %ecx  # Recupera il moltiplicatore dalla pila.
bp2:
```

```
bp1:
    push  ecx   ; Salva il moltiplicatore nella pila.
    push  edx   ; Salva il moltiplicando nella pila.
    call  f_mol ; Esegue la moltiplicazione: EAX = EDX * ECX
    pop   edx   ; Recupera il moltiplicando dalla pila.
    pop   ecx   ; Recupera il moltiplicatore dalla pila.
bp2:
```

Come si può intendere, il recupero dei valori dalla pila deve avvenire in senso inverso.

64.10.3 Passaggio di parametri attraverso la pila

Per rendere più libera la funzione dal programma chiamante, conviene utilizzare la stessa pila per il passaggio dei parametri. In pratica, dopo avere salvato i registri che contengono dati importanti (ammesso che ciò vada fatto), occorre accumulare nella pila gli argomenti della chiamata della funzione, secondo un ordine convenuto per la funzione stessa. All'interno della funzione, poi, si vanno a pescare questi valori per usarli nell'elaborazione.

```
# op1 * op2
#
.section .data
op1:    .byte   0x07, 0x00 # little endian = 0x0005
                           # intero senza segno.
op2:    .byte   0x03, 0x00 # little endian = 0x0003
                           # intero senza segno.
#
.section .text
.globl _start
#-------------------------------------------------------
_start:
    movzx op1,  %edx # Moltiplicando.
    movzx op2,  %ecx # Moltiplicatore.
bp1:
    push  %ecx       # Salva il moltiplicatore nella pila.
    push  %edx       # Salva il moltiplicando nella pila.
    #
    push  %ecx       # Inserisce il secondo parametro nella
                     # pila.
    push  %edx       # Inserisce il primo parametro nella
```

```
                        # pila.
    call  f_mol         # Esegue la moltiplicazione:
                        # EAX = EDX * ECX
    add   $4,   %esp    # Espelle il primo parametro della
                        # chiamata.
    add   $4,   %esp    # Espelle il secondo parametro della
                        # chiamata.
    #
    pop   %edx          # Recupera il moltiplicando dalla pila.
    pop   %ecx          # Recupera il moltiplicatore dalla
                        # pila.
bp2:
    mov   %eax, %ebx    # Restituisce il valore del prodotto,
    mov   $1,   %eax    # ammesso che sia abbastanza piccolo
    int   $0x80         # da poter essere rappresentato come
                        # valore di uscita.
#-----------------------------------------------------------
# Moltiplicazione di due numeri interi.
# f_mol (a, b) => EAX
# EAX = a * b
#
f_mol:
    mov   4(%esp), %edx # Copia il primo parametro in EDX.
    mov   8(%esp), %ecx # Copia il secondo parametro in ECX.
    mov   $0,      %eax # Azzera EAX per sicurezza.
do_mol:
    cmp   $0,      %ecx # Se il moltiplicatore è pari a
    jz    end_do_mol    # zero, il ciclo deve terminare.
    shr   $1,      %ecx # Fa scorrere a destra il
    jnc   end_do_somma  # moltiplicatore e se l'indicatore
                        # di riporto non è attivo, salta
                        # la somma.
do_somma:
    add   %edx,    %eax # Aggiunge il moltiplicando al
```

```
                        # risultato.
end_do_somma:
    shl   $1,       %edx # Fa scorrere il moltiplicando a
                         # sinistra.
    jmp   do_mol         # Torna all'inizio del ciclo di
                         # moltiplicazione.
end_do_mol:
    ret                  # Torna all'istruzione successiva
                         # alla chiamata.
```

```
; op1 * op2
;
section .data
op1:    dw      0x0007      ; Intero senza segno.
op2:    dw      0x0003      ; Intero senza segno.
;
section .text
global _start
;-------------------------------------------------------
_start:
    movzx edx, word [op1]   ; Moltiplicando.
    movzx ecx, word [op2]   ; Moltiplicatore.
bp1:
    push  ecx       ; Salva il moltiplicatore nella pila.
    push  edx       ; Salva il moltiplicando nella pila.
    ;
    push  ecx       ; Inserisce il secondo parametro nella
                    ; pila.
    push  edx       ; Inserisce il primo parametro nella
                    ; pila.
    call  f_mol     ; Esegue la moltiplicazione:
                    ; EAX = EDX * ECX
    add   esp,  4   ; Espelle il primo parametro della
                    ; chiamata.
```

```
    add   esp,  4    ; Espelle il secondo parametro della
                     ; chiamata.
    ;
    pop   edx        ; Recupera il moltiplicando dalla pila.
    pop   ecx        ; Recupera il moltiplicatore dalla pila.
bp2:
    mov   ebx,  eax ; Restituisce il valore del prodotto,
    mov   eax,  1    ; ammesso che sia abbastanza piccolo
    int   0x80       ; da poter essere rappresentato come
                     ; valore di uscita.
;-----------------------------------------------------------
; Moltiplicazione di due numeri interi.
; f_mol (a, b) => EAX
; EAX = a * b
;
f_mol:
    mov   edx,  [esp+4] ; Copia il primo parametro in EDX.
    mov   ecx,  [esp+8] ; Copia il secondo parametro in ECX.
    mov   eax,  0       ; Azzera EAX per sicurezza.
    ;
do_mol:
    cmp   ecx,  0       ; Se il moltiplicatore è pari a zero,
    jz    end_do_mol    ; il ciclo deve terminare.
    shr   ecx,  1       ; Fa scorrere a destra il
    jnc   end_do_somma ; moltiplicatore e se l'indicatore di
                        ; riporto non è attivo, salta
                        ; la somma.
do_somma:
    add   eax,  edx     ; Aggiunge il moltiplicando al
                        ; risultato.
end_do_somma:
    shl   edx,  1       ; Fa scorrere il moltiplicando a
                        ; sinistra.
    jmp   do_mol        ; Torna all'inizio del ciclo di
```

```
                        ; moltiplicazione.
end_do_mol:
    ret                 ; Torna all'istruzione successiva
                        ; alla chiamata.
```

Come si vede, rispetto alla versione precedente dello stesso programma, il risultato del prodotto, calcolato all'interno della funzione, continua a essere restituito attraverso il registro ***EAX***, ma sarebbe stato possibile accumulare nella pila, prima della chiamata, un valore in più, da considerare poi come il risultato generato dalla funzione.

All'inizio della funzione vengono recuperati i valori che costituiscono gli argomenti della chiamata. Tale operazione viene eseguita attraverso queste istruzioni:

```
    mov 4(%esp), %edx # Copia il primo parametro in EDX.
    mov 8(%esp), %ecx # Copia il secondo parametro in ECX.
```

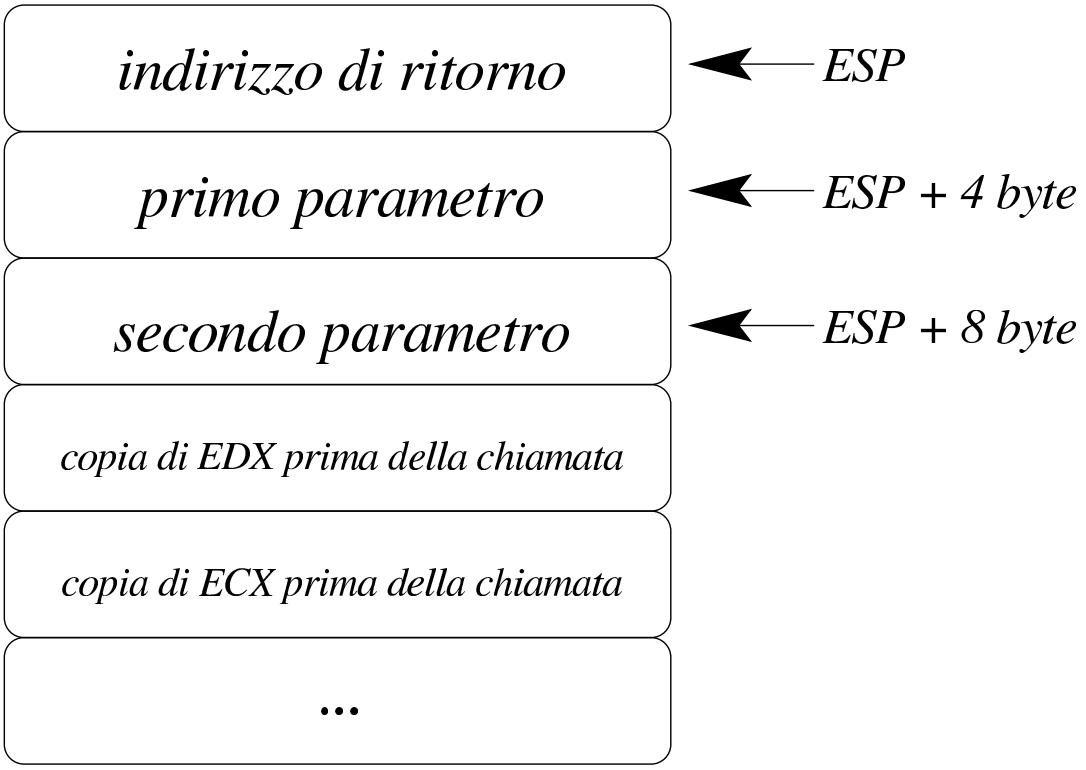
```
    mov edx, [esp+4] ; Copia il primo parametro in EDX.
    mov ecx, [esp+8] ; Copia il secondo parametro in ECX.
```

L'operando '`4(%esp)`', ovvero '`[esp+4]`', individua l'indirizzo di memoria corrispondente al valore del registro ***ESP***, più quattro byte.

Il registro ***ESP*** è l'indice della pila (*stack pointer*) che punta all'ultimo elemento presente (quello in cima alla pila). Nei sistemi Unix (compresi i sistemi GNU) la pila parte da una posizione elevata della memoria e «cresce» utilizzando indirizzi che invece decrescono. Pertanto, considerato che l'ultimo elemento della pila è l'indirizzo di ritorno, l'elemento immediatamente precedente lo si raggiunge quattro byte dopo (32 bit) e quello ancora precedente si trova otto

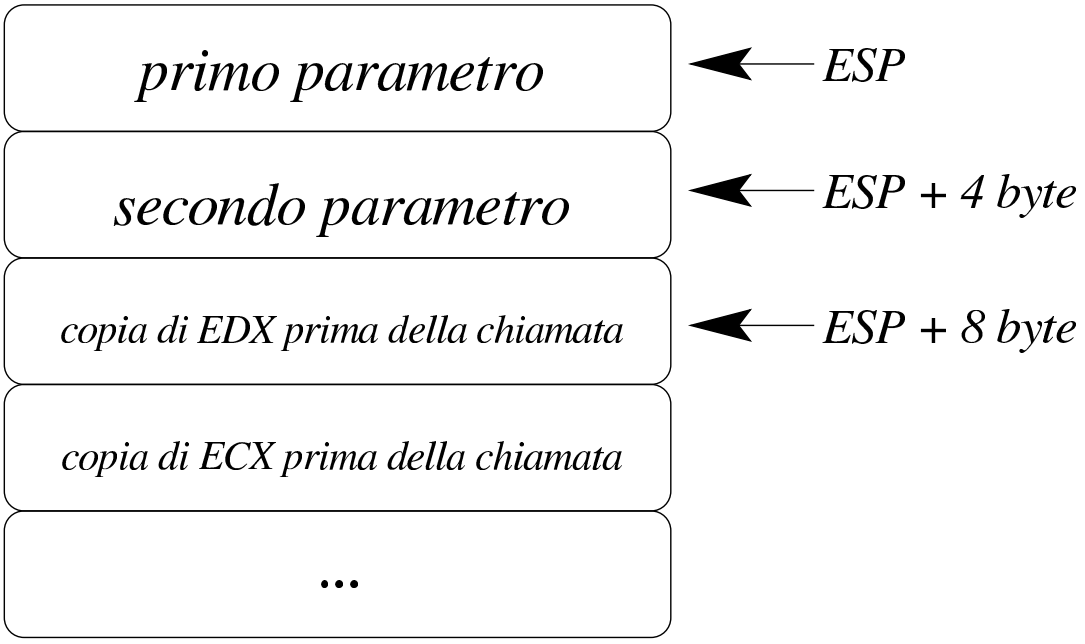
byte dopo la posizione finale.

Figura 64.169. Situazione della pila in corrispondenza del simbolo '**`f_mol`**'.



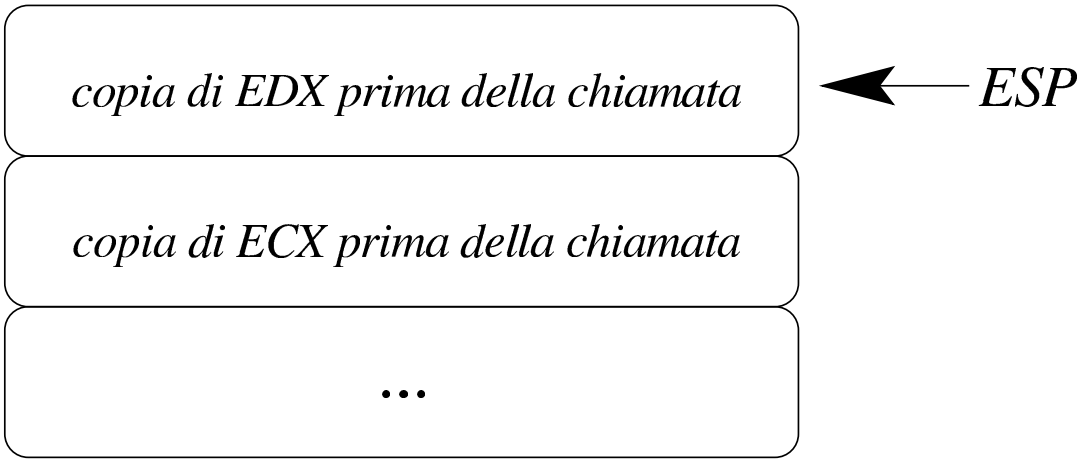
La funzione elabora il prodotto dei valori forniti come argomento e ne lascia il risultato nel registro ***EAX***. Al ritorno, la pila si presenta come si vede nella figura successiva:

Figura 64.170. Situazione della pila immediatamente dopo la chiamata della funzione.



Come si vede, occorre espellere dalla pila i parametri usati per la chiamata. Dal momento che non c'è bisogno di rileggere il loro valore, ci si limita a decrementare l'indice della pila, ovvero si incrementa il valore del registro ***ESP*** a gruppi di quattro byte alla volta.

Figura 64.171. Situazione della pila dopo l'espulsione dei parametri della chiamata.



Naturalmente, considerato che la funzione non altera i valori ac-

cumulati nella pila, la chiamata potrebbe essere semplificata un po':

```
bp1:
    push  %ecx   # Salva ECX, inserendolo come secondo
                 # parametro nella pila.
    push  %edx   # Salva EDX, inserendolo come primo
                 # parametro nella pila.
    call  f_mol # Esegue la moltiplicazione: EAX = EDX * ECX
    pop   %edx   # Recupera EDX dalla pila.
    pop   %ecx   # Recupera ECX dalla pila.
bp2:
```

```
bp1:
    push  ecx    ; Salva ECX, inserendolo come secondo
                 ; parametro nella pila.
    push  edx    ; Salva EDX, inserendolo come primo
                 ; parametro nella pila.
    call  f_mol ; Esegue la moltiplicazione: EAX = EDX * ECX
    pop   edx    ; Recupera EDX dalla pila.
    pop   ecx    ; Recupera ECX dalla pila.
bp2:
```

64.10.4 Utilizzo del registro «EBP»

«

Perché una funzione possa gestire delle variabili «locali», ovvero tali da avere un campo di azione limitato alla funzione stessa, senza lasciare tracce al ritorno dalla chiamata, si deve usare la pila aggiungendovi altri elementi. Questo fatto complica l'accesso ai parametri della chiamata, perché durante l'esecuzione delle istruzioni della funzione, l'indice della pila può spostarsi. A questo proposito, all'inizio di una funzione, conviene conservare una copia del registro ***ESP*** in un altro registro apposito: ***EBP*** (*base pointer*). In pra-

tica, l'indice contenuto in ***EBP*** dovrebbe essere sempre usato per rappresentare la posizione in cui si trova l'indirizzo di ritorno della funzione in cui ci si trova.

Viene riproposto il programma già presentato nella sezione precedente, con le semplificazioni già descritte a proposito della chiamata e con le modifiche relative all'uso del registro ***EBP***.

```
# op1 * op2
#
.section .data
op1:    .byte   0x07, 0x00 # little endian = 0x0005
                           # intero senza segno.
op2:    .byte   0x03, 0x00 # little endian = 0x0003
                           # intero senza segno.
#
.section .text
.globl _start
#-------------------------------------------------------
_start:
    movzx op1,  %edx # Moltiplicando.
    movzx op2,  %ecx # Moltiplicatore.
    mov   $0,   %eax # Risultato.
bp1:
    push  %ebp       # Salva il registro EBP
                     # prima della chiamata.
    push  %ecx       # Inserisce il moltiplicando nella
                     # pila.
    push  %edx       # Inserisce il moltiplicatore nella
                     # pila.
    call  f_mol      # Esegue la moltiplicazione:
                     # EAX = EDX * ECX
    pop   %edx       # Recupera il moltiplicando dalla pila.
    pop   %ecx       # Recupera il moltiplicatore dalla
                     # pila.
```

```
    pop   %ebp          # Recupera il registro EBP
                        # dopo la chiamata.
bp2:
    mov   %eax, %ebx # Restituisce il valore del prodotto,
    mov   $1,   %eax # ammesso che sia abbastanza piccolo
    int   $0x80         # da poter essere rappresentato come
                        # valore di uscita.
#-----------------------------------------------------------
# Moltiplicazione di due numeri interi.
# f_mol (a, b) => EAX
# EAX = a * b
#
f_mol:
    mov   %esp,  %ebp    # Copia ESP in EBP.
    mov   4(%ebp), %edx # Copia il primo parametro in EDX.
    mov   8(%ebp), %ecx # Copia il secondo parametro in ECX.
    mov   $0,    %eax    # Azzera EAX per sicurezza.
    #
do_mol:
    cmp   $0,    %ecx    # Se il moltiplicatore è pari a
    jz    end_do_mol     # zero, il ciclo deve terminare.
    shr   $1,    %ecx    # Fa scorrere a destra il
    jnc   end_do_somma  # moltiplicatore e se l'indicatore
                        # di riporto non è attivo, salta la
                        # somma.
do_somma:
    add   %edx,  %eax    # Aggiunge il moltiplicando al
                        # risultato.
end_do_somma:
    shl   $1,    %edx    # Fa scorrere il moltiplicando a
                        # sinistra.
    jmp   do_mol         # Torna all'inizio del ciclo di
                        # moltiplicazione.
end_do_mol:
```

```
    mov %ebp,    %esp   # Ripristina ESP, espellendo
                        # le variabili locali.
    ret                 # Torna all'istruzione successiva
                        # alla chiamata.
```

```
; op1 * op2
;
section .data
op1:    dw      0x0007      ; Intero senza segno.
op2:    dw      0x0003      ; Intero senza segno.
;
section .text
global _start
;-------------------------------------------------------
_start:
    movzx edx, word [op1]   ; Moltiplicando.
    movzx ecx, word [op2]   ; Moltiplicatore.
bp1:
    push  ebp     ; Salva il registro EBP prima
                  ; della chiamata.
    push  ecx     ; Inserisce il moltiplicando nella pila.
    push  edx     ; Inserisce il moltiplicatore nella pila.
    call  f_mol   ; Esegue la moltiplicazione: EAX = EDX * ECX
    pop   edx     ; Recupera il moltiplicando dalla pila.
    pop   ecx     ; Recupera il moltiplicatore dalla pila.
    pop   ebp     ; Recupera il registro EBP dopo
                  ; la chiamata.
bp2:
    mov   ebx,   eax ; Restituisce il valore del prodotto,
    mov   eax,   1   ; ammesso che sia abbastanza piccolo
    int   0x80       ; da poter essere rappresentato come
                     ; valore di uscita.
;-------------------------------------------------------
; Moltiplicazione di due numeri interi.
```

```
; f_mol (a, b) => EAX
; EAX = a * b
;
f_mol:
    mov   ebp,   esp      ; Copia ESP in EBP.
    mov   edx,   [ebp+4]  ; Copia il primo parametro in EDX.
    mov   ecx,   [ebp+8]  ; Copia il secondo parametro
                          ; in ECX.
    mov   eax,   0        ; Azzera EAX per sicurezza.
    ;
do_mol:
    cmp   ecx,   0        ; Se il moltiplicatore è pari a
    jz    end_do_mol      ; zero, il ciclo deve terminare.
    shr   ecx,   1        ; Fa scorrere a destra il
    jnc   end_do_somma    ; moltiplicatore e se l'indicatore
                          ; di riporto non è attivo, salta
                          ; la somma.
do_somma:
    add   eax,   edx      ; Aggiunge il moltiplicando al
                          ; risultato.
end_do_somma:
    shl   edx,   1        ; Fa scorrere il moltiplicando a
                          ; sinistra.
    jmp   do_mol          ; Torna all'inizio del ciclo di
                          ; moltiplicazione.
end_do_mol:
    mov   esp,   ebp      ; Ripristina ESP, espellendo
                          ; le variabili locali.
    ret                   ; Torna all'istruzione successiva
                          ; alla chiamata.
```

Nei listati appena mostrati si vede che ***EBP*** viene salvato prima della chiamata e ripristinato successivamente. Esiste però un altro modo, più diffuso, per cui il registro ***EBP*** va salvato nella pila all'inizio

della funzione, con le conseguenze che ciò comporta:

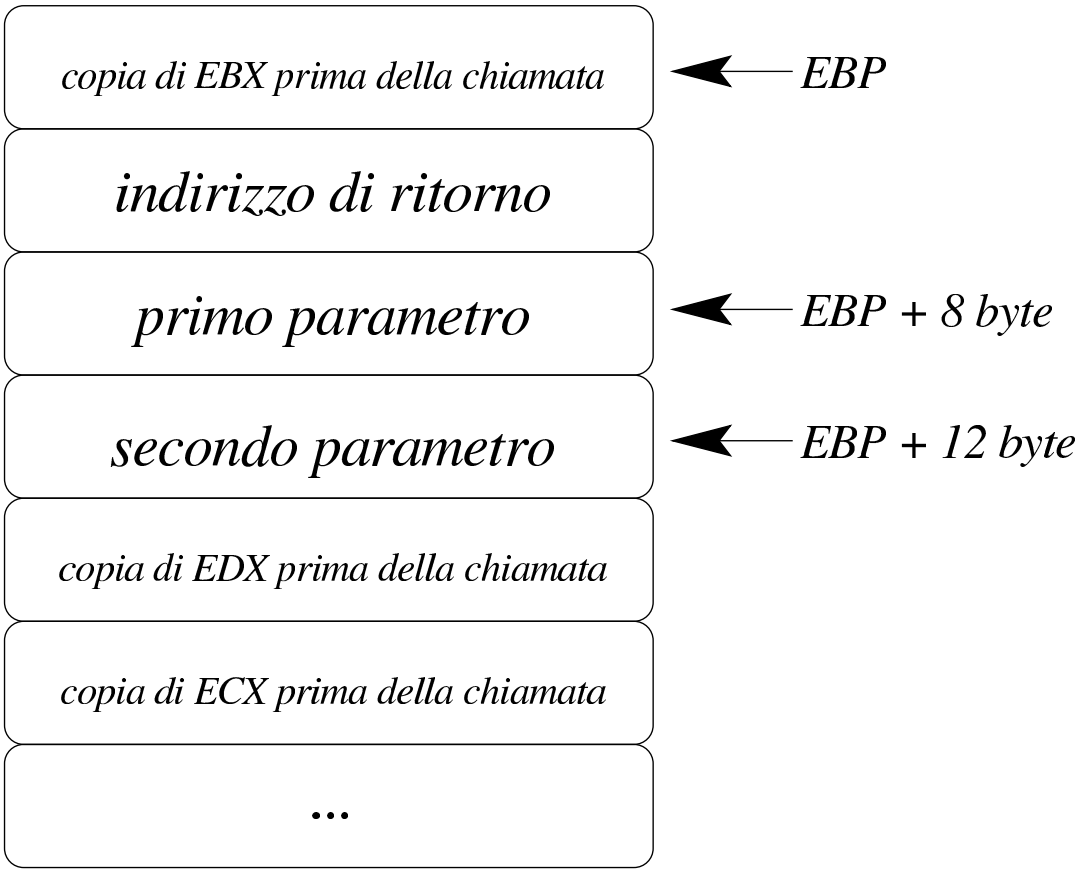
```
…
bp1:
    push  %ecx  # Inserisce il moltiplicando nella pila.
    push  %edx  # Inserisce il moltiplicatore nella pila.
    call  f_mol # Esegue la moltiplicazione: EAX = EDX * ECX
    pop   %edx  # Recupera il moltiplicando dalla pila.
    pop   %ecx  # Recupera il moltiplicatore dalla pila.
bp2:
…
f_mol:
    push   %ebp             # Salva il registro EBP.
bp3:
    mov   %esp,     %ebp    # Copia ESP in EBP.
    mov   8(%ebp),  %edx    # Copia il primo parametro in EDX.
    mov   12(%ebp), %ecx    # Copia il secondo parametro
                            # in ECX.
    mov   $0,       %eax    # Azzera EAX per sicurezza.
…
end_do_mol:
    mov   %ebp,     %esp    # Ripristina ESP, espellendo le
                            # variabili locali.
    pop   %ebp              # Riporta il registro EBP allo
                            # stato precedente.
    ret                     # Torna all'istruzione successiva
                            # alla chiamata.
```

```
…
bp1:
    push  ecx   ; Inserisce il moltiplicando nella pila.
    push  edx   ; Inserisce il moltiplicatore nella pila.
    call  f_mol ; Esegue la moltiplicazione: EAX = EDX * ECX
    pop   edx   ; Recupera il moltiplicando dalla pila.
    pop   ecx   ; Recupera il moltiplicatore dalla pila.
```

```
bp2:
…
f_mol:
    push  ebp                ; Salva il registro EBP.
bp3:
    mov   ebp,      esp      ; Copia ESP in EBP.
    mov   edx,      [ebp+8]  ; Copia il primo parametro
                             ; in EDX.
    mov   ecx,      [ebp+12] ; Copia il secondo parametro
                             ; in ECX.
    mov   eax,      0        ; Azzera EAX per sicurezza.
…
end_do_mol:
    mov   esp,      ebp      ; Ripristina ESP, espellendo le
                             ; variabili locali.
    pop   ebp                ; Ripristina il registro EBP.
    ret                      ; Torna all'istruzione successiva
                             ; alla chiamata.
```

In tal caso, in corrispondenza del simbolo '**bp3**', la pila ha i contenuti che sono schematizzati nella figura successiva.

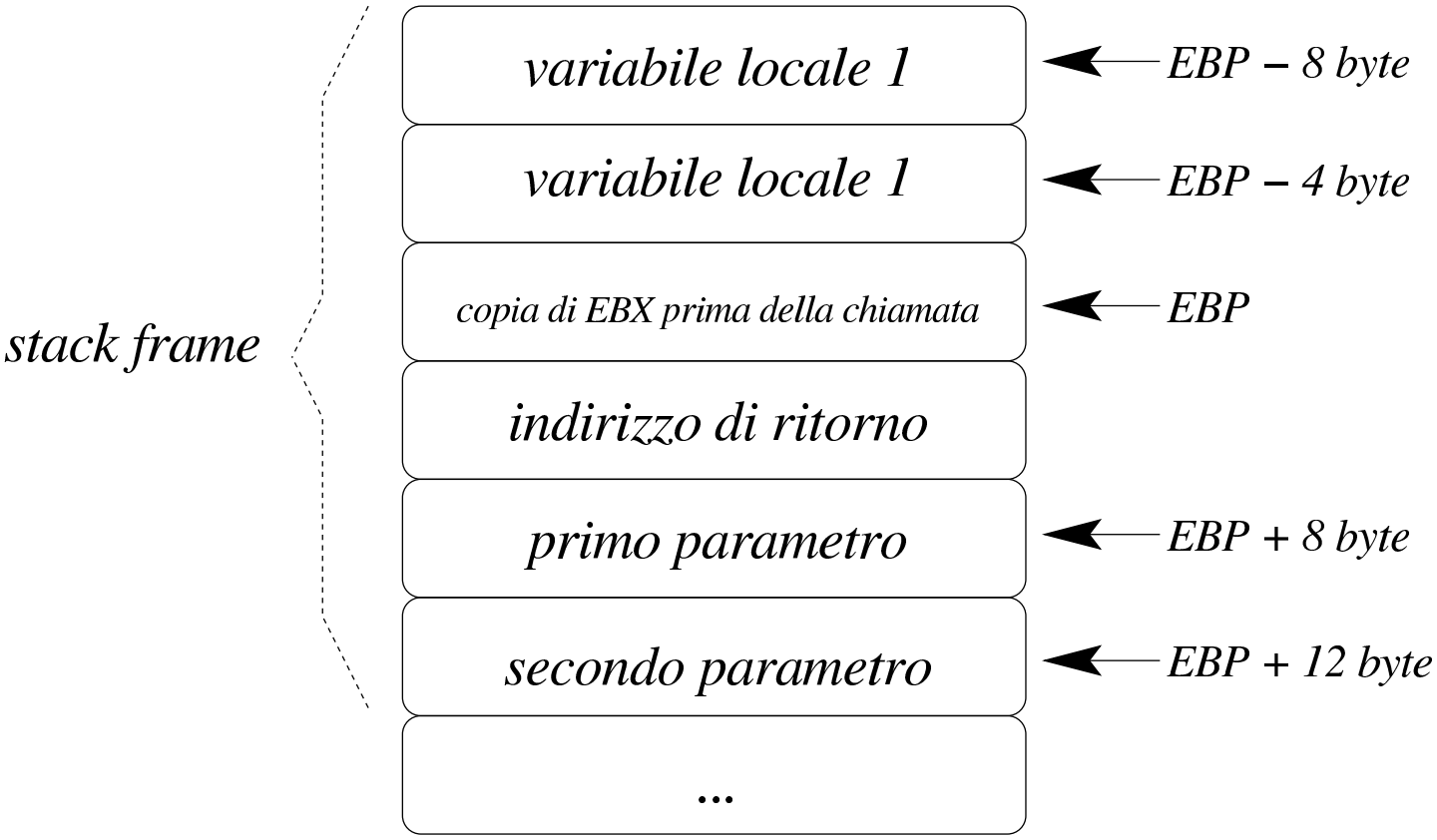
Figura 64.178. Situazione della pila in corrispondenza del simbolo '**bp3**'.



64.10.5 Allocazione dello spazio per le variabili locali e preservazione dei registri

Come accennato nella sezione precedente, una volta salvato il valore di ***EBP*** nella pila e assegnatovi il valore di ***ESP***, le variabili locali possono essere accumulate nella pila quando servono. Tuttavia, di solito si preferisce definire subito lo spazio utilizzato dalle variabili locali. Per esempio, supponendo di averne due, la pila potrebbe mostrarsi come nella figura successiva.

Figura 64.179. Situazione della pila, all'interno di una funzione con due variabili locali.



L'insieme degli elementi della pila, costituito dai parametri della funzione fino alle variabili locali, è noto come *stack frame*. Spesso, le «convenzioni di chiamata» prescrivono che siano le funzioni stesse a preservare lo stato precedente dei registri, pertanto, di solito, dopo la definizione dello spazio usato dalle variabili locali, si salvano nella pila anche tutti i registri principali, eventualmente con l'aiuto dell'istruzione '**PUSHA**'. I listati successivi mostrano una modifica ulteriore del programma già utilizzato nelle sezioni precedenti.

```
# op1 * op2
#
.section .data
op1:    .byte   0x07, 0x00      # little endian = 0x0005
                                # intero senza segno.
```

```
op2:    .byte   0x03, 0x00    # little endian = 0x0003
                              # intero senza segno.
#
.section .text
.globl _start
#-------------------------------------------------------
_start:
    movzx op1,  %edx # Moltiplicando.
    movzx op2,  %ecx # Moltiplicatore.
    mov   $0,   %eax # Risultato.
bp1:
    push  %ecx       # Inserisce il moltiplicando nella
                     # pila.
    push  %edx       # Inserisce il moltiplicatore nella
                     # pila.
    call  f_mol      # Esegue la moltiplicazione:
                     # EAX = EDX * ECX
    add   $8, %esp   # Espelle i parametri
                     # di chiamata.
bp2:
    mov   %eax, %ebx # Restituisce il valore del prodotto,
    mov   $1,   %eax # ammesso che sia abbastanza piccolo
    int   $0x80      # da poter essere rappresentato come
                     # valore di uscita.
#-------------------------------------------------------
# Moltiplicazione di due numeri interi.
# f_mol (a, b) => EAX
# EAX = a * b
#
f_mol:
    push %ebp           # Salva il registro EBP.
    mov  %esp,    %ebp  # Copia ESP in EBP.
    sub  $4,       %esp # Crea lo spazio per
                        # una variabile locale.
```

```
    pusha                   # Salva i registri
                            # principali.
    #
    mov  8(%ebp), %edx      # Copia il primo parametro in EDX.
    mov  12(%ebp), %ecx     # Copia il secondo parametro in ECX.
    mov  $0,      %eax      # Azzera EAX per sicurezza.
    #
do_mol:
    cmp  $0,      %ecx      # Se il moltiplicatore è pari a
    jz   end_do_mol         # zero, il ciclo deve terminare.
    shr  $1,      %ecx      # Fa scorrere a destra il
    jnc  end_do_somma       # moltiplicatore e se l'indicatore
                            # di riporto non è attivo, salta
                            # la somma.
do_somma:
    add  %edx,    %eax      # Aggiunge il moltiplicando al
                            # risultato.
end_do_somma:
    shl  $1,      %edx      # Fa scorrere il moltiplicando a
                            # sinistra.
    jmp  do_mol             # Torna all'inizio del ciclo di
                            # moltiplicazione.
end_do_mol:
    mov  %eax, -4(%ebp)     # Copia EAX nella variabile
                            # locale prevista.
    popa                    # Ripristina i registri principali.
    mov -4(%ebp), %eax      # Rimette a posto il valore di EAX
                            # che deve essere restituito.
    mov   %ebp,    %esp     # Ripristina ESP, espellendo le
                            # variabili locali.
    pop   %ebp              # Riporta il registro EBP allo
                            # stato precedente.
    ret                     # Torna all'istruzione successiva
                            # alla chiamata.
```

```
; op1 * op2
;
section .data
op1:    dw      0x0007      ; Intero senza segno.
op2:    dw      0x0003      ; Intero senza segno.
;
section .text
global _start
;-------------------------------------------------------
_start:
    movzx edx, word [op1]   ; Moltiplicando.
    movzx ecx, word [op2]   ; Moltiplicatore.
bp1:
    push  ecx               ; Inserisce il moltiplicando
                            ; nella pila.
    push  edx               ; Inserisce il moltiplicatore
                            ; nella pila.
    call  f_mol             ; Esegue la moltiplicazione:
                            ; EAX = EDX * ECX
    add esp,        8       ; Espelle i parametri
                            ; della chiamata.
bp2:
    mov   ebx,      eax     ; Restituisce il valore del
    mov   eax,      1       ; prodotto, ammesso che sia
    int   0x80              ; abbastanza piccolo da poter
                            ; essere rappresentato come
                            ; valore di uscita.
;-------------------------------------------------------
; Moltiplicazione di due numeri interi.
; f_mol (a, b) => EAX
; EAX = a * b
;
f_mol:
    push  ebp               ; Salva il registro EBP.
```

```
    mov   ebp,      esp       ; Copia ESP in EBP.
    sub   esp,      4         ; Crea lo spazio per una
                              ; variabile locale.
    pusha                     ; Salva i registri principali.
    ;
    mov   edx,      [ebp+8]   ; Copia il primo parametro
                              ; in EDX.
    mov   ecx,      [ebp+12]  ; Copia il secondo parametro in
                              ; ECX.
    mov   eax,      0         ; Azzera EAX per sicurezza.
    ;
do_mol:
    cmp   ecx,      0         ; Se il moltiplicatore è pari
    jz    end_do_mol          ; a zero, il ciclo deve
                              ; terminare.
    shr   ecx,      1         ; Fa scorrere a destra il
    jnc   end_do_somma        ; moltiplicatore e se
                              ; l'indicatore di riporto non
                              ; è attivo, salta la somma.
do_somma:
    add   eax,      edx       ; Aggiunge il moltiplicando al
                              ; risultato.
end_do_somma:
    shl   edx,      1         ; Fa scorrere il moltiplicando
                              ; a sinistra.
    jmp   do_mol              ; Torna all'inizio del ciclo
                              ; di moltiplicazione.
end_do_mol:
    mov   [ebp-4],  eax       ; Copia EAX nella variabile
                              ; locale prevista.
    popa                      ; Ripristina i registri
                              ; principali.
    mov   eax,      [ebp-4]   ; Rimette a posto il valore
                              ; di EAX che deve essere
```

```
                                ; restituito.
    mov   esp,      ebp         ; Ripristina ESP, espellendo
                                ; le variabili locali.
    pop   ebp                   ; Ripristina il registro EBP.
    ret                         ; Torna all'istruzione
                                ; successiva alla chiamata.
```

Come si vede, avendo usato la coppia di istruzioni '**PUSHA**', '**POPA**', alla fine occorre prendersi cura del risultato che è già disponibile nel registro ***EAX***: infatti viene salvato prima nello spazio riservato per la variabile locale, quindi vengono ripristinati tutti i registri (tutti eccetto '**ESP**') e ancora viene ripristinato ***EAX***, che deve trasmettere il valore alla chiamata.

A questo punto occorre sapere che le istruzioni seguenti possono essere sostituite dall'istruzione '**ENTER**', dove ***n*** rappresenta una quantità di byte:

```
    push  %ebp
    mov   %esp, %ebp
    sub   $n, %esp
```

```
    push  ebp
    mov   ebp, esp
    sub   esp, n
```

Per la precisione, il rimpiazzo avviene come nei due brani seguenti: si osservi che in questo caso, gli attributi di '**ENTER**' non vengono invertiti nelle due sintassi.

```
    enter $n, $0
```

```
    enter n, 0
```

Le istruzioni che invece va a rimpiazzare '**LEAVE**' sono quelle seguenti:

```
    mov   %ebp, %esp
    pop   %ebp
```

```
    mov   esp, ebp
    pop   ebp
```

Logicamente, '**LEAVE**' non richiede operandi, quindi si usa nello stesso modo nelle due sintassi:

```
    leave
```

64.10.6 Convenzioni di chiamata

«

Da quanto descritto si comprende che si possono usare diversi modi per chiamare una funzione, ma anche se esistono delle modalità equivalenti, occorre definire una convenzione. In generale, per chi scrive programmi in un sistema compatibile con la tradizione Unix, la cosa migliore è uniformarsi alle convenzioni di chiamata del linguaggio C (precisamente servono quelle usate dal proprio compilatore), in modo da poter mettere assieme programmi scritti in parte in linguaggio assemblatore e in parte anche in C. Di solito, le regole per chi scrive funzioni in linguaggio assemblatore sono sostanzialmente quelle dell'ultimo esempio mostrato nella sezione precedente:

- i parametri vanno messi sulla pila in ordine inverso, in modo tale che prima della chiamata appaia in cima il primo parametro;

- all'inizio della funzione, occorre accumulare nella pila il contenuto di ***EBP***, che deve essere ripristinato immediatamente prima del ritorno;
- all'interno della funzione si accede ai parametri contenuti nella pila, senza estrarli dalla stessa e senza modificarli, perché le modifiche non verrebbero considerate;
- i registri principali devono essere preservati (ogni compilatore ha la sua politica e non si può dire, in generale, quali siano);
- la funzione deve restituire il risultato della sua elaborazione attraverso ***EAX***, oppure, se si richiede una dimensione più grande, deve essere usata la coppia ***EDX***:***EAX***.[10]

Nel caso si vogliano utilizzare funzioni scritte in linguaggio C, all'interno di un programma scritto in linguaggio assemblatore, occorre verificare quali registri le funzioni scritte in C non preservano (oltre alla coppia ***EDX***:***EAX***, usata per restituire il risultato della chiamata). In generale, si può considerare che le funzioni scritte in linguaggio C potrebbero alterare i registri ***EAX***, ***ECX*** e ***EDX***. Se però si vuole avere la certezza assoluta sul contenuto dei registri dopo la chiamata di una funzione realizzata con un altro linguaggio, conviene organizzarsi salvando tutti quelli che si stanno utilizzando prima della chiamata e ripristinandoli subito dopo, come in parte è stato mostrato.

64.10.7 Nota sugli array «locali»

Generalmente, un array viene gestito attraverso uno spazio di memoria condiviso da tutto il programma, dove le funzioni che devono manipolarlo ricevono l'indirizzo di questo, tra i parametri della

chiamata. Tuttavia, nel caso si volesse gestire, all'interno di una funzione, un array locale, il cui contenuto viene abbandonato alla conclusione della stessa, l'unico modo per ottenere ciò è attraverso la pila dei dati. In pratica, come per le variabili locali scalari, andrebbe riservato un certo spazio aumentando la dimensione della pila in modo adeguato, per poi scandire tale spazio con indici appropriati.

64.11 Esempi di funzioni ricorsive

«

Vengono mostrati alcuni esempi molto semplici in cui si applica la ricorsione, adatti alla compilazione con GNU AS e NASM.

64.11.1 Elevamento a potenza

«

Viene proposta una soluzione ricorsiva del problema dell'elevamento a potenza. In pratica, x^y è pari a $x \cdot x^{(y-1)}$, ma in particolare: se x è pari a zero, il risultato è zero; altrimenti, se y è pari a zero, il risultato è uno.

All'interno della funzione '**f_pwr**' viene riservato lo spazio per una sola variabile locale, che serve a conservare il valore di *EAX*, per poi recuperarlo quando si ripristinano tutti i registri, prima della conclusione della funzione stessa.

```
# op1 ^ op2
#
.section .data
op1:    .int    3
op2:    .int    4
#
.section .text
.globl _start
#
# Main.
```

```
#
_start:
    mov   op1, %esi   # Base.
    mov   op2, %edi   # Esponente.
bp1:
    push  %edi        # f_pwr (ESI, EDI) ==> EAX
    push  %esi        #
    call  f_pwr       #
    add   $8, %esp    #
bp2:
    mov   %eax, %ebx  # Restituisce il valore della potenza,
    mov   $1, %eax    # ammesso che sia abbastanza piccolo
    int   $0x80       # da poter essere rappresentato come
                      # valore di uscita.
#
# Potenza di due numeri interi senza segno.
# f_pwr (a, b) ==> EAX
# EAX = a ^ b
#
f_pwr:
    enter $4, $0
    pusha
    #
    mov   8(%ebp), %esi  # Base.
    mov   12(%ebp), %edi # Esponente.
    #
    cmp   $0, %esi       # Se la base è pari a 0,
    jz    f_pwr_end_0    # restituisce 0.
    #
    cmp   $0, %edi       # Se l'esponente è pari a 0,
    jz    f_pwr_end_1    # restituisce 1.
    #
    dec   %edi           # Riduce l'esponente di una unità.
    push  %edi           # f_pwr (ESI, EDI) ==> EAX
```

```
    push  %esi            #
    call  f_pwr           #
    add   $8, %esp        #
    mul   %esi            # EDX:EAX = EAX*ESI
    mov   %eax, -4(%ebp)  # Salva il risultato.
    jmp   f_pwr_end_X     # Conclude la funzione.
    #
f_pwr_end_0:
    popa                  # Conclude la funzione con EAX = 0.
    mov $0, %eax          #
    leave                 #
    ret                   #
f_pwr_end_1:
    popa                  # Conclude la funzione con EAX = 1.
    mov $1, %eax          #
    leave                 #
    ret                   #
f_pwr_end_X:
    popa                  # Conclude la funzione con EAX pari
    mov -4(%ebp), %eax    # al valore salvato nella variabile
    leave                 # locale.
    ret                   #
```

```
; op1 ^ op2
;
section .data
op1:    dd      3
op2:    dd      4
;
section .text
global _start
;
; Main.
;
_start:
```

```
    mov   esi, [op1] ; Base.
    mov   edi, [op2] ; Esponente.
bp1:
    push  edi        ; f_pwr (ESI, EDI) ==> EAX
    push  esi        ;
    call  f_pwr      ;
    add   esp, 8     ;
bp2:
    mov   ebx, eax   ; Restituisce il valore della potenza,
    mov   eax, 1     ; ammesso che sia abbastanza piccolo
    int   0x80       ; da poter essere rappresentato come
                     ; valore di uscita.
;
; Potenza di due numeri interi senza segno.
; f_pwr (a, b) ==> EAX
; EAX = a ^ b
;
f_pwr:
    enter 4,0
    pusha
    ;
    mov   esi, [ebp+8]  ; Base.
    mov   edi, [ebp+12] ; Esponente.
    ;
    cmp   esi, 0        ; Se la base è pari a 0,
    jz    f_pwr_end_0   ; restituisce 0.
    ;
    cmp   edi, 0        ; Se l'esponente è pari a 0,
    jz    f_pwr_end_1   ; restituisce 1.
    ;
    dec   edi           ; Riduce l'esponente di una unità.
    push  edi           ; f_pwr (ESI, EDI) ==> EAX
    push  esi           ;
    call  f_pwr         ;
```

```
    add   esp, 8          ;
    mul   esi             ; EDX:EAX = EAX*ESI
    mov   [ebp-4], eax    ; Salva il risultato.
    jmp   f_pwr_end_X     ; Conclude la funzione.
    ;
f_pwr_end_0:
    popa                  ; Conclude la funzione con EAX = 0.
    mov eax, 0            ;
    leave                 ;
    ret                   ;
f_pwr_end_1:
    popa                  ; Conclude la funzione con EAX = 1.
    mov eax, 1            ;
    leave                 ;
    ret                   ;
f_pwr_end_X:
    popa                  ; Conclude la funzione con EAX pari
    mov eax, [ebp-4]      ; al valore salvato nella variabile
    leave                 ; locale.
    ret                   ;
```

64.11.2 Fattoriale

«

Viene proposta una soluzione ricorsiva del problema del fattoriale. In pratica, $x!$ è pari a $x \cdot (x-1)!$, ma in particolare: se x è pari a 1, il risultato è uno.

All'interno della funzione '**f_fact**' viene riservato lo spazio per una sola variabile locale, che serve a conservare il valore di ***EAX***, per poi recuperarlo quando si ripristinano tutti i registri, prima della conclusione della funzione stessa.

```
# op1!
#
```

```
.section .data
op1:    .int    5
#
.section .text
.globl _start
#
# Main.
#
_start:
    mov   op1, %esi  # ESI contiene il valore di cui si
                     # vuole calcolare il fattoriale.
bp1:
    push  %esi       # f_fact (ESI) ==> EAX
    call  f_fact     #
    add   $4, %esp   #
bp2:
    mov   %eax, %ebx # Restituisce il valore del fattoriale,
    mov   $1, %eax   # ammesso che sia abbastanza piccolo
    int   $0x80      # da poter essere rappresentato come
                     # valore di uscita.
#
# Fattoriale di un numero senza segno.
# f_fatt (a) ==> EAX
# EAX = a!
#
f_fact:
    enter $4, $0
    pusha
    #
    mov   8(%ebp), %edi  # Valore di cui calcolare il
                         # fattoriale.
    cmp   $1, %edi       # Il fattoriale di 1 è 1.
    jz    f_fact_end_1   #
    #
```

```
    mov   %edi, %esi      # ESI contiene il valore di cui si
    dec   %esi            # vuole il fattoriale, ridotto di
                          # una unità.
    push  %esi            # f_fact (ESI) ==> EAX
    call  f_fact          #
    add   $4, %esp        #
    mul   %edi            # EDX:EAX = EAX*EDI
    mov   %eax, -4(%ebp)  # Salva il risultato.
    jmp   f_fact_end_X    # Conclude la funzione.
    #
f_fact_end_1:
    popa                  # Conclude la funzione con EAX = 1.
    mov $1, %eax          #
    leave                 #
    ret                   #
f_fact_end_X:
    popa                  # Conclude la funzione con EAX pari
    mov -4(%ebp), %eax    # al valore salvato nella variabile
    leave                 # locale.
    ret                   #
```

```
; op1!
;
section .data
op1:    dd      5
;
section .text
global _start
;
; Main.
;
_start:
    mov   esi, [op1] ; ESI contiene il valore di cui si
                     ; vuole calcolare il fattoriale.
bp1:
```

```
    push  esi          ; f_fact (ESI) ==> EAX
    call  f_fact       ;
    add   esp, 4       ;
bp2:
    mov   ebx, eax     ; Restituisce il valore del fattoriale,
    mov   eax, 1       ; ammesso che sia abbastanza piccolo
    int   0x80         ; da poter essere rappresentato come
                       ; valore di uscita.
;
; Fattoriale di un numero senza segno.
; f_fatt (a) ==> EAX
; EAX = a!
;
f_fact:
    enter 4,0
    pusha
    ;
    mov   edi, [ebp+8] ; Valore di cui calcolare il
                       ; fattoriale.
    cmp   edi, 1       ; Il fattoriale di 1 è 1.
    jz    f_fact_end_1 ;
    ;
    mov   esi, edi     ; ESI contiene il valore di cui si
    dec   esi          ; vuole il fattoriale, ridotto di
                       ; una unità.
    push  esi          ; f_fact (ESI) ==> EAX
    call  f_fact       ;
    add   esp, 4       ;
    mul   edi          ; EDX:EAX = EAX*EDI
    mov   [ebp-4], eax ; Salva il risultato.
    jmp   f_fact_end_X ; Conclude la funzione.
    ;
f_fact_end_1:
    popa               ; Conclude la funzione con EAX = 1.
```

```
    mov eax, 1          ;
    leave               ;
    ret                 ;
f_fact_end_X:
    popa                ; Conclude la funzione con EAX pari
    mov eax, [ebp-4]    ; al valore salvato nella variabile
    leave               ; locale.
    ret                 ;
```

64.12 Indirizzamento dei dati

«

In generale, con l'architettura x86 ci si preoccupa di definire il modo in cui fare riferimento ai dati nel sorgente in linguaggio assemblatore (incluso ciò che riguarda la pila), mentre per il riferimento alle istruzioni si usano simboli che il compilatore traduce normalmente in indirizzi relativi (il «dislocamento»). Viene mostrata una tabella che riepiloga i vari modi con cui è possibile fare riferimento ai dati, secondo le due sintassi più comuni.

Tabella 64.193. Indirizzamento dei dati.

Esempio AT&T	Esempio Intel	Descrizione
`$21`	`21`	Si sta facendo riferimento al numero 21_{10} in modo letterale.
`$0x15`	`0x15`	
`$025`	`025`	
`$0b10101`	`10101b`	

Esempio AT&T	Esempio Intel	Descrizione
`$`***nome_simbolo***	***nome_simbolo***	Si fa riferimento all'indirizzo corrispondente al simbolo, ovvero al numero che costituisce tale indirizzo.
nome_simbolo	`[`***nome_simbolo***`]`	Si fa riferimento all'area di memoria che inizia in corrispondenza del simbolo.
nome_simbolo $\pm$***n***	`[`***nome_simbolo*** $\pm$***n***`]`	Si fa riferimento all'area di memoria che inizia in corrispondenza del simbolo, $\pm$***n*** byte.
`%eax`	`eax`	Si sta facendo riferimento al registro ***EAX***, in qualità di variabile.
`(%eax)`	`[eax]`	Si sta facendo riferimento all'area di memoria che inizia a partire dall'indirizzo contenuto nel registro ***EAX***
$\pm$***n*** `(%eax)`	`[eax]`$\pm$***n***	Si sta facendo riferimento all'area di memoria che inizia a partire dall'indirizzo contenuto nel registro ***EAX***, $\pm$***n*** byte.

Esempio AT&T	Esempio Intel	Descrizione
nome_simbolo `(%eax)`	`[eax+`***nome_simbolo***`]`	Si sta facendo riferimento all'area di memoria che inizia a partire dall'indirizzo a cui fa riferimento il simbolo, più il contenuto del registro ***EAX***.
`(`***nome_simbolo*** $\pm$***n***`)(%eax)`	`[eax+`***nome_simbolo*** $\pm$***n***`]`	Si sta facendo riferimento all'area di memoria che inizia a partire dall'indirizzo a cui fa riferimento il simbolo, più il contenuto del registro ***EAX***, $\pm$***n*** byte.
`(%eax,%ebx,`***n***`)`	`[eax+%ebx*`***n***`]`	Si sta facendo riferimento all'area di memoria che inizia a partire da ***EAX***+(***EBX***$\cdot$***n***).
$\pm$***m*** `(%eax,%ebx,`***n***`)`	`[eax+%ebx*`***n*** $\pm$***m***`]`	Si sta facendo riferimento all'area di memoria che inizia a partire da (***EAX***+(***EBX***$\cdot$***n***))$\pm$***m***.

64.12.1 Gestione di array

«

Per comprendere l'uso della forma più complessa di indirizzamento, si prenda l'esempio seguente, in cui appare un simbolo, denominato '**`record`**', che descrive una sequenza di valori, contenenti anche ciò che si vuole considerare una matrice di numeri.

```
#
.section .data
```

```
record:           .ascii "matrice"
                  .int    0,  1,  2,  3,  4,  5,  6,  7,  8,  9
                  .int   10, 11, 12, 13, 14, 15, 16, 17, 18, 19
                  .int   20, 21, 22, 23, 24, 25, 26, 27, 28, 29
                  .int   30, 31, 32, 33, 34, 35, 36, 37, 38, 39
                  .int   40, 41, 42, 43, 44, 45, 46, 47, 48, 49
                  .int   50, 51, 52, 53, 54, 55, 56, 57, 58, 59
                  .int   60, 61, 62, 63, 64, 65, 66, 67, 68, 69
                  .int   70, 71, 72, 73, 74, 75, 76, 77, 78, 79
                  .int   80, 81, 82, 83, 84, 85, 86, 87, 88, 89
                  .int   90, 91, 92, 93, 94, 95, 96, 97, 98, 99
dimensione_int:   .int    4       # 32 bit
dimensione_riga: .int    40       # 4 * 10 byte
riga:             .int    3       # da 0 a 9
colonna:          .int    5       # da 0 a 9
#
.section .text
.globl _start
#
_start:
    mov   riga, %eax        # Legge le coordinate di riga e
    mov   colonna, %ecx     # colonna, copiandole in EAX e ECX.
    mull  dimensione_riga   # EDX:EAX := EAX * 40.
    mov   (record+7)(%eax,%ecx,4), %ebx
                            # EBX := array[riga,colonna].
bp1:
    mov   $1, %eax          # Restituisce il valore trovato
    int   $0x80             # nella matrice.
```

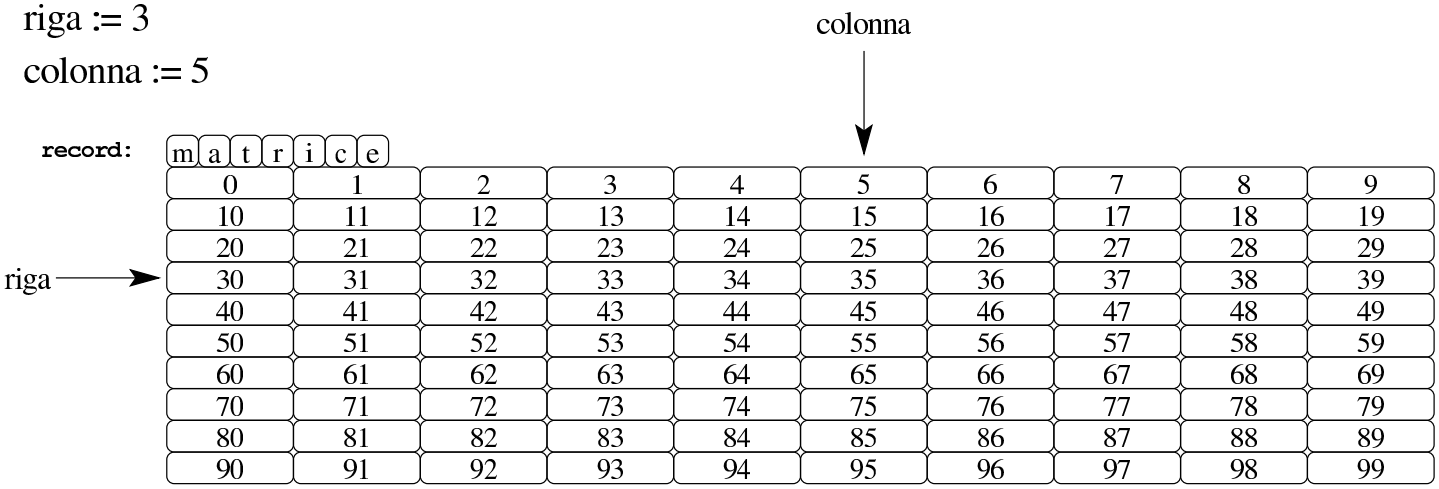
```
;
section .data
record:           db     "matrice"
                  dd      0,  1,  2,  3,  4,  5,  6,  7,  8,  9
                  dd     10, 11, 12, 13, 14, 15, 16, 17, 18, 19
                  dd     20, 21, 22, 23, 24, 25, 26, 27, 28, 29
```

```
                dd      30, 31, 32, 33, 34, 35, 36, 37, 38, 39
                dd      40, 41, 42, 43, 44, 45, 46, 47, 48, 49
                dd      50, 51, 52, 53, 54, 55, 56, 57, 58, 59
                dd      60, 61, 62, 63, 64, 65, 66, 67, 68, 69
                dd      70, 71, 72, 73, 74, 75, 76, 77, 78, 79
                dd      80, 81, 82, 83, 84, 85, 86, 87, 88, 89
                dd      90, 91, 92, 93, 94, 95, 96, 97, 98, 99
dimensione_int:  dd       4       ; 32 bit
dimensione_riga: dd      40       ; 4 * 10 byte
riga:            dd       3       ; da 0 a 9
colonna:         dd       5       ; da 0 a 9
;
section .text
global _start
;
_start:
    mov   eax, [riga]             ; Legge le coordinate di riga
    mov   ecx, [colonna]          ; e colonna, copiandole in
                                  ; EAX e ECX.
    mul   long [dimensione_riga] ; EDX:EAX := EAX * 40.
    mov   ebx, [record+7+eax+ecx*4]
                                  ; EBX := array[riga,colonna].
bp1:
    mov   eax, 1                  ; Restituisce il valore
    int   0x80                    ; trovato nella matrice.
```

In pratica, '**record**' individua l'inizio di una struttura composta da una stringa di sette byte, seguita da un centinaio di interi da 32 bit, suddivisi idealmente in righe da 10 unità. In pratica, una «riga» di questi numeri occupa 40 byte.

Per accedere a un certo elemento della matrice contenuta nel «record», occorre considerare uno scostamento iniziale di 7 byte, quindi occorre calcolare la posizione dell'elemento, secondo lo schema

che si vede nella figura successiva.



0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

$$7 + (\text{riga} \times 40) + (\text{colonna} \times 4)$$

Viene proposto un altro esempio equivalente, ma realizzato gestendo in modo diverso l'indirizzamento. La dichiarazione dei dati è la stessa:

```
...
_start:
    mov   riga, %eax          # Legge le coordinate di riga.
    mull  dimensione_riga     # EDX:EAX := EAX * 40.
    mov   %eax, %ecx          # ECX := riga * 40.
    #
    mov   colonna, %eax       # Legge le coordinate di colonna.
    mull  dimensione_int      # EDX:EAX := EAX * 4.
    add   %eax, %ecx          # ECX := (riga*40)+(colonna*4)
    #
    add   $7, %ecx            # ECX := 7+(riga*40)+(colonna*4)
    mov   record(%ecx), %ebx  # EBX := array[riga,colonna].
bp1:
    mov   $1, %eax            # Restituisce il valore trovato
    int   $0x80               # nella matrice.
```

```
…
_start:
    mov eax, [riga]             ; Legge le coordinate di riga.
    mul long [dimensione_riga] ; EDX:EAX := EAX * 40.
    mov ecx, eax                ; ECX := riga * 40.
    ;
    mov eax, [colonna]          ; Legge le coord. di colonna.
    mul long [dimensione_int]  ; EDX:EAX := EAX * 4.
    add ecx, eax                ; ECX := (riga*40)+(colonna*4)
    ;
    add ecx, 7                  ; ECX := 7+(riga*40)+(colonna*4)
    mov ebx, [record+ecx]       ; EBX := array[riga,colonna].
bp1:
    mov eax, 1                  ; Restituisce il valore trovato
    int 0x80                    ; nella matrice.
```

Come si vede, si arriva a mettere nel registro ***ECX*** il risultato di 7+(***riga***×40)+(***colonna***×4); pertanto, alla fine si aggiunge solo l'indirizzo iniziale della struttura e si ottiene, complessivamente, l'indirizzo del valore cercato nella matrice.

Un altro esempio, dove si vede un altro modo di usare l'indirizzamento indiretto fornito dal linguaggio macchina:

```
…
_start:
    mov   riga, %eax          # Legge le coordinate di riga.
    mull  dimensione_riga     # EDX:EAX := EAX * 40.
    mov   %eax, %ecx          # ECX := riga * 40.
    #
    mov   colonna, %eax       # Legge le coordinate di colonna.
    mull  dimensione_int      # EDX:EAX := EAX * 4.
    add   %eax, %ecx          # ECX := (riga*40)+(colonna*4)
    #
    add   $7, %ecx            # ECX := 7+(riga*40)+(colonna*4)
```

```
    mov   $record, %edx      # EDX := indirizzo iniziale di
                             # "record".
    mov   (%edx,%ecx), %ebx  # EBX := array[riga,colonna].
bp1:
    mov   $1, %eax           # Restituisce il valore trovato
    int   $0x80              # nella matrice.
```

```
…
_start:
    mov eax, [riga]             ; Legge le coordinate di riga.
    mul long [dimensione_riga] ; EDX:EAX := EAX * 40.
    mov ecx, eax                ; ECX := riga * 40.
    ;
    mov eax, [colonna]          ; Legge le coordinate di colonna.
    mul long [dimensione_int]  ; EDX:EAX := EAX * 4.
    add ecx, eax                ; ECX := (riga*40)+(colonna*4)
    ;
    add ecx, 7          ; ECX := 7+(riga*40)+(colonna*4)
    mov edx, record     ; EDX := indirizzo iniziale di "record".
    mov ebx, [edx+ecx] ; EBX := array[riga,colonna].
bp1:
    mov eax, 1                  ; Restituisce il valore trovato
    int 0x80                    ; nella matrice.
```

Infine, un esempio dove si fa in modo che ***ECX*** contenga l'indirizzo completo dell'elemento cercato:

```
…
_start:
    mov   riga, %eax       # Legge le coordinate di riga.
    mull  dimensione_riga # EDX:EAX := EAX * 40.
    mov   %eax, %ecx       # ECX := riga * 40.
    #
    mov   colonna, %eax    # Legge le coordinate di colonna.
    mull  dimensione_int   # EDX:EAX := EAX * 4.
```

```
    add   %eax, %ecx       # ECX := (riga*40)+(colonna*4)
    #
    add   $7, %ecx         # ECX := 7+(riga*40)+(colonna*4)
    add   $record, %ecx    # ECX := indirizzo completo.
    mov   (%ecx), %ebx     # EBX := array[riga,colonna].
bp1:
    mov   $1, %eax         # Restituisce il valore trovato
    int   $0x80            # nella matrice.
```

```
…
_start:
    mov eax, [riga]             ; Legge le coordinate di riga.
    mul long [dimensione_riga] ; EDX:EAX := EAX * 40.
    mov ecx, eax                ; ECX := riga * 40.
    ;
    mov eax, [colonna]          ; Legge le coord. di colonna.
    mul long [dimensione_int]  ; EDX:EAX := EAX * 4.
    add ecx, eax                ; ECX := (riga*40)+(colonna*4)
    ;
    add ecx, 7          ; ECX := 7+(riga*40)+(colonna*4)
    add ecx, record     ; ECX := indirizzo completo.
    mov ebx, [ecx]      ; EBX := array[riga,colonna].
bp1:
    mov eax, 1                ; Restituisce il valore trovato
    int 0x80                  ; nella matrice.
```

64.12.2 Istruzione «LEA»

«

L'istruzione '**LEA**' consente di calcolare un indirizzo di memoria e di salvarlo in un registro. Per esempio, le due istruzioni successive, una con '**MOV**' e l'altra con '**LEA**', fanno la stessa cosa:

```
    mov   (%ecx), %ebx
    lea   %ecx, %ebx
```

Ovvero:

```
    mov   ebx, [ecx]
    lea   ebx, ecx
```

Pertanto, con l'istruzione '**LEA**', ciò che appare nell'operando che svolge il ruolo di «origine» viene considerato solo per il suo indirizzo e copiato nella destinazione. La differenza sostanziale, rispetto a '**MOV**', sta nel poter usare le espressioni di indirizzamento indiretto. Per riprendere l'esempio usato precedentemente per mostrare come accedere a una matrice di numeri, si potrebbe fare una cosa come si vede nel listato successivo, anche se sarebbe poco utile in tale circostanza particolare:

```
…
_start:
    mov  riga, %eax       # Legge le coordinate di riga e
    mov  colonna, %ecx    # colonna, copiandole in EAX e ECX.
    #
    mull dimensione_riga # EDX:EAX := EAX * 40.
    #
    lea  (record+7)(%eax,%ecx,4), %edx # EDX contiene
                                       # l'indirizzo
                                       # dell'elemento.
    mov  (%edx), %ebx          # EBX := array[riga,colonna].
bp1:
    mov  $1, %eax              # Restituisce il valore trovato
    int  $0x80                 # nella matrice.
```

```
…
_start:
    mov   eax, [riga]    ; Legge le coordinate di riga e
    mov   ecx, [colonna] ; colonna, copiandole in EAX e ECX.
    ;
```

```
    mul   long [dimensione_riga]  ; EDX:EAX := EAX * 40.
    ;
    lea   edx, [record+7+eax+ecx*4] ; EDX contiene
                                    ; l'indirizzo
                                    ; dell'elemento.
    mov   ebx, [edx]              ; EBX := array[riga,colonna].
bp1:
    mov   eax, 1                  ; Restituisce il valore trovato
    int   0x80                    ; nella matrice.
```

64.13 Rappresentazione dei dati in memoria attraverso un esempio

«

Viene proposto un esempio in cui si utilizzano i tipi principali di variabili inizializzate in memoria, con due listati equivalenti, il primo adatto a GNU AS mentre il secondo è per NASM. In particolare, nel secondo listato, non potendo dichiarare variabili da 64 bit, sono state usate coppie di variabili a 32 bit, inizializzate tenendo conto dell'inversione *little endian*.

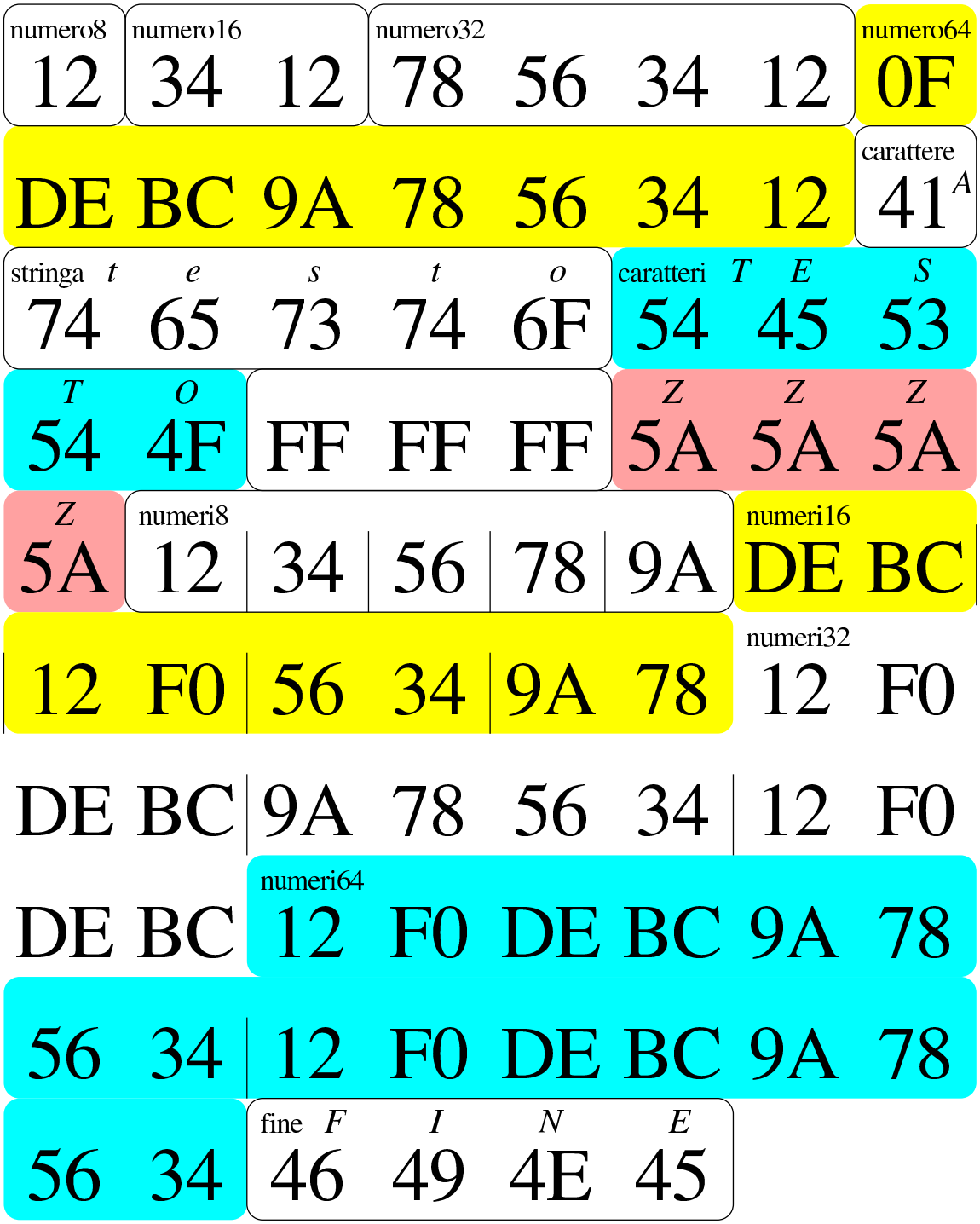
```
#
.section .data
numero8:    .byte   0b00010010            # 0x12        # 18
numero16:   .short  0b0001001000110100    # 0x1234      # 4660
numero32:   .int    0b00010010001101000101011001111000
                                          # 0x12345678
                                          # 305419896
numero64:   .quad   0x123456789ABCDEF0
carattere:  .byte   'A'
stringa:    .ascii  "testo"
caratteri:  .byte   'T', 'E', 'S', 'T', 'O'
            .skip 3, 0xFF
            .rept 4
```

```
            .byte   'Z'
            .endr
numeri8:    .byte   0x12, 0x34, 0x56, 0x78, 0x9A
numeri16:   .short  0xBCDE, 0xF012, 0x3456, 0x789A
numeri32:   .int    0xBCDEF012, 0x3456789A, 0xBCDEF012
numeri64:   .quad   0x3456789ABCDEF012, 0x3456789ABCDEF012
fine:       .byte   'F', 'I', 'N', 'E'
#
.section .text
.globl _start
#
_start:
    mov   $0, %ebx
    mov   $1, %eax
    int   $0x80
```

```
;
section .data
numero8:    db   00010010b              ; 0x12       = 18
numero16:   dw   0001001000110100b      ; 0x1234     = 4660
numero32:   dd   00010010001101000101011001111000b
                                        ; 0x12345678
                                        ;  = 305419896
numero64:   dd   0x9ABCDEF0, 0x12345678 ; 0x123456789ABCDEF0
carattere:  db   'A'
stringa:    db   "testo"
caratteri:  db   'T', 'E', 'S', 'T', 'O'
times 3     db   0xFF
times 4     db   'Z'
;
;
numeri8:    db   0x12, 0x34, 0x56, 0x78, 0x9A
numeri16:   dw   0xBCDE, 0xF012, 0x3456, 0x789A
numeri32:   dd   0xBCDEF012, 0x3456789A, 0xBCDEF012
```

```
numeri64:   dd   0xBCDEF012, 0x3456789A, 0xBCDEF012,
                 0x3456789A
fine:       db   'F', 'I', 'N', 'E'
;
section .text
global _start
;
_start:
    mov   ebx, 0
    mov   eax, 1
    int   0x80
```

Il programma in questione non fa alcunché e serve solo per verificare con GDB come sono rappresentati i dati in memoria e come questi si possono leggere. Complessivamente, le variabili occupano 78 byte, secondo lo schema che si vede nella figura successiva, dove vengono riprodotte anche le inversioni dovute alla rappresentazione *little endian*:



Per leggere la memoria già inizializzata con GDB non è necessario avviare il programma, in quanto, appena aperto, questa è subito accessibile:

`$` **`gdb`** ***nome_programma*** [*Invio*]

```
(gdb) print /x (char[78])numero8[Invio]

  $1 = {0x12, 0x34, 0x12, 0x78, 0x56, 0x34, 0x12, 0xf0, 0xde,
    0xbc, 0x9a, 0x78, 0x56, 0x34, 0x12, 0x41, 0x74, 0x65,
    0x73, 0x74, 0x6f, 0x54, 0x45, 0x53, 0x54, 0x4f, 0xff,
    0xff, 0xff, 0x5a, 0x5a, 0x5a, 0x5a, 0x12, 0x34, 0x56,
    0x78, 0x9a, 0xde, 0xbc, 0x12, 0xf0, 0x56, 0x34, 0x9a,
    0x78, 0x12, 0xf0, 0xde, 0xbc, 0x9a, 0x78, 0x56, 0x34,
    0x12, 0xf0, 0xde, 0xbc, 0x12, 0xf0, 0xde, 0xbc, 0x9a,
    0x78, 0x56, 0x34, 0x12, 0xf0, 0xde, 0xbc, 0x9a, 0x78,
    0x56, 0x34, 0x46, 0x49, 0x4e, 0x45}
```

Quello che viene richiesto con il comando appena mostrato è di mostrare la memoria a partire dall'indirizzo corrispondente al simbolo '**numero8**' (pertanto dall'inizio), in forma di array di byte, composto da 78 elementi. Si può verificare la corrispondenza tra quanto ottenuto e la figura mostrata in precedenza.

Con i comandi successivi si ispezionano le variabili, singolarmente. È sempre necessario dichiarare la dimensione a cui si è interessati, quando questa è diversa da quella predefinita (32 bit). Non è possibile ispezionare le variabili da 64 bit in modo complessivo, pertanto occorre arrangiarsi, come se fossero array di due numeri a 32 bit. Nei comandi mostrati appare spesso l'opzione '***/f***', con la quale si dichiara il tipo di rappresentazione che si vuole ottenere.

```
(gdb) print /x (char)numero8[Invio]

  $2 = 0x12

(gdb) print (char)numero8[Invio]

  $3 = 18 '\022'

(gdb) print /x (short)numero16[Invio]
```

```
  $4 = 0x1234
(gdb) print (short)numero16[Invio]

  $5 = 4660
(gdb) print /x (int)numero32[Invio]

  $6 = 0x12345678
(gdb) print (int)numero32[Invio]

  $7 = 305419896
(gdb) print /x (int[2])numero64[Invio]

  $8 = {0x9abcdef0, 0x12345678}
(gdb) print (char)carattere[Invio]

  $9 = 65 'A'
(gdb) print (char[11])carattere[Invio]

  $10 = "AtestoTESTO"
(gdb) print (char[5])stringa[Invio]

  $11 = "testo"
(gdb) print (char[5])caratteri[Invio]

  $12 = "TESTO"
(gdb) print /x (char[12])caratteri[Invio]

  $13 = {0x54, 0x45, 0x53, 0x54, 0x4f, 0xff, 0xff, 0xff, ↩
 ↪0x5a, 0x5a, 0x5a, 0x5a}
```

```
(gdb) print /x (char[5])numeri8[Invio]

  $14 = {0x12, 0x34, 0x56, 0x78, 0x9a}
(gdb) print /d (char[5])numeri8[Invio]

  $15 = {18, 52, 86, 120, -102}
(gdb) print /x (short[4])numeri16[Invio]

  $16 = {0xbcde, 0xf012, 0x3456, 0x789a}
(gdb) print /d (short[4])numeri16[Invio]

  $17 = {-17186, -4078, 13398, 30874}
(gdb) print /d (unsigned short[4])numeri16[Invio]

  $18 = {48350, 61458, 13398, 30874}
(gdb) print /x (int[3])numeri32[Invio]

  $19 = {0xbcdef012, 0x3456789a, 0xbcdef012}
(gdb) print /d (int[3])numeri32[Invio]

  $20 = {-1126240238, 878082202, -1126240238}
(gdb) print /d (unsigned int[3])numeri32[Invio]

  $21 = {3168727058, 878082202, 3168727058}
(gdb) print /x (int[4])numeri64[Invio]

  $22 = {0xbcdef012, 0x3456789a, 0xbcdef012, 0x3456789a}
(gdb) print (char[4])fine[Invio]

  $23 = "FINE"
```

64.14 Esempi con gli array

Vengono mostrati alcuni esempi elementari di scansione di array, adatti alla compilazione con GNU AS e NASM. Si dà per scontato che si sappia utilizzare GDB per ispezionare la memoria e leggere, in particolare, il contenuto degli array stessi.

Negli esempi viene usata la direttiva '**.equ**', o '**equ**', per associare una sigla al livello in cui si trovano i dati nella pila (più precisamente nello *stack frame*).

Tutti gli esempi sono mostrati con listati a coppie: uno valido per GNU AS e l'altro per NASM.

64.14.1 Ricerca sequenziale

Viene mostrato un esempio di programma contenente una funzione che esegue una ricerca sequenziale all'interno di un array di interi, senza segno. Il metodo utilizzato si rifà a quanto descritto in pseudo-codifica nella sezione 62.4.1. Il risultato della scansione viene emesso attraverso il valore restituito dal programma; ciò che si ottiene è precisamente l'indice dell'elemento trovato, oppure -1 se nessun elemento corrisponde.

```
# Ricerca sequenziale.
#
.section .data
lista: .int 1, 4, 3, 7, 9, 10, 22, 44, 11, 23 # Interi senza
                                               # segno.
a:     .int 0            # Indice minimo.
z:     .int 9            # Indice massimo.
#
.section .text
.globl _start
```

```
#
# Main.
#
_start:
    push  z           # f_rs ($lista, $7, a, z) ==> EAX
    push  a           # Si cerca il valore 7 nell'array
    push  $7          # «lista», tra gli indici «a» e «z».
    push  $lista      # Viene restituito l'indice
    call  f_rs        # dell'elemento trovato, oppure
    add   $16, %esp   # -1 se non è presente.
bp1:
    mov   %eax, %ebx # Restituisce l'indice trovato,
    mov   $1, %eax   # ammesso che sia abbastanza piccolo
    int   $0x80      # da poter essere rappresentato come
                     # valore di uscita.
#
# Ricerca sequenziale all'interno di una lista di valori.
# f_rs (lista, x, a, z) ==> EAX
# Al termine EAX contiene l'indice del valore trovato,
# oppure -1 se questo non c'è.
#
f_rs:
    enter $4, $0
    pusha
    .equ  rs_i,     -4            # Gli si associa EAX.
    .equ  rs_lista,  8            # Gli si associa ESI.
    .equ  rs_x,     12            # Gli si associa EDX.
    .equ  rs_a,     16
    .equ  rs_z,     20
    #
    mov   rs_lista(%ebp), %esi    # ESI contiene l'indirizzo
                                  # dell'array.
    mov   rs_x(%ebp),     %edx    # EDX contiene il valore
                                  # cercato.
```

```
    #
    mov   rs_a(%ebp),     %eax   # EAX viene usato come
                                 # indice di scansione.
f_rs_loop:
    cmp   rs_z(%ebp),     %eax   # Se EAX è maggiore
                                 # dell'indice massimo,
    ja    f_rs_non_trovato       # l'elemento cercato non
                                 # c'è.
    #
    cmp   (%esi,%eax,4),  %edx   # Se il valore cercato
    je    f_rs_trovato           # corrisponde a quello
                                 # dell'indice corrente,
                                 # termina la scansione.
    #
    inc   %eax                   # Incrementa l'indice
    jmp   f_rs_loop              # di scansione e salta
                                 # all'inizio del ciclo.
    #
f_rs_non_trovato:
    popa                    # Conclude la funzione con
    mov $-1, %eax           # EAX = -1.
    leave                   #
    ret                     #
f_rs_trovato:
    mov %eax, rs_i(%ebp)    # Salva EAX nella variabile
                            # locale prevista.
    popa                    # Conclude la funzione con EAX
    mov rs_i(%ebp), %eax    # pari al valore salvato nella
    leave                   # variabile locale.
    ret                     #
```

```
; Ricerca sequenziale.
;
section .data
```

```
lista: dd 1, 4, 3, 7, 9, 10, 22, 44, 11, 23 ; Interi senza
                                            ; segno.
a:     dd 0              ; Indice minimo.
z:     dd 9              ; Indice massimo.
;
section .text
global _start
;
; Main.
;
_start:
    push  long [z] ; f_rs ($lista, $7, a, z) ==> EAX
    push  long [a] ; Si cerca il valore 7 nell'array
    push  long 7   ; «lista», tra gli indici «a» e «z».
    push  lista    ; Viene restituito l'indice dell'elemento
    call  f_rs     ; trovato, oppure -1 se non è presente.
    add   esp, 16  ;
bp1:
    mov   ebx, eax ; Restituisce l'indice trovato,
    mov   eax, 1   ; ammesso che sia abbastanza piccolo
    int   0x80     ; da poter essere rappresentato come
                   ; valore di uscita.
;
; Ricerca sequenziale all'interno di una lista di valori.
; f_rs (lista, x, a, z) ==> EAX
; Al termine EAX contiene l'indice del valore trovato,
; oppure -1 se questo non c'è.
;
f_rs:
    enter 4, 0
    pusha
    rs_i     equ -4              ; Gli si associa EAX.
    rs_lista equ  8              ; Gli si associa ESI.
    rs_x     equ 12              ; Gli si associa EDX.
```

```
    rs_a     equ 16
    rs_z     equ 20
    ;
    mov   esi, [rs_lista+ebp] ; ESI contiene l'indirizzo
                              ; dell'array.
    mov   edx, [rs_x+ebp]     ; EDX contiene il valore
                              ; cercato.
    ;
    mov   eax, [rs_a+ebp]     ; EAX viene usato come indice
                              ; di scansione.
f_rs_loop:
    cmp   eax, [rs_z+ebp]     ; Se EAX è maggiore
    ja    f_rs_non_trovato    ; dell'indice massimo,
                              ; l'elemento cercato non c'è.
    ;
    cmp   edx, [esi+eax*4]    ; Se il valore cercato
    je    f_rs_trovato        ; corrisponde a quello
                              ; dell'indice corrente,
                              ; termina la scansione.
    ;
    inc   eax                 ; Incrementa l'indice di
    jmp   f_rs_loop           ; scansione e salta
                              ; all'inizio del ciclo.
    ;
f_rs_non_trovato:
    popa             ; Conclude la funzione con EAX = -1.
    mov eax, -1      ;
    leave            ;
    ret              ;
f_rs_trovato:
    mov [rs_i+ebp], eax ; Salva EAX nella variabile locale
                        ; prevista.
    popa                ; Conclude la funzione con EAX pari
    mov eax, [rs_i+ebp] ; al valore salvato nella variabile
```

```
    leave                   ; locale.
    ret                     ;
```

64.14.2 Ricerca binaria

«

Viene mostrato un esempio di programma contenente una funzione che esegue una ricerca binaria all'interno di un array ordinato, di interi con segno. Il metodo utilizzato si rifà a quanto descritto in pseudocodifica nella sezione 62.4.2. Il risultato della scansione viene emesso attraverso il valore restituito dal programma; ciò che si ottiene è precisamente l'indice dell'elemento trovato, oppure −1 se nessun elemento corrisponde.

```
# Ricerca binaria.
#
.section .data
lista: .int 1, 3, 4, 7, 9, 10, 11, 22, 23, 44 # Interi con
                                               # segno.
a:     .int 0                 # Indice minimo.
z:     .int 9                 # Indice massimo.
#
.section .text
.globl _start
#
# Main.
#
_start:
    push z              # f_rb ($lista, $7, a, z) ==> EAX
    push a              # Si cerca il valore 7 nell'array
    push $7             # ordinato «lista», tra gli indici
    push $lista         # «a» e «z». Viene restituito l'indice
    call f_rb           # dell'elemento trovato, oppure -1 se
    add  $16, %esp      # non è presente.
```

```
bp1:
    mov  %eax, %ebx # Restituisce l'indice trovato,
    mov  $1, %eax   # ammesso che sia abbastanza piccolo
    int  $0x80      # da poter essere rappresentato come
                    # valore di uscita.
#
# Ricerca binaria all'interno di una lista ordinata di
# valori.
# f_rb (lista, x, a, z) ==> EAX
# Al termine EAX contiene l'indice del valore trovato,
# oppure -1 se questo non c'è.
#
f_rb:
    enter $4, $0
    pusha
    .equ rb_m,     -4           # Gli si associa EAX.
    .equ rb_lista,  8           # Gli si associa ESI.
    .equ rb_x,     12           # Gli si associa EDX.
    .equ rb_a,     16
    .equ rb_z,     20
    #
    mov  rb_lista(%ebp), %esi # ESI contiene l'indirizzo
                              # dell'array.
    mov  rb_x(%ebp),     %edx # EDX contiene il valore
                              # cercato.
                              # EAX viene usato come
                              # «elemento centrale».
    #
    mov  rb_a(%ebp), %eax     # Calcola l'indice
    add  rb_z(%ebp), %eax     # dell'elemento centrale e lo
    sar  $1, %eax             # mette in EAX. Lo scorrimento
                              # a destra serve a dividere
                              # per due EAX.
    #
```

```
bp2:
    cmp  rb_a(%ebp), %eax      # Se EAX, ovvero l'indice,
    jb   f_rb_non_trovato      # è minore dell'indice minimo,
                               # l'elemento non c'è.
    #
    cmp  (%esi,%eax,4), %edx   # Se il valore cercato è
    jl   f_rb_minore           # minore di quello trovato,
    jg   f_rb_maggiore         # cerca nella parte inferiore;
    je   f_rb_fine             # se è maggiore cerca in
                               # quella superiore; se è
                               # uguale, l'elemento è stato
                               # trovato.
    #
f_rb_minore:
    dec   %eax
    push  %eax         # f_rb (lista, x, a, z) ==> EAX
    push  rb_a(%ebp)   # Si cerca il valore nell'array
    push  %edx         # ordinato «lista», tra gli indici
    push  %esi         # «a» e «z». Viene restituito l'indice
    call  f_rb         # dell'elemento trovato, oppure -1 se
    add   $16, %esp    # non è presente.
    jmp   f_rb_fine
    #
f_rb_maggiore:
    inc   %eax
    push  rb_z(%ebp)   # f_rb (lista, x, a, z) ==> EAX
    push  %eax         # Si cerca il valore nell'array
    push  %edx         # ordinato «lista», tra gli indici
    push  %esi         # «a» e «z». Viene restituito l'indice
    call  f_rb         # dell'elemento trovato, oppure -1 se
    add   $16, %esp    # non è presente.
    jmp   f_rb_fine
    #
f_rb_non_trovato:
```

```
    mov $-1, %eax    # Conclude la funzione con EAX = -1.
    jmp f_rb_fine
f_rb_fine:
    mov %eax, rb_m(%ebp) # Salva EAX nella variabile locale
    popa                 # prevista. Conclude la funzione
    mov rb_m(%ebp), %eax # con EAX pari al valore salvato
    leave                # nella variabile locale.
    ret                  #
```

```
; Ricerca binaria.
;
section .data
lista:  dd 1, 3, 4, 7, 9, 10, 11, 22, 23, 44 ; Interi con
                                             ; segno.
a:      dd 0                 ; Indice minimo.
z:      dd 9                 ; Indice massimo.
;
section .text
global _start
;
; Main.
;
_start:
    push long [z] ; f_rb ($lista, $7, a, z) ==> EAX
    push long [a] ; Si cerca il valore 7 nell'array ordinato
    push long 7   ; «lista», tra gli indici «a» e «z».
    push lista    ; Viene restituito l'indice dell'elemento
    call f_rb     ; trovato, oppure -1 se non è presente.
    add  esp, 16  ;
bp1:
    mov  ebx, eax ; Restituisce l'indice trovato,
    mov  eax, 1   ; ammesso che sia abbastanza piccolo
    int  0x80     ; da poter essere rappresentato come
                  ; valore di uscita.
```

```
;
; Ricerca binaria all'interno di una lista ordinata di
; valori.
; f_rb (lista, x, a, z) ==> EAX
; Al termine EAX contiene l'indice del valore trovato,
; oppure -1 se questo non c'è.
;
f_rb:
    enter 4, 0
    pusha
    rb_m     equ -4              ; Gli si associa EAX.
    rb_lista equ  8              ; Gli si associa ESI.
    rb_x     equ 12              ; Gli si associa EDX.
    rb_a     equ 16
    rb_z     equ 20
    ;
    mov esi, [rb_lista+ebp] ; ESI contiene l'indirizzo
                            ; dell'array.
    mov edx, [rb_x+ebp]     ; EDX contiene il valore
                            ; cercato.
                            ; EAX viene usato come
                            ; «elemento centrale».
    ;
    mov eax, [rb_a+ebp]     ; Calcola l'indice dell'elemento
    add eax, [rb_z+ebp]     ; centrale e lo mette in EAX. Lo
    sar eax, 1              ; scorrimento a destra serve a
                            ; dividere per due EAX.
    ;
bp2:
    cmp eax, [rb_a+ebp]     ; Se EAX, ovvero l'indice, è
    jb  f_rb_non_trovato    ; minore dell'indice minimo,
                            ; l'elemento non c'è.
    ;
    cmp edx, [esi+eax*4]    ; Se il valore cercato è minore
```

```
    jl  f_rb_minore          ; di quello trovato, cerca nella
    jg  f_rb_maggiore        ; parte inferiore; se è maggiore
    je  f_rb_fine            ; cerca in quella superiore; se
                             ; è uguale, l'elemento è stato
                             ; trovato.
    ;
f_rb_minore:
    dec   eax
    push  eax              ; f_rb (lista, x, a, z) ==> EAX
    push  long [rb_a+ebp]  ; Si cerca il valore nell'array
    push  edx              ; ordinato «lista», tra gli
    push  esi              ; indici «a» e «z». Viene
    call  f_rb             ; restituito l'indice
    add   esp, 16          ; dell'elemento trovato, oppure
    jmp   f_rb_fine        ; -1 se non è presente.
    ;
f_rb_maggiore:
    inc   eax
    push  long [rb_z+ebp]  ; f_rb (lista, x, a, z) ==> EAX
    push  eax              ; Si cerca il valore nell'array
    push  edx              ; ordinato «lista», tra gli
    push  esi              ; indici «a» e «z». Viene
    call  f_rb             ; restituito l'indice
    add   esp, 16          ; dell'elemento trovato, oppure
                           ; -1 se non è presente.
    jmp   f_rb_fine
    ;
f_rb_non_trovato:
    mov eax, -1            ; Conclude la funzione con
    jmp f_rb_fine          ; EAX = -1.
f_rb_fine:
    mov [rb_m+ebp], eax    ; Salva EAX nella variabile
    popa                   ; locale prevista. Conclude la
    mov eax, [rb_m+ebp]    ; funzione con EAX pari al
```

```
    leave                   ; valore salvato nella variabile
    ret                     ; locale.
```

64.14.3 Bubblesort

«

Viene mostrato un esempio di programma che esegue il riordino di array attraverso l'algoritmo Bubblesort. L'array in question contiene numeri interi con segno. L'algoritmo è descritto in pseudocodifica nella sezione 62.5.1.

```
# Bubblesort
#
.section .data
lista: .int 1, 4, 3, 7, 9, 10, 22, 44, 11, 23 # Interi con
                                               # segno.
a:     .int 0                  # Indice minimo.
z:     .int 9                  # Indice massimo.
#
.section .text
.globl _start
#
# Main.
#
_start:
    push  z          # f_bs ($lista, a, z)
    push  a          # Riordina l'array «lista» senza
    push  $lista     # restituire alcun valore.
    call  f_bs       #
    add   $12, %esp #
bp1:
    mov   $0, %ebx   # Restituisce sempre zero.
    mov   $1, %eax   #
    int   $0x80      #
#
```

```
# Riordino con l'algoritmo «bubblesort».
# f_bs (lista, a, z)
#
f_bs:
    enter $0, $0
    pusha
    .equ  bs_lista,  8          # EDX
    .equ  bs_a,     12          #
    .equ  bs_z,     16          #
                                #
    mov bs_lista(%ebp), %edx    # EDX contiene il riferimento
                                # alla lista.
                                # ESI viene usato come indice
                                # di scansione.
                                # EDI viene usato come indice
                                # di scansione.
                                # EAX viene usato per
                                # scambiare i valori.
                                # EBX viene usato per
                                # scambiare i valori.
    mov   bs_a(%ebp), %esi      # ESI parte dall'indice
                                # iniziale.
f_bs_loop_1:
    cmp   bs_z(%ebp), %esi      # Se ESI >= z, termina.
    jae   f_bs_end_loop_1       #
    #
    mov   %esi, %edi            # EDI := ESI - 1
    inc   %edi                  #
f_bs_loop_2:
    cmp   bs_z(%ebp), %edi      # Se EDI > z, termina.
    ja    f_bs_end_loop_2       #
    #
    mov   (%edx,%esi,4), %eax   # Se EBX < EAX scambia
    mov   (%edx,%edi,4), %ebx   # i valori
```

```
    cmp   %eax, %ebx           #
    jl    f_bs_scambio         #
f_bs_loop_2_inc_edi:
    inc   %edi                 # EDI++
    jmp   f_bs_loop_2          #
f_bs_scambio:
    mov   %eax, (%edx,%edi,4) # lista[ESI] :==: lista[EDI]
    mov   %ebx, (%edx,%esi,4) #
    jmp   f_bs_loop_2          #
f_bs_end_loop_2:
    inc   %esi                 # ESI++
    jmp   f_bs_loop_1          #
f_bs_end_loop_1:
    popa
    leave
    ret
```

```
; Bubblesort
;
section .data
lista: dd 1, 4, 3, 7, 9, 10, 22, 44, 11, 23 ; Interi con
                                             ; segno.
a:     dd 0                      ; Indice minimo.
z:     dd 9                      ; Indice massimo.
;
section .text
global _start
;
; Main.
;
_start:
    push  long [z]   ; f_bs ($lista, a, z)
    push  long [a]   ; Riordina l'array «lista» senza
    push  long lista ; restituire alcun valore.
```

```
    call  f_bs        ;
    add   esp, 12     ;
bp1:
    mov   ebx, 0      ; Restituisce sempre zero.
    mov   eax, 1      ;
    int   0x80        ;
;
; Riordino con l'algoritmo «bubblesort».
; f_bs (lista, a, z)
;
f_bs:
    enter 0, 0
    pusha
    bs_lista equ  8           ; EDX
    bs_a     equ 12           ;
    bs_z     equ 16           ;
                              ;
    mov   edx, [bs_lista+ebp] ; EDX contiene il riferimento
                              ; alla lista.
                              ; ESI viene usato come indice
                              ; di scansione.
                              ; EDI viene usato come indice
                              ; di scansione.
                              ; EAX viene usato per
                              ; scambiare i valori.
                              ; EBX viene usato per
                              ; scambiare i valori.
    mov   esi, [bs_a+ebp]     ; ESI parte dall'indice
                              ; iniziale.
f_bs_loop_1:
    cmp   esi, [bs_z+ebp]     ; Se ESI >= z, termina.
    jae   f_bs_end_loop_1     ;
    ;
    mov   edi, esi            ; EDI := ESI - 1
```

```
    inc   edi                   ;
f_bs_loop_2:
    cmp   edi, [bs_z+ebp]       ; Se EDI > z, termina.
    ja    f_bs_end_loop_2       ;
    ;
    mov   eax, [edx+esi*4]      ; Se EBX < EAX scambia
    mov   ebx, [edx+edi*4]      ; i valori.
    cmp   ebx, eax              ;
    jl    f_bs_scambio          ;
f_bs_loop_2_inc_edi:
    inc   edi                   ; EDI++
    jmp   f_bs_loop_2           ;
f_bs_scambio:
    mov   [edx+edi*4], eax      ; lista[ESI] :==: lista[EDI]
    mov   [edx+esi*4], ebx      ;
    jmp   f_bs_loop_2           ;
f_bs_end_loop_2:
    inc   esi                   ; ESI++
    jmp   f_bs_loop_1           ;
f_bs_end_loop_1:
    popa
    leave
    ret
```

Per verificare il funzionamento del programma si deve usare necessariamente GDB. Inizialmente, prima di mettere in esecuzione il programma, si vede l'array nel suo stato originale:

```
(gdb) print (int[10])lista[Invio]

  $1 = {1, 4, 3, 7, 9, 10, 22, 44, 11, 23}
```

Si fissa quindi uno stop e si avvia il programma:

```
(gdb) break bp1[Invio]
```

(gdb) **run** [*Invio*]

Quando il programma viene sospeso in corrispondenza di '**bp1**', l'array è ordinato:

(gdb) **print (int[10])lista** [*Invio*]

```
$1 = {1, 3, 4, 7, 9, 10, 11, 22, 23, 44}
```

64.15 Calcoli con gli indirizzi in fase di compilazione

Attraverso le funzionalità del compilatore è possibile calcolare la distanza tra due indirizzi, espressa in byte. È anche possibile fare riferimento all'indirizzo attuale, attraverso un simbolo predefinito. Nelle sezioni successive vengono mostrati alcuni esempi per dimostrare l'uso di queste funzionalità.

64.15.1 Distanza tra due indirizzi

L'esempio seguente serve a dimostrare come il compilatore possa calcolare la distanza tra due indirizzi, contrassegnati da dei simboli, inizializzando con tale valore calcolato una variabile globale. In pratica, viene calcolata la grandezza complessiva in byte di un array di numeri interi; grandezza che viene poi emessa come valore di uscita.

```
#
.section .data
lista:  .int    1, 4, 3, 7, 9, 10, 22, 44, 11, 23
z:      .int    (z - lista)
#
.section .text
.globl _start
#
_start:
```

```
    mov   z,  %ebx
    mov   $1, %eax
    int   $0x80
```

```
;
section .data
lista:  dd      1, 4, 3, 7, 9, 10, 22, 44, 11, 23
z:      dd      (z - lista)
;
section .text
global _start
;
_start:
    mov   ebx, [z]
    mov   eax, 1
    int   0x80
```

Il significato dell'istruzione è intuitivo: alla variabile '**z**' si assegna la differenza tra gli indirizzi utilizzati da '**lista**' a '**z**'. In questo caso, il risultato che si ottiene è 40, dal momento che si contano 10 valori da 4 byte ciascuno. Eventualmente, si può fare riferimento alla posizione attuale in modo differente, in questo caso significa sostituire il riferimento esplicito alla variabile '**z**' con un riferimento implicito:

```
…
lista:  .int    1, 4, 3, 7, 9, 10, 22, 44, 11, 23
z:      .int    (. - lista)
…
```

```
…
lista:  dd      1, 4, 3, 7, 9, 10, 22, 44, 11, 23
z:      dd      ($ - lista)
```

```
...
```

Si possono anche fare dei calcoli più complessi, come nel caso dell'esempio seguente in cui si determina l'indice superiore dell'array. Il risultato che si ottiene è nove, dal momento che l'indice del primo elemento deve essere zero.

```
...
lista:  .int    1, 4, 3, 7, 9, 10, 22, 44, 11, 23
z:      .int    (z - lista) / 4 - 1
...
```

```
...
lista:  dd      1, 4, 3, 7, 9, 10, 22, 44, 11, 23
z:      dd      (z - lista) / 4 - 1
...
```

64.15.2 Riempimento di spazio inutilizzato

In certe situazioni è necessario riempire una certa area di memoria (o di codice) in modo che complessivamente occupi una dimensione data. Questo procedimento si usa specialmente quando si genera un file binario, privo di formato (come nel caso di un settore di avvio), che deve avere una dimensione stabilita e che in una certa posizione deve contenere un'impronta determinata. Qui viene dimostrato il concetto intervenendo solo nell'area della memoria che viene inizializzata.

```
#
.section .data
lista:  .int    1, 4, 3, 7, 9, 10, 22, 44, 11, 23
        .skip (0x30 - (. - lista)), 0xFF
#
.section .text
.globl _start
#
_start:
    mov   $0, %ebx
    mov   $1, %eax
    int   $0x80
```

```
;
section .data
lista:  dd      1, 4, 3, 7, 9, 10, 22, 44, 11, 23
times   (30h - ($ - lista))  db  0xFF
;
section .text
global _start
;
_start:
    mov   ebx, 0
    mov   eax, 1
    int   0x80
```

In questo caso, dopo la definizione dell'array, si richiede al compilatore di allocare altro spazio di memoria, in modo da occupare complessivamente 48 byte (30_{16}), riempiendo lo spazio ulteriore con caratteri FF_{16}. Naturalmente, il valore complessivo dello spazio da utilizzare può essere espresso in qualunque base di numerazione; in questo caso, la scelta di rappresentare in base sedici è motivata dal fatto che con NASM non si può usare la forma consueta (non si può

scrivere '**0x30**'), perché si otterrebbe un risultato differente, a causa di un errore di interpretazione da parte del compilatore.

In quasi tutti gli esempi di questo capitolo, realizzati per il compilatore NASM, si usa la notazione '**0x*nn*...**' per esprimere un numero in base sedici. Generalmente l'interpretazione da parte di NASM è corretta, ma nella situazione particolare mostrata, il compilatore si confonde e riconosce solo la forma '***nn*...h**'.

Dal momento che l'array occupa già 40 byte, vengono aggiunti semplicemente 8 byte, pari a due gruppi da 32 bit:

```
(gdb) print /x (int[12])lista[Invio]

$1 = {0x1, 0x4, 0x3, 0x7, 0x9, 0xa, 0x16, 0x2c, 0xb, 0x17,
  0xffffffff, 0xffffffff}
```

64.16 Interazione con il sistema operativo

È possibile gestire un certo grado di comunicazione tra il programma in linguaggio assemblatore e il sistema GNU/Linux. In particolare si possono ottenere i parametri della chiamata del programma (gli argomenti della riga di comando) ed è possibile chiamare delle funzioni di sistema attraverso delle «interruzioni».

64.16.1 Parametri di chiamata del programma

All'avvio del programma, questo riceve una pila contenente il numero degli argomenti della riga di comando (nome del programma incluso), il nome del programma che è stato avviato e quindi gli argomenti stessi. Si può realizzare un sorgente molto semplice per l'indagine con GDB:

```
.section .text
.globl _start
_start:
    mov   %esp, %ebp
bp1:
    mov   $0, %ebx    # Restituisce zero.
    mov   $1, %eax    #
    int   $0x80       #
```

```
section .text
global _start
_start:
    mov   ebp, esp
bp1:
    mov   ebx, 0    ; Restituisce zero.
    mov   eax, 1    ;
    int   0x80      ;
```

Supponendo che il programma compilato si chiami '**argomenti**', lo si avvia sotto il controllo di GDB nello stesso modo di sempre:

```
$ gdb argomenti [Invio]
```

Gli argomenti del programma vanno passati necessariamente attraverso un comando di GDB:

```
(gdb) set args 1 2 3 4 "ciao amore" [Invio]
```

Si fissa il punto di sospensione del programma e quindi si avvia:

```
(gdb) break bp1 [Invio]
```

```
(gdb) run [Invio]
```

Il programma viene sospeso in corrispondenza del simbolo '**bp1**' e si può consultare la pila, o più precisamente lo *stack frame* della pila:

```
(gdb) backtrace[Invio]

 #0  bp1 () at argomenti.s:6
 #1  0x00000006 in ?? ()
 #2  0xbfeb1a89 in ?? ()
 #3  0xbfeb1a98 in ?? ()
 #4  0xbfeb1a9a in ?? ()
 #5  0xbfeb1a9c in ?? ()
 #6  0xbfeb1a9e in ?? ()
 #7  0xbfeb1aa0 in ?? ()
 #8  0x00000000 in ?? ()
```

In questa situazione non sono presenti variabili locali; quindi, nella pila, dopo l'indirizzo corrispondente al simbolo '**bp1**' (che si trova lì solo perché si sta usando GDB ed è stato sospeso il corso del programma), appare la quantità di argomenti (sei); gli elementi successivi contengono dei puntatori alle stringhe che rappresentano i vari argomenti ricevuti (si osservi che gli argomenti sono rappresentati tutti in forma di stringa), stringhe che sono tutte terminate con un byte a zero. Per leggere gli argomenti con GDB si devono fare dei tentativi; qui vengono indicate le dimensioni esatte, ma se si usano dimensioni maggiori si possono vedere delle porzioni degli argomenti successivi:

```
(gdb) print (char[33])*0xbfeb1a89[Invio]

 $1 = "/tmp/argomenti\0001\0002\0003\0004\000ciao amore"
```

In questo caso si legge il primo argomento, ma usando una dimensione eccessiva, si vedono di seguito anche gli altri, separati dai vari byte a zero, rappresentati con la sequenza '**\000**'. Il primo argomento da solo sarebbe:

```
(gdb) print (char[14])*0xbfeb1a89[Invio]
```

```
  $2 = "/tmp/argomenti"
```

Gli altri argomenti:

```
(gdb) print (char[1])*0xbfeb1a98[Invio]

  $3 = "1"
(gdb) print (char[1])*0xbfeb1a9a[Invio]

  $4 = "2"
(gdb) print (char[1])*0xbfeb1a9c[Invio]

  $5 = "3"
(gdb) print (char[1])*0xbfeb1a9e[Invio]

  $6 = "4"
(gdb) print (char[9])*0xbfeb1aa0[Invio]

  $7 = "ciao amore"
(gdb) quit[Invio]
```

64.16.2 Funzioni del sistema operativo

«

Si accede alle funzioni offerte dal sistema operativo attraverso quella che è nota come «interruzione software» (*interrupt*) e si genera con l'istruzione '**INT**'. Per la precisione, in un sistema GNU/Linux occorre l'interruzione 80_{16} (128_{10}), come è stato mostrato in tutti gli esempi apparsi fino a questo punto. Per selezionare il tipo di funzione e per passarle degli argomenti si usano i registri in questo modo:

Registro	Utilizzo
EAX	Contiene il numero che rappresenta la funzione di sistema.
EBX	Primo parametro della funzione.
ECX	Secondo parametro della funzione.
EDX	Terzo parametro della funzione.
ESI	Quarto parametro della funzione.
EDI	Quinto parametro della funzione.

Se la funzione deve restituire un valore, questo viene ottenuto attraverso il registro ***EAX***.

Per una mappa completa delle chiamate di sistema si può consultare *http://wayback.archive.org/web/*/www.lxhp.in-berlin.de/lhpsysc0.html* , come annotato nei riferimenti alla fine del capitolo. Qui vengono mostrate delle tabelle riepilogative di alcune funzioni importanti.

Pagina di manuale	Descrizione	***EAX***	***EBX***	***ECX***	***EDX***
exit(2)	Conclude il funzionamento del programma restituendo un valore.	1	Valore da restituire: numero intero compreso tra zero e 255.		

Pagina di manuale	Descrizione	***EAX***	***EBX***	***ECX***	***EDX***
read(2)	Legge da un descrittore di file e mette i dati letti in una memoria tampone. Attraverso ***EAX*** restituisce la quantità di byte letta effettivamente e aggiorna il puntatore del file per la lettura successiva.	3	Descrittore del file da leggere.	Indirizzo iniziale della memoria tampone.	Dimensione in byte della memoria tampone.
write(2)	Scrive il contenuto di una memoria tampone in un descrittore di file. Attraverso ***EAX*** restituisce la quantità di byte scritta effettivamente.	4	Descrittore del file da scrivere.	Indirizzo iniziale della memoria tampone.	Dimensione in byte della memoria tampone.

64.16.3 Esempi di lettura e scrittura con i flussi standard

«

Di solito, il primo programma che si scrive è quello che visualizza un messaggio e termina, ma in questo caso, un'operazione così semplice sul piano teorico, in pratica è già abbastanza complicata. Quello che segue è un programma che, avvalendosi di una chiamata di sistema, visualizza un messaggio attraverso lo standard output. Come si può osservare, si utilizza anche una tecnica per far calcolare al compilatore la lunghezza della stringa da visualizzare.

```
#
.equ SYS_EXIT,  1   # exit(2)
.equ SYS_WRITE, 4   # write(2)
.equ STDOUT,    1   # Descrittore di standard output.
#
```

```
.section .data                    # Qui si dichiara la stringa
msg:  .ascii "Ciao a tutti!\n"    # da visualizzare,
size = . - msg                    # calcolandone la
                                  # dimensione.
#
.section .text
.globl _start
_start:
    mov   $SYS_WRITE, %eax        # Scrive nello standard
    mov   $STDOUT,    %ebx        # output.
    mov   $msg,       %ecx        #
    mov   $size,      %edx        #
    int   $0x80
exit:
    mov   $SYS_EXIT,  %eax        # Conclude il funzionamento.
    mov   $0,         %ebx        #
    int   $0x80
```

```
;
SYS_EXIT   equ     1  ; exit(2)
SYS_WRITE  equ     4  ; write(2)
STDOUT     equ     1  ; Descrittore di standard output.
;
section .data                     ; Qui si dichiara la stringa
msg:  db  "Ciao a tutti!", 0x0A   ; da visualizzare,
size  equ $ - msg                 ; calcolandone la
                                  ; dimensione.
;
section text
global _start
_start:
    mov   eax, SYS_WRITE          ; Scrive nello standard
    mov   ebx, STDOUT             ; output.
    mov   ecx, msg                ;
```

```
    mov   edx, size             ;
    int   0x80
exit:
    mov   eax, SYS_EXIT         ; Conclude il funzionamento.
    mov   ebx, 0                ;
    int   0x80
```

Segue un esempio di programma che legge dallo standard input e scrive ciò che ha letto attraverso lo standard output. Come già nell'esempio precedente, vengono dichiarate inizialmente delle costanti per semplificare la lettura del codice; inoltre vengono usate aree di memoria non inizializzate e delle funzioni banali senza parametri, per le quali non si utilizzano variabili locali. Si mostrano due listati, uno adatto per GNU AS e l'altro per NASM.

```
#
.equ SYS_EXIT,  1     # exit(2)
.equ SYS_READ,  3     # read(2)
.equ SYS_WRITE, 4     # write(2)
.equ STDIN,     0     # Descrittore di standard input.
.equ STDOUT,    1     # Descrittore di standard output.
.equ STDERR,    2     # Descrittore di standard error.
.equ MAX_SIZE,  1000  # Dimensione massima dei dati da
                      # leggere.
#
.section .data        # Non ci sono variabili già
                      # inizializzate.
#
.section .bss
.lcomm record, MAX_SIZE # Memoria tampone per la lettura dei
                        # dati.
.lcomm size, 4          # Quantità di byte letti
                        # effettivamente.
```

```
#
.section .text
.globl _start
_start:
read_write_begin:
    call  read              # Legge dallo standard input.
    cmp   $0, %eax          # Se sono stati letti zero byte,
    jz    read_write_end    # il ciclo termina.
    call  write             # Scrive i byte letti nello
                            # standard output.
    jmp   read_write_begin # Ripete il ciclo.
read_write_end:
    jmp   exit
read:
    mov   $SYS_READ,  %eax # Legge dallo standard input.
    mov   $STDIN,     %ebx #
    mov   $record,    %ecx #
    mov   $MAX_SIZE,  %edx #
    int   $0x80
    mov   %eax,       size # Salva la dimensione letta
                           # effettivamente.
    ret
write:
    mov   $SYS_WRITE, %eax # Scrive nello standard output.
    mov   $STDOUT,    %ebx #
    mov   $record,    %ecx #
    mov   size,       %edx #
    int   $0x80
    ret
exit:
    mov   $SYS_EXIT,  %eax # Conclude il funzionamento.
    mov   $0,         %ebx #
    int   $0x80
```

```
;
SYS_EXIT   equ     1 ; exit(2)
SYS_READ   equ     3 ; read(2)
SYS_WRITE  equ     4 ; write(2)
STDIN      equ     0 ; Descrittore di standard input.
STDOUT     equ     1 ; Descrittore di standard output.
STDERR     equ     2 ; Descrittore di standard error.
MAX_SIZE   equ  1000 ; Dimensione massima dei dati da
                     ; leggere.
;
section .data        ; Non ci sono variabili già
                     ; inizializzate.
;
section .bss
record resb MAX_SIZE ; Memoria tampone per la lettura dei
                     ; dati.
size   resd 1        ; Quantità di byte letti
                     ; effettivamente.
;
section .text
global _start
_start:
read_write_begin:
    call  read             ; Legge dallo standard input.
    cmp   eax, 0           ; Se sono stati letti zero byte,
    jz    read_write_end   ; il ciclo termina.
    call  write            ; Scrive i byte letti nello
                           ; standard output.
    jmp   read_write_begin ; Ripete il ciclo.
read_write_end:
    jmp   exit
read:
    mov   eax, SYS_READ    ; Legge dallo standard input.
    mov   ebx, STDIN       ;
```

```
    mov   ecx, record       ;
    mov   edx, MAX_SIZE     ;
    int   0x80
    mov   [size], eax       ; Salva la dimensione letta
                            ; effettivamente.
    ret
write:
    mov   eax, SYS_WRITE    ; Scrive nello standard output.
    mov   ebx, STDOUT       ;
    mov   ecx, record       ;
    mov   edx, [size]       ;
    int   0x80
    ret
exit:
    mov   eax, SYS_EXIT     ; Conclude il funzionamento.
    mov   ebx, 0            ;
    int   0x80
```

64.17 Riferimenti

«

- *Intel 80386 Reference Programmer's Manual*, http://pdos.csail.mit.edu/6.828/2006/readings/i386/toc.htm
  - *80386 Instruction Set*, http://pdos.csail.mit.edu/6.828/2006/readings/i386/c17.htm
- *Using Assembly Language in Linux*, http://asm.sourceforge.net/articles/linasm.html
- Norman Matloff, *Introduction to Linux Intel Assembly Language*, http://heather.cs.ucdavis.edu/~matloff/50/LinuxAssembly.html

• H.-Peter Recktenwald, *i386-PC-Linux System Calls*, 2000, *http://wayback.archive.org/web/*/www.lxhp.in-berlin.de/lhpsysc0.html*

[1] **GNU Binutils** GNU GPL

[2] **NASM** GNU LGPL

[3] **GNU Binutils** GNU GPL

[4] **GDB** GNU GPL

[5] **DDD** GNU GPL

[6] Il formato ELF prevede altri tipi di sezione, ma quelle di uso più frequente sono rappresentate nel modello sintattico.

[7] Evidentemente, il valore di uscita viene espresso in base otto: 24_8 è uguale a 20_{10}.

[8] Dipende dal compilatore se è possibile limitare effettivamente l'uso al solo registro ***CX***.

[9] L'indirizzo di memoria da raggiungere con l'istruzione '**`CALL`**', può essere fornito in modo «immediato», attraverso l'indicazione di un simbolo, oppure con un registro o con un indirizzo di memoria. Nell'ipotesi di un registro o di un indirizzo di memoria, si intende che il contenuto del registro o della variabile in memoria vadano considerati come l'indirizzo di destinazione della chiamata.

[10] Nel caso di un valore in virgola mobile, il risultato potrebbe essere atteso dal registro ***ST0***, ma la gestione della virgola mobile non viene affrontata in questo capitolo.

Capitolo 65

Compilazione e formato binario eseguibile

65.1 Compilazione di programmi composti da più file sorgenti 410

65.1.1 Inclusione di file 410
65.1.2 Due file sorgenti da collegare assieme 411
65.1.3 Incorporazione di codice in linguaggio C 418

65.2 Librerie dinamiche e librerie statiche 419

65.2.1 Il processo di «collegamento» dinamico 420
65.2.2 Creazione di una libreria dinamica 421
65.2.3 Creare un programma che utilizza una libreria dinamica 426
65.2.4 Creare un file che utilizza una libreria dinamica standard 429
65.2.5 Librerie statiche 432

65.3 Dal sorgente all'immagine in memoria 432

65.3.1 File oggetto 433
65.3.2 File eseguibile 435
65.3.3 Immagine del processo nella memoria virtuale . 439
65.3.4 Allineamento dei segmenti in memoria 442
65.3.5 Script per il collegamento 443
65.3.6 Osservazioni sui simboli 449
65.3.7 Formati dei file oggetto 452

65.4 Formato ELF 453

65.4.1 Sezioni e segmenti 453
65.4.2 Intestazione ELF 454
65.4.3 Descrizione dei segmenti 459
65.4.4 Definizione manuale di un formato ELF 462
65.4.5 Esempio più complesso 467
65.5 Programmi completamente autonomi 470
65.5.1 Le specifiche «multiboot» 471
65.5.2 Esempio di programma da avviare secondo le specifiche «multiboot» 478
65.5.3 Visualizzazione di messaggi 482
65.5.4 Colori dello schermo 493
65.6 Compilazione C dal basso in alto 495
65.6.1 Compilazione di un programma che non fa uso di librerie 495
65.6.2 Uso di GDB e di DDD 499
65.6.3 Da «_start» a «main» 504
65.6.4 Compilazione naturale di un programma in linguaggio C 507
65.7 Compilazione C dall'alto in basso 509
65.7.1 Le fasi della compilazione 510
65.7.2 Precompilatore 511
65.7.3 Compilazione dei file intermedi 512
65.7.4 L'uso di librerie 513
65.7.5 Librerie statiche e librerie dinamiche 514

65.7.6 L'ordine dei file e delle librerie nella compilazione 517
65.7.7 Prevenzione e ricerca degli errori 518
65.7.8 Problemi con l'ottimizzazione 520
65.8 Compilazione guidata con Make 523
65.8.1 Obiettivo, dipendenze e comandi 524
65.8.2 Obiettivi fittizi 527
65.8.3 Scelta dell'obiettivo 529
65.8.4 Interpretazione dei comandi che portano a un obiettivo 530
65.8.5 Variabili o «macro» 533
65.8.6 Utilizzo oculato delle variabili 538
65.8.7 Espansione e continuazione al di fuori dei comandi 539
65.8.8 Variabili automatiche 540
65.8.9 Regole implicite 542
65.8.10 Uno script per ogni sottodirectory 544
65.8.11 Una regola per più obiettivi 546
65.8.12 Regole fittizie tipiche 547
65.8.13 Variabili per l'installazione 549
65.8.14 Definizione della shell 550
65.8.15 Installazione dei programmi 551
65.9 Riferimenti 552

65.1 Compilazione di programmi composti da più file sorgenti

«

Per poter compilare un programma distribuito tra più file sorgenti, all'interno di questi file occorre dichiarare quali simboli (riferiti a variabili e funzioni) devono essere pubblici e come tali accessibili anche dagli altri; inoltre, nei file in cui si fa riferimento a simboli esterni, occorre dichiarare questa dipendenza.

65.1.1 Inclusione di file

«

Prima di affrontare il problema del collegamento di più file oggetto in un file eseguibile singolo, conviene considerare l'inclusione automatica del contenuto di un file. In altri termini, si può ottenere una funzione simile al «copia-incolla», dichiarando in un file che, in un certo punto, va incluso il contenuto di un altro. Per esempio, per incorporare in un certo punto, il contenuto del file '`funzioni.s`', occorre scrivere la direttiva seguente:

```
…
; GNU AS
.include "funzioni.s"
…
```

```
…
; NASM
%include "funzioni.s"
…
```

Naturalmente, il file '`funzioni.s`' contiene qualcosa che si può copiare e incollare, tale e quale, in quel certo punto del sorgente che

vi fa riferimento.

Il file in questione viene cercato nella directory corrente, ma nella riga di comando di GNU AS è possibile aggiungere l'opzione '**-I**' per indicare altri percorsi di ricerca; nello stesso modo, con NASM si può usare l'opzione '**-i**':

```
$ as -I directory ... [Invio]

$ nasm -i directory ... [Invio]
```

65.1.2 Due file sorgenti da collegare assieme

Per dimostrare come si gestiscono più file sorgenti assieme, viene mostrato un esempio molto semplice, composto da due soli file: nel primo si trova la funzione '**_start**'; nel secondo si trova la funzione '**f_rs**' che esegue una ricerca sequenziale all'interno di un array (nella sezione 64.14.1 si vede lo stesso programma, tutto intero, in un file unico). I due listati seguenti sono realizzati per GNU AS:

```
# rs-main.s
#
.section .data
lista: .int 1, 4, 3, 7, 9, 10, 22, 44, 11, 23 # Interi senza
                                               # segno.
a:     .int 0                          # Indice minimo.
z:     .int 9                          # Indice massimo.
#
.section .text
.globl _start
.extern f_rs
#
_start:
    push  z           # f_rs ($lista, $7, a, z) ==> EAX
```

```
    push  a           # Si cerca il valore 7 nell'array
    push  $7          # «lista», tra gli indici «a» e «z».
    push  $lista      # Viene restituito l'indice
    call  f_rs        # dell'elemento trovato, oppure -1 se
    add   $16, %esp   # non è presente.
bp1:
    mov   %eax, %ebx # Restituisce l'indice trovato,
    mov   $1, %eax   # ammesso che sia abbastanza piccolo
    int   $0x80      # da poter essere rappresentato come
                     # valore di uscita.
```

```
# rs-f.s
#
.section .data
#
.section .text
.globl f_rs
#
# Ricerca sequenziale all'interno di una lista di valori.
# f_rs (lista, x, a, z) ==> EAX
# Al termine EAX contiene l'indice del valore trovato,
# oppure -1 se questo non c'è.
#
f_rs:
    enter $4, $0
    pusha
    .equ  rs_i,     -4          # Gli si associa EAX.
    .equ  rs_lista,  8          # Gli si associa ESI.
    .equ  rs_x,     12          # Gli si associa EDX.
    .equ  rs_a,     16
    .equ  rs_z,     20
    #
    mov   rs_lista(%ebp), %esi # ESI contiene l'indirizzo
                               # dell'array.
```

```
    mov   rs_x(%ebp),     %edx # EDX contiene il valore
                               # cercato.
    #
    mov   rs_a(%ebp),     %eax # EAX viene usato come indice
                               # di scansione.
f_rs_loop:
    cmp   rs_z(%ebp),     %eax # Se EAX è maggiore
    ja    f_rs_non_trovato     # dell'indice massimo,
                               # l'elemento cercato non c'è.
    #
    cmp   (%esi,%eax,4),  %edx # Se il valore cercato
    je    f_rs_trovato         # corrisponde a quello
                               # dell'indice corrente,
                               # termina la scansione.
    #
    inc   %eax                 # Incrementa l'indice di
    jmp   f_rs_loop            # scansione e salta
                               # all'inizio del ciclo.
    #
f_rs_non_trovato:
    popa                # Conclude la funzione con EAX = -1.
    mov $-1, %eax       #
    leave               #
    ret                 #
f_rs_trovato:
    mov %eax, rs_i(%ebp)   # Salva EAX nella variabile
                           # locale prevista.
    popa                   # Conclude la funzione con EAX
    mov rs_i(%ebp), %eax   # pari al valore salvato nella
    leave                  # variabile locale.
    ret                    #
```

Nel primo dei due listati, corrispondente al file '`rs-main.s`', si deve osservare la dichiarazione esterna del simbolo '**`f_rs`**', cor-

rispondente al nome della funzione contenuta nel file ‘rs-f.s’.

```
…
.section .text
.globl _start
.extern f_rs
…
```

Dal momento che i dati necessari all’elaborazione vengono passati alla funzione attraverso i parametri della chiamata, a parte ‘**_start**’, non ci sono altre dichiarazioni di simboli pubblici nel file ‘f-main.s’. Nel secondo listato, corrispondente al file ‘rs-f.s’, il simbolo ‘**f_rs**’ viene reso pubblico, per consentire al file ‘rs-main.s’ di farvi riferimento.

```
…
.section .text
.globl f_rs
…
```

Seguono gli stessi due listati, nella versione adatta a NASM:

```
; rs-main.s
;
section .data
lista: dd 1, 4, 3, 7, 9, 10, 22, 44, 11, 23 ; Interi senza
                                            ; segno.
a:     dd 0                              ; Indice minimo.
z:     dd 9                              ; Indice massimo.
;
section .text
global _start
extern f_rs
```

```
;
_start:
    push  long [z] ; f_rs ($lista, $7, a, z) ==> EAX
    push  long [a] ; Si cerca il valore 7 nell'array
    push  long 7   ; «lista», tra gli indici «a» e «z».
    push  lista    ; Viene restituito l'indice dell'elemento
    call  f_rs     ; trovato, oppure -1 se non è presente.
    add   esp, 16  ;
bp1:
    mov   ebx, eax ; Restituisce l'indice trovato,
    mov   eax, 1   ; ammesso che sia abbastanza piccolo
    int   0x80     ; da poter essere rappresentato come
                   ; valore di uscita.
```

```
; rs-f.s
;
section .data
;
section .text
global f_rs
;
; Ricerca sequenziale all'interno di una lista di valori.
; f_rs (lista, x, a, z) ==> EAX
; Al termine EAX contiene l'indice del valore trovato,
; oppure -1 se questo non c'è.
;
f_rs:
    enter 4, 0
    pusha
    rs_i     equ -4              ; Gli si associa EAX.
    rs_lista equ  8              ; Gli si associa ESI.
    rs_x     equ 12              ; Gli si associa EDX.
    rs_a     equ 16
    rs_z     equ 20
```

```
    ;
    mov   esi, [rs_lista+ebp] ; ESI contiene l'indirizzo
                              ; dell'array.
    mov   edx, [rs_x+ebp]     ; EDX contiene il valore
                              ; cercato.
    ;
    mov   eax, [rs_a+ebp]     ; EAX viene usato come indice
                              ; di scansione.
f_rs_loop:
    cmp   eax, [rs_z+ebp]     ; Se EAX è maggiore dell'indice
    ja    f_rs_non_trovato    ; massimo, l'elemento cercato
                              ; non c'è.
    ;
    cmp   edx, [esi+eax*4]    ; Se il valore cercato
    je    f_rs_trovato        ; corrisponde a quello
                              ; dell'indice corrente,
                              ; termina la scansione.
    ;
    inc   eax                 ; Incrementa l'indice di
    jmp   f_rs_loop           ; scansione e salta all'inizio
                              ; del ciclo.
    ;
f_rs_non_trovato:
    popa                ; Conclude la funzione con EAX = -1.
    mov eax, -1         ;
    leave               ;
    ret                 ;
f_rs_trovato:
    mov [rs_i+ebp], eax ; Salva EAX nella variabile
                        ; locale prevista.
    popa                ; Conclude la funzione con EAX pari
    mov eax, [rs_i+ebp] ; al valore salvato nella variabile
    leave               ; locale.
    ret                 ;
```

In questo caso, le direttive salienti sono, rispettivamente:

```
…
section .text
global _start
extern f_rs
…
```

```
…
section .text
global f_rs
…
```

Per compilare il tutto in un solo file eseguibile, occorre procedere secondo i comandi seguenti. Nel caso di GNU AS:

```
$ as --gstabs -o rs-main.o rs-main.s [Invio]

$ as --gstabs -o rs-f.o rs-f.s [Invio]

$ ld -o rs rs-main.o rs-f.o [Invio]
```

Nel caso di NASM:

```
$ nasm -g -f elf -o rs-main.o rs-main.s [Invio]

$ nasm -g -f elf -o rs-f.o rs-f.s [Invio]

$ ld -o rs rs-main.o rs-f.o [Invio]
```

Questo programma restituisce l'indice dell'elemento cercato e trovato nell'array. In questo caso, si tratta del quarto elemento che corrisponde all'indice 3:

```
$ ./rs ; echo $? [Invio]
```

3

65.1.3 Incorporazione di codice in linguaggio C

«

Per collegare assieme sorgenti scritti in linguaggi differenti, si agisce in modo analogo a quanto già mostrato per il solo linguaggio assemblatore. C'è però da considerare che ogni compilatore ha le proprie caratteristiche, sia per ciò che riguarda le convenzioni di chiamata delle funzioni, sia per il modo di nominare i simboli associati alle funzioni stesse. Nel caso di GCC (*GNU compiler collection*), valgono le convenzioni di chiamata comuni e i nomi delle funzioni non vengono modificati.

Qui si mostra un listato, in linguaggio C, da usare in sostituzione del file '`rs-f.s`' descritto nella sezione precedente:

```
/* f_rs (<lista>, <x>, <ele-inf>, <ele-sup>) */

int f_rs (int lista[], int x, int a, int z)
{
    int i;

    /* Scandisce l'array alla ricerca dell'elemento. */

    for (i = a; i <= z; i++)
      {
        if (x == lista[i])
          {
            return i;
          }
      }

    /* La corrispondenza non è stata trovata. */
```

```
    return -1;
}
```

Per compilare questo file e generare un file oggetto, ammesso che il sorgente si chiami ‘`rs-f.c`’, si procede con il comando seguente:

```
$ cc -c -o rs-f.o rs-f.c[Invio]
```

Il collegamento con il file ‘`rs-main.o`’ avviene nel modo già visto:

```
$ ld -o rs rs-main.o rs-f.o[Invio]
```

65.2 Librerie dinamiche e librerie statiche

La compilazione dei programmi, secondo quanto descritto in precedenza, genera sempre file eseguibili «completi», in quanto incorporano tutto il codice necessario al proprio funzionamento. Oltre che suddividere il sorgente in file separati, da riunire assieme in un file eseguibile unico, è possibile costruire una libreria di funzioni, a cui i programmi accedono dopo essere stati avviati, senza incorporarne il codice. Un libreria di questo genere è nota come ***libreria dinamica***, in quanto richiede la creazione di un «collegamento» (*link*) istantaneo, mentre il programma che la richiede è in funzione.

Il concetto di libreria dinamica si contrappone a quello di ***libreria statica***, la quale comporta l’inclusione del proprio codice nel file eseguibile, in fase di compilazione.

65.2.1 Il processo di «collegamento» dinamico

«

Il programma eseguibile che ha bisogno di utilizzare una libreria dinamica, si avvale di un altro programma che a sua volta deve eseguire il «collegamento dinamico» (*dynamic link*). Il nome di questo collegatore dinamico viene definito in fase di compilazione del primo programma e in un sistema GNU/Linux è costituito generalmente dal file '`/lib/ld-linux.so.2`'. A sua volta, il collegatore dinamico cerca il file contenente la libreria richiesta dal programma in un gruppo di directory che solitamente sono '`/lib/`', '`/usr/lib/`' e altre, secondo la configurazione contenuta nel file '`/etc/ld.so.conf`'.

Il file '`/etc/ld.so.conf`' deve essere elaborato attraverso il programma '**`ldconfig`**' che a sua volta produce il file '`/etc/ld.so.cache`', il quale viene interpellato effettivamente da '`/lib/ld-linux.so.2`'. Pertanto, quando si modifica il file '`/etc/ld.so.conf`', occorre ricordarsi di riavviare '**`ldconfig`**'.

Se esiste la variabile di ambiente '**`LD_LIBRARY_PATH`**', i file delle librerie vengono cercati nei percorsi che questa contiene. Per esempio, per utilizzare i file contenuti nella directory corrente, continuando eventualmente in altre directory consuete, basta assegnare il percorso '`.`', seguito dagli altri a cui si è interessati:

```
$ export LD_LIBRARY_PATH=".:/lib:/usr/lib:/usr/local/lib" [Invio]
```

65.2.2 Creazione di una libreria dinamica

Per compilare dei file sorgenti in modo che diventino una libreria dinamica, occorre usare delle opzioni particolari in fase di collegamento (*link*) e nei file sorgenti è necessario pubblicizzare le funzioni in modo particolare. A titolo di esempio si prendono due funzioni, rispettivamente per il calcolo della potenza e del fattoriale (sono già usate nella sezione 64.11 in programmi compilati in modo statico), contenute in due file separati. La coppia di listati è completa e vengono mostrate entrambe le versioni per GNU AS e NASM, evidenziando le direttive significative per ottenere una libreria dinamica.

```
# lib_pwr.s
.section .text
.globl f_pwr
.type f_pwr, @function
#
f_pwr:
    enter $4, $0
    pusha
    #
    mov   8(%ebp), %esi  # Base.
    mov   12(%ebp), %edi # Esponente.
    #
    cmp   $0, %esi       # Se la base è pari a 0,
    jz    f_pwr_end_0    # restituisce 0.
    #
    cmp   $0, %edi       # Se l'esponente è pari a 0,
    jz    f_pwr_end_1    # restituisce 1.
    #
    dec   %edi           # Riduce l'esponente di una unità.
    push  %edi           # f_pwr (ESI, EDI) ==> EAX
    push  %esi           #
```

```
    call  f_pwr           #
    add   $8, %esp        #
    mul   %esi            # EDX:EAX = EAX*ESI
    mov   %eax, -4(%ebp)  # Salva il risultato.
    jmp   f_pwr_end_X     # Conclude la funzione.
    #
f_pwr_end_0:
    popa                  # Conclude la funzione con EAX = 0.
    mov $0, %eax          #
    leave                 #
    ret                   #
f_pwr_end_1:
    popa                  # Conclude la funzione con EAX = 1.
    mov $1, %eax          #
    leave                 #
    ret                   #
f_pwr_end_X:
    popa                  # Conclude la funzione con EAX pari
    mov -4(%ebp), %eax    # al valore salvato nella variabile
    leave                 # locale.
    ret                   #
```

```
# lib_fact.s
.section .text
.globl f_fact
.type f_fact, @function
#
f_fact:
    enter $4, $0
    pusha
    #
    mov   8(%ebp), %edi   # Valore di cui calcolare il
                          # fattoriale.
    #
```

```
    cmp   $1, %edi        # Il fattoriale di 1 è 1.
    jz    f_fact_end_1    #
    #
    mov   %edi, %esi      # ESI contiene il valore di cui si
    dec   %esi            # vuole il fattoriale, ridotto di
                          # una unità.
    #
    push  %esi            # f_fact (ESI) ==> EAX
    call  f_fact          #
    add   $4, %esp        #
    mul   %edi            # EDX:EAX = EAX*EDI
    mov   %eax, -4(%ebp)  # Salva il risultato.
    jmp   f_fact_end_X    # Conclude la funzione.
    #
f_fact_end_1:
    popa                  # Conclude la funzione con EAX = 1.
    mov $1, %eax          #
    leave                 #
    ret                   #
f_fact_end_X:
    popa                  # Conclude la funzione con EAX pari
    mov -4(%ebp), %eax    # al valore salvato nella variabile
    leave                 # locale.
    ret                   #
```

```
; lib_pwr.s
section .text
global f_pwr:function
;
f_pwr:
    enter 4,0
    pusha
    ;
    mov   esi, [ebp+8]  ; Base.
```

```
    mov   edi, [ebp+12] ; Esponente.
    ;
    cmp   esi, 0        ; Se la base è pari a 0,
    jz    f_pwr_end_0   ; restituisce 0.
    ;
    cmp   edi, 0        ; Se l'esponente è pari a 0,
    jz    f_pwr_end_1   ; restituisce 1.
    ;
    dec   edi           ; Riduce l'esponente di una unità.
    push  edi           ; f_pwr (ESI, EDI) ==> EAX
    push  esi           ;
    call  f_pwr         ;
    add   esp, 8        ;
    mul   esi           ; EDX:EAX = EAX*ESI
    mov   [ebp-4], eax  ; Salva il risultato.
    jmp   f_pwr_end_X   ; Conclude la funzione.
    ;
f_pwr_end_0:
    popa                ; Conclude la funzione con EAX = 0.
    mov eax, 0          ;
    leave               ;
    ret                 ;
f_pwr_end_1:
    popa                ; Conclude la funzione con EAX = 1.
    mov eax, 1          ;
    leave               ;
    ret                 ;
f_pwr_end_X:
    popa                ; Conclude la funzione con EAX pari
    mov eax, [ebp-4]    ; al valore salvato nella variabile
    leave               ; locale.
    ret                 ;
```

```
; lib_fact.s
```

```
section .text
global f_fact:function
;
f_fact:
    enter 4,0
    pusha
    ;
    mov   edi, [ebp+8] ; Valore di cui calcolare il
                       ; fattoriale.
    ;
    cmp   edi, 1       ; Il fattoriale di 1 è 1.
    jz    f_fact_end_1 ;
    ;
    mov   esi, edi     ; ESI contiene il valore di cui si
    dec   esi          ; vuole il fattoriale, ridotto di
                       ; una unità.
    ;
    push  esi          ; f_fact (ESI) ==> EAX
    call  f_fact       ;
    add   esp, 4       ;
    mul   edi          ; EDX:EAX = EAX*EDI
    mov   [ebp-4], eax ; Salva il risultato.
    jmp   f_fact_end_X ; Conclude la funzione.
    ;
f_fact_end_1:
    popa               ; Conclude la funzione con EAX = 1.
    mov eax, 1         ;
    leave              ;
    ret                ;
f_fact_end_X:
    popa               ; Conclude la funzione con EAX pari
    mov eax, [ebp-4]   ; al valore salvato nella variabile
    leave              ; locale.
    ret                ;
```

Come si può osservare, non basta dichiarare come globale il simbolo della funzione: occorre anche specificare il suo ruolo di funzione.

Ammesso che i file si chiamino, rispettivamente, '`lib_pwr.s`' e '`lib_fact.s`', si compilano come di consueto per ottenere i file oggetto relativi:

```
$ as --gstabs -o lib_pwr.o lib_pwr.s[Invio]

$ as --gstabs -o lib_fact.o lib_fact.s[Invio]
```

Ovvero:

```
$ nasm -g -f elf -o lib_pwr.o lib_pwr.s[Invio]

$ nasm -g -f elf -o lib_fact.o lib_fact.s[Invio]
```

Poi, per ottenere la libreria vera e propria, si procede con '**`ld`**' nel modo seguente (a questo punto non fa differenza l'origine dei file oggetto):

```
$ ld -shared -o libmate.so lib_pwr.o lib_fact.o[Invio]
```

Così facendo si ottiene il file '`libmate.so`' che costituisce la libreria voluta (la sigla «so» sta per *Shared object*).

65.2.3 Creare un programma che utilizza una libreria dinamica

«

Seguendo l'esempio della sezione precedente, si può creare un programma che si avvale della funzione '**`f_fact`**', contenuta nella libreria dinamica '`libmate.so`':

```
# op1!
#
.section .data
```

```
op1:    .int    5
#
.section .text
.globl _start
.extern f_fact
#
_start:
    push  op1         # f_fact (op1) ==> EAX
    call  f_fact      #
    add   $4, %esp    #
    #
    mov   %eax, %ebx # Restituisce il valore del fattoriale,
    mov   $1, %eax    # ammesso che sia abbastanza piccolo
    int   $0x80       # da poter essere rappresentato come
                      # valore di uscita.
```

```
; op1!
;
section .data
op1:    dd    5
;
section .text
global _start
extern f_fact
;
_start:
    push  long [op1] ; f_fact (op1) ==> EAX
    call  f_fact     ;
    add   esp, 4     ;
    ;
    mov   ebx, eax   ; Restituisce il valore del fattoriale,
    mov   eax, 1     ; ammesso che sia abbastanza piccolo
    int   0x80       ; da poter essere rappresentato come
                     ; valore di uscita.
```

La compilazione per produrre il file oggetto avviene nel modo consueto:

```
$ as --gstabs -o fact.o fact.s[Invio]
```

Ovvero:

```
$ nasm -g -f elf -o fact.o fact.s[Invio]
```

Poi, la trasformazione in file eseguibile richiede l'uso di opzioni particolari per '**ld**':

```
$ ld -L . ↩
↪    -dynamic-linker /lib/ld-linux.so.2 ↩
↪    -lmate ↩
↪    -o fact ↩
↪    fact.o[Invio]
```

Vanno osservate alcune opzioni:

Opzione	Descrizione
`-L .`	Indica di cercare la libreria nella directory corrente.
`-dynamic-linker /lib/ld-linux.so.2`	Indica di usare, al momento dell'avvio del programma che si sta creando, il «collegatore dinamico» costituito dal file '`/lib/ld-linux.so.2`'.
`-lmate`	Indica di instaurare il collegamento dinamico con la libreria «mate», ovvero il file '`libmate.so`' (che secondo l'opzione '**`-L .`**' si trova nella directory corrente).

Dato il modo in cui viene usata l'opzione '**-l**', si comprende che i file delle librerie devono avere sempre un nome che inizia per '`lib`…'.

Dall'ultimo comando mostrato si ottiene il file eseguibile '**fact**' nella directory corrente, il quale ha bisogno della libreria '`libmate.so`'. Se si vuole avviare questo programma, è necessario che il file della libreria si trovi in uno dei percorsi previsti. In questo caso si trova provvisoriamente nella directory corrente e si può utilizzare la variabile di ambiente '**LD_LIBRARY_PATH**' per istruire di conseguenza il collegatore dinamico:

```
$ LD_LIBRARY_PATH="." ./fact ; echo $?[Invio]

  120
```

Con '**ldd**' si può verificare la dipendenza del programma dalle librerie, ma anche in questo caso va utilizzata la variabile di ambiente '**LD_LIBRARY_PATH**':

```
$ LD_LIBRARY_PATH="." ldd fact[Invio]

        linux-gate.so.1 =>  (0xffffe000)
        libmate.so => ./libmate.so (0xb7f46000)
```

65.2.4 Creare un file che utilizza una libreria dinamica standard

Così come è possibile utilizzare le proprie librerie dinamiche, si possono sfruttare benissimo quelle scritte da altri autori. Per poter utilizzare le funzioni comuni del linguaggio C, ci si può avvalere della libreria omonima, '**c**', ovvero '`libc.so`', che di norma si trova nella directory '`/lib/`'. A titolo di esempio viene mostrato un programma che emette un messaggio attraverso lo standard output:

```
# hello.s
#
.section .data
msg:    .ascii  "Ciao mondo!\n\0"
#
.section .text
.globl _start
.extern printf
.extern exit
#
_start:
    push  $msg          # printf (msg)
    call  printf        #
    add   $4, %esp      #
    #
    push  $0            # exit (0)
    call  exit          #
```

```
; hello.s
;
section .data
msg:    db      "Ciao mondo!", 0x0A, 0x00
;
section .text
global _start
extern printf
extern exit
;
_start:
    push  long msg      ; printf (msg)
    call  printf        ;
    add   esp, 4        ;
    ;
    push  long 0        ; exit (0)
```

```
    call  exit          ;
```

Il programma utilizza due funzioni, '**printf**' e '**exit**', la prima per visualizzare un messaggio e la seconda per concluderne il funzionamento. La funzione '**printf**' richiede come primo argomento (in questo caso anche l'unico) l'indirizzo iniziale di una stringa terminata da un byte completamente a zero: nel sorgente per GNU AS il codice di interruzione di riga e lo zero vengono inseriti con le sequenze '**\n\0**', mentre in quello per NASM i codici relativi sono messi direttamente in forma numerica.

Il sorgente si compila come di consueto per ottenere il file oggetto. Successivamente, il collegamento avviene con il comando seguente:

```
$ ld -dynamic-linker /lib/ld-linux.so.2 ↩
↪    -lc ↩
↪    -o hello ↩
↪    hello.o[Invio]
```

Rispetto al caso descritto nella sezione precedente, si può osservare che manca l'opzione '**-L**', in quanto la libreria va cercata nei percorsi standard previsti; inoltre, conformemente all'esempio già visto, per indicare la libreria è stato usato solo il nome '**c**', da cui l'opzione '**-lc**'.

Dal momento che la libreria si trova nei percorsi standard, per avviare il programma non servono accorgimenti particolari:

```
$ ./hello[Invio]

  Ciao mondo!
```

Con '**ldd**' si può verificare la dipendenza del programma dalle librerie:

```
$ ldd hello[Invio]

        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/tls/libc.so.6 (0xb7e71000)
        /lib/ld-linux.so.2 (0xb7fb3000)
```

65.2.5 Librerie statiche

«

È utile sapere come sono organizzate le «librerie statiche» in un sistema GNU. Di per sé sono semplicemente file oggetto, compilati in modo da rendere pubblici i simboli delle funzioni a cui si può essere interessati esternamente, ma raccolti in un archivio che costituisce la libreria.

```
$ ar -cvq libmate.a lib_pwr.o lib_fact.o[Invio]
```

Il comando appena mostrato crea la libreria «mate» nel file ‘`libmate.a`’, composta dai file oggetto ‘`lib_pwr.o`’ e ‘`lib_fact.o`’. Il file della libreria è un semplice archivio «ar», che non prevede la compressione.

Il modo più semplice per collegare un programma che utilizza una libreria statica di questo genere è quello di indicare il file della libreria come se fosse un file oggetto:

```
$ ld -o fact fact.o libmate.a[Invio]
```

65.3 Dal sorgente all'immagine in memoria

«

Ciò che succede a partire da un file sorgente fino al programma in esecuzione in memoria è definito da un procedimento molto complesso, anche se il compilatore e il sistema operativo consentono di ignorarlo quasi completamente. Qui si mostra un esempio banale,

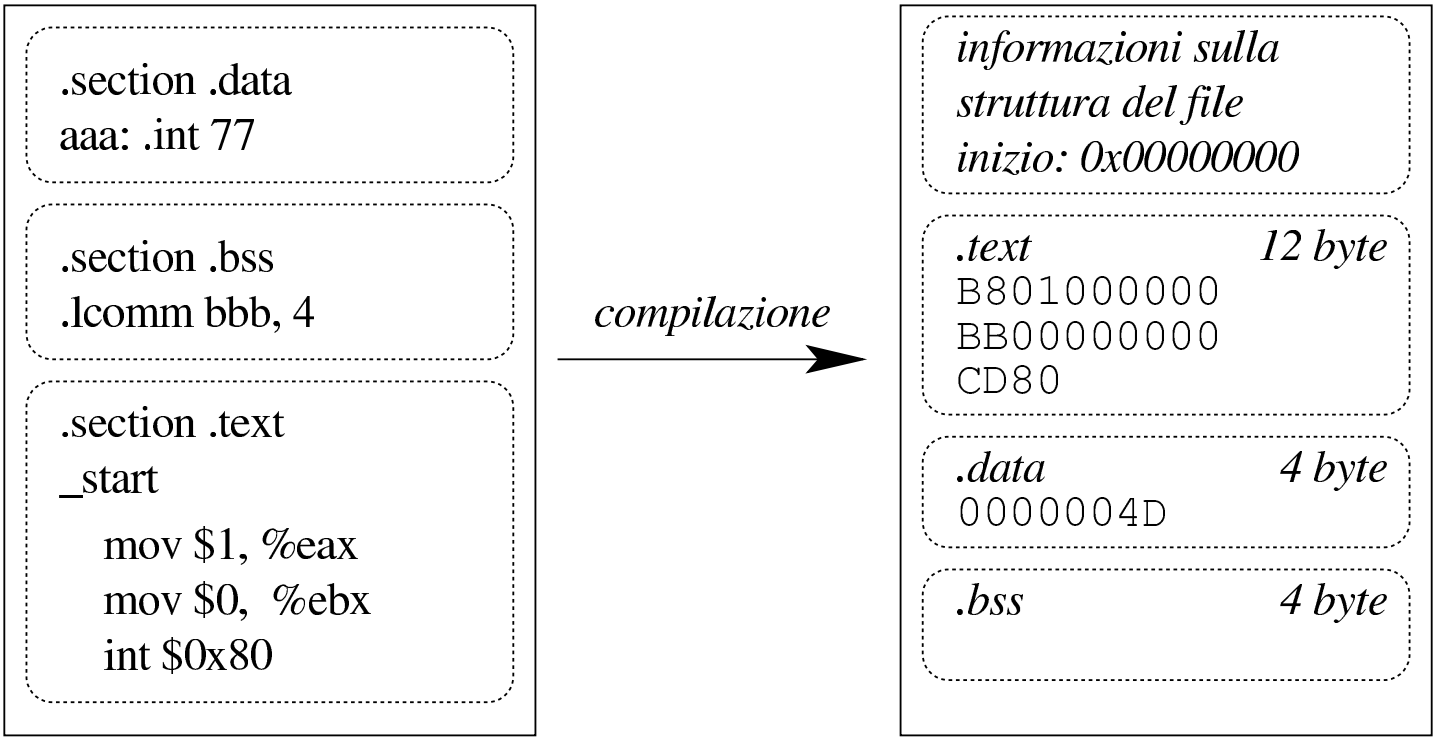
con tutti i passaggi fino ad arrivare all'immagine in memoria, senza però entrare nella questione dell'uso di librerie dinamiche.

La dimostrazione che appare si basa implicitamente sul formato ELF, sia per i file oggetto rilocabili, sia per i file eseguibili, senza entrare per ora nel dettaglio del formato stesso.

65.3.1 File oggetto

Di norma, la compilazione di un sorgente produce un file oggetto ***rilocabile***, ma non eseguibile. Questo file oggetto contiene il codice ottenuto dall'interpretazione del sorgente, diviso in sezioni (come descritto nel sorgente stesso) che possono essere ricomposte, successivamente, con una cerca libertà.

Figura 65.26. Dalle sezioni del file sorgente a quelle del file oggetto rilocabile.



A titolo di esempio si può prendere il file seguente (che riproduce quanto si vede nella figura), compilandolo nel modo consueto (qui

si mostra solo l'uso di GNU AS, per semplicità). Si suppone che il file sorgente si chiami '`prg.s`':

```
.section .data
aaa:   .int 77
.section .bss
.lcomm bbb, 4
.section .text
.globl _start
_start:
    mov   $1, %eax
    mov   $0, %ebx
    int   $0x80
```

```
$ as -o prg.o prg.s[Invio]
```

Con Objdump si può analizzare il contenuto del file oggetto generato:

```
$ objdump -x prg.o[Invio]
```

```
prg.o:     file format elf32-i386
prg.o
architecture: i386, flags 0x00000010:
HAS_SYMS
start address 0x00000000

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000000c  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000004  00000000  00000000  00000040  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  00000000  00000000  00000044  2**2
                  ALLOC
SYMBOL TABLE:
```

```
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l       .data  00000000 aaa
00000000 l     O .bss   00000004 bbb
00000000 g       .text  00000000 _start
```

Anche solo intuitivamente, si comprende che il file oggetto riproduce le tre sezioni del sorgente, assegnando loro degli attributi. Per esempio, la sezione '**.text**' deve essere caricata in memoria, usata in sola lettura e può essere eseguita; in modo analogo, la sezione '**.data**' deve essere caricata in memoria in lettura-scrittura (l'informazione è implicita, in quanto non appare l'attributo '**READONLY**'), ma non può essere eseguita; la sezione '**.bss**' viene allocata soltanto e non prevede limitazioni particolari, a parte il fatto di non poter essere eseguibile.

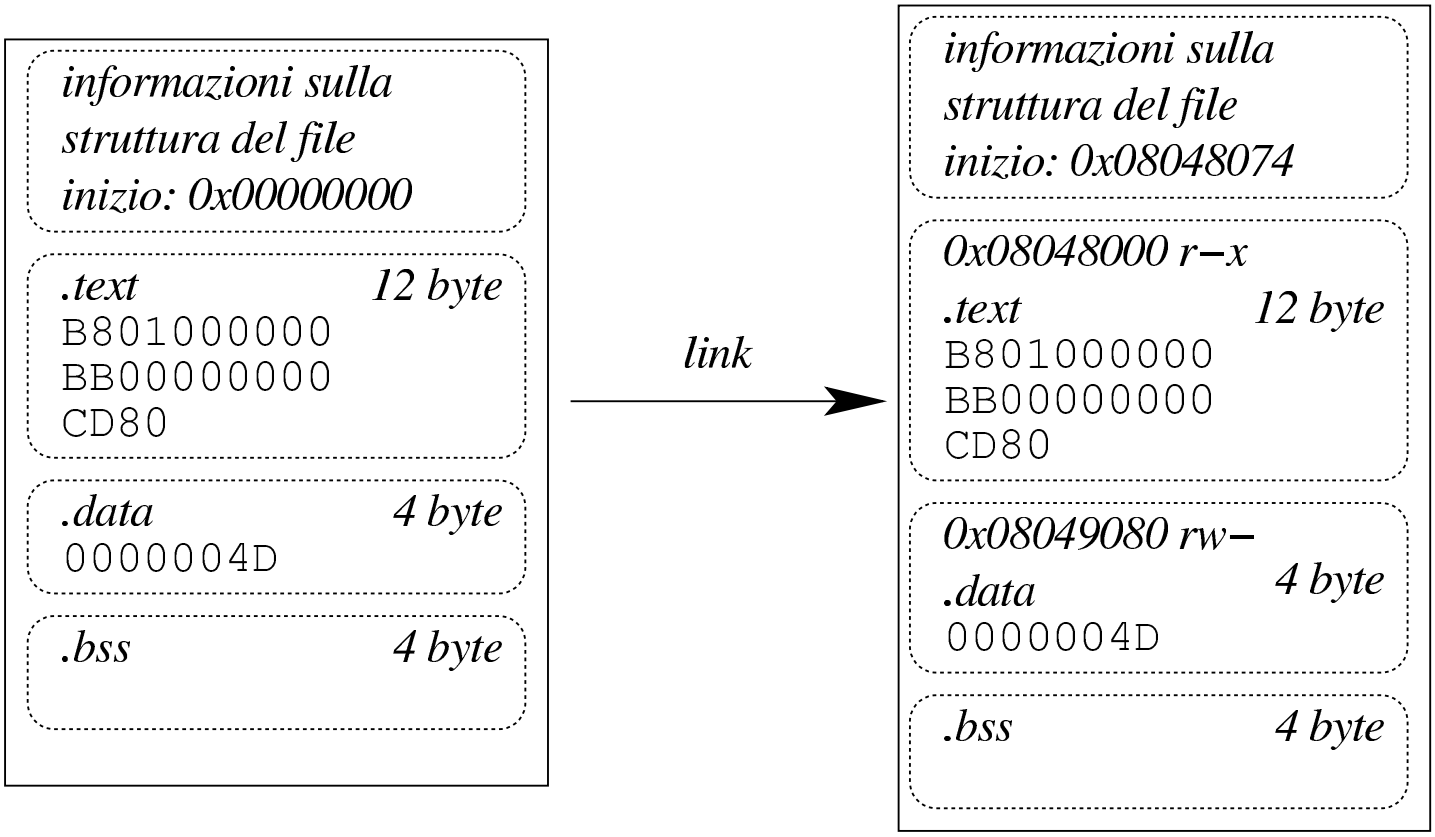
Per il momento, l'indirizzo iniziale di riferimento è 00000000_{16}.

Un file oggetto di questo tipo non può essere eseguito perché non contiene le informazioni necessarie al caricamento in memoria.

65.3.2 File eseguibile

Per ottenere un file eseguibile, i file oggetto che servono vengono raccolti da un collegatore (*link editor*) che riordina i vari componenti e produce un file con le informazioni necessarie al caricamento in memoria.

Figura 65.29. Dalle sezioni del file oggetto rilocabile ai segmenti del file oggetto eseguibile.



Continuando nell'ipotesi della sezione precedente, si passa a generare un file eseguibile a partire dal file oggetto precedente:

```
$ ld -o prg prg.o[Invio]
```

Con Objdump si può analizzare il contenuto del file eseguibile generato:

```
$ objdump -x prg[Invio]

prg:     file format elf32-i386
prg
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048074

Program Header:
```

```
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x00000080 memsz 0x00000080 flags r-x
    LOAD off    0x00000080 vaddr 0x08049080 paddr 0x08049080 align 2**12
         filesz 0x00000004 memsz 0x00000008 flags rw-

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000000c  08048074  08048074  00000074  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000004  08049080  08049080  00000080  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  08049084  08049084  00000084  2**2
                  ALLOC
SYMBOL TABLE:
08048074 l    d  .text  00000000 .text
08049080 l    d  .data  00000000 .data
08049084 l    d  .bss   00000000 .bss
00000000 l    d  *ABS*  00000000 .shstrtab
00000000 l    d  *ABS*  00000000 .symtab
00000000 l    d  *ABS*  00000000 .strtab
08049080 l       .data  00000000 aaa
08049084 l     O .bss   00000004 bbb
08048074 g       .text  00000000 _start
08049084 g       *ABS*  00000000 __bss_start
08049084 g       *ABS*  00000000 _edata
08049088 g       *ABS*  00000000 _end
```

Il file eseguibile è organizzato in ***segmenti*** che possono riferiti, ognuno, a una o più sezioni; ma il file può contenere anche sezioni che non sono riconducibili a un segmento. Il segmento, a differenza della sezione pura e semplice, deve descrivere in che modo il contenuto deve essere caricato in memoria e quali caratteristiche deve avere durante il funzionamento.

Dal rapporto generato da Objdump, precisamente nel riepilogo intitolato '**`Program Header`**', si può vedere cosa deve essere caricato in memoria e in quale posizione (gli indirizzi si riferiscono alla me-

moria virtuale). Si notano solo due segmenti, riferiti rispettivamente alla sezione '**.text**', contenente il codice da eseguire, e alla sezione '**.data**'.

Il segmento che riguarda la sezione '**.text**' deve essere caricato in memoria a partire dall'indirizzo 08048000_{16}, con permessi di lettura ed esecuzione; il segmento che si riferisce alla sezione '**.data**' deve essere caricato in memoria a partire dall'indirizzo 08049080_{16}, con permessi di lettura e scrittura. Non esiste un segmento per la sezione '**.bss**' in quanto non contiene dati e si sa comunque che deve essere allocata a partire dall'indirizzo 08049084_{16} (i permessi di lettura e scrittura sono impliciti in questo caso).

Quello che appare indicato come indirizzo iniziale è la posizione in cui si trova la prima istruzione da eseguire, pertanto è la posizione a cui deve passare il controllo il sistema di caricamento, dopo che è stata prodotta l'immagine del processo elaborativo in memoria. Questo indirizzo è interno al primo segmento, il quale è lungo 128_{10}byte (80_{16}byte), pertanto, tra l'indirizzo iniziale e quello ci devono essere delle informazioni amministrative, mentre nello spazio rimanente (esattamente 12 byte) ci sono le istruzioni vere e proprie.

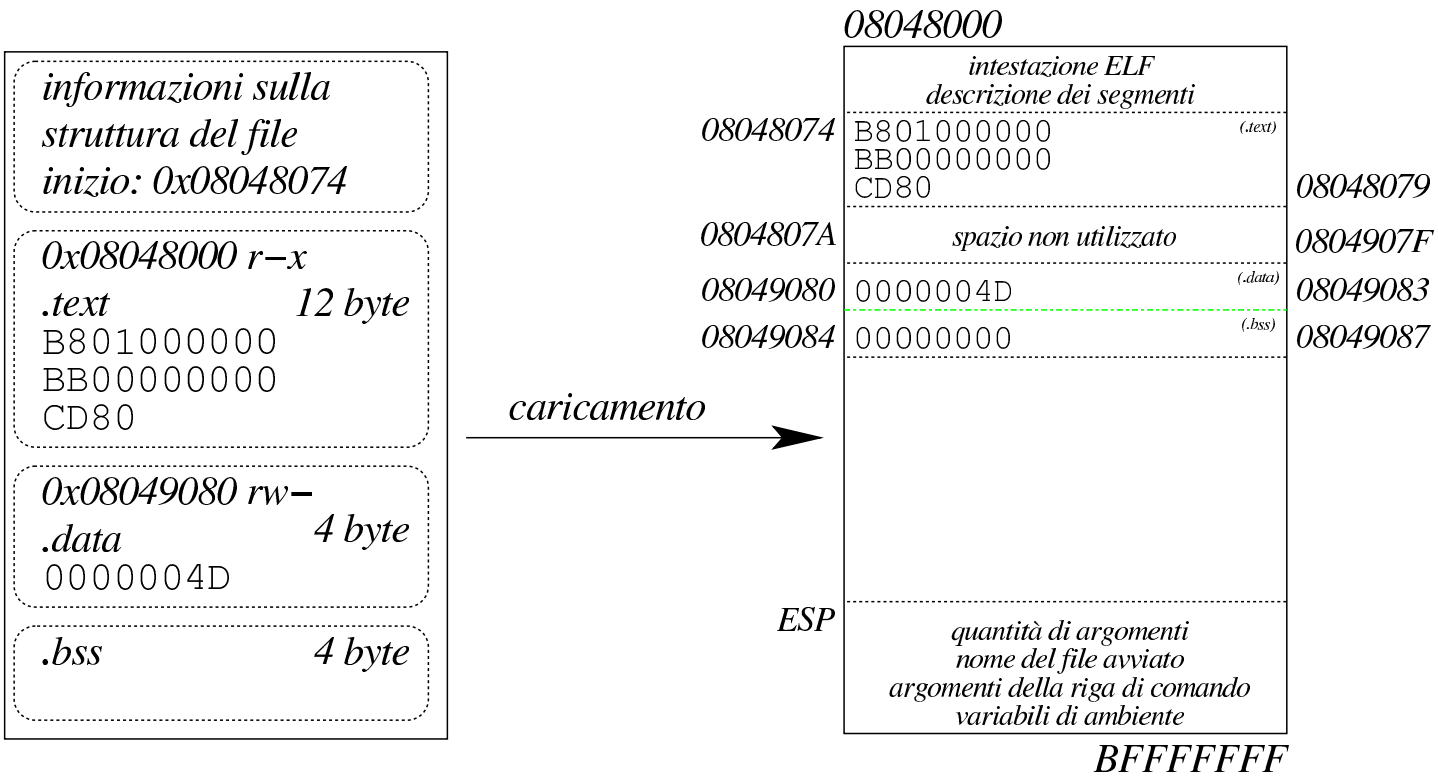
I 116 byte iniziali del primo segmento, di questo esempio, contengono precisamente l'intestazione ELF e la descrizione dei due segmenti del programma.

65.3.3 Immagine del processo nella memoria virtuale

Il sistema operativo legge il file eseguibile ed estrapola i segmenti, collocandoli in memoria, allocando anche lo spazio non inizializzato (privo pertanto di un segmento nel file eseguibile). Oltre a questo impila sul fondo le variabili di ambiente, gli argomenti della chiamata, il nome del file avviato effettivamente,... Per ultimo, su questa pila, mette la quantità di argomenti ricevuti nella riga di comando e lì posiziona il registro ***ESP***; successivamente, l'incremento di questo registro implica la crescita della pila dei dati.

In un sistema GNU/Linux i processi elaborativi utilizzano un'area della memoria virtuale che va da 08048000_{16} a $BFFFFFFF_{16}$, come se ognuno di questi disponesse della stessa dotazione di memoria e fosse sempre tutta propria. È il sistema operativo che crea questa astrazione e alloca o libera la memoria quando serve. Si osservi che lo spazio non allocato non può essere utilizzato e se il programma vi volesse fare riferimento (senza seguire la procedura prevista per l'allocazione) si otterrebbe un ***errore di segmentazione*** (*segmentation fault*).

Figura 65.31. Dal file oggetto eseguibile all'immagine del processo nella memoria virtuale.



Continuando nell'ipotesi delle sezioni precedenti, si può eseguire il programma sotto il controllo di GDB:

```
$ gdb prg [Invio]
```

Per fissare uno stop occorre indicare un indirizzo che punti almeno all'inizio della seconda istruzione (se si pretende di puntare alla prima istruzione, GDB poi non si ferma). Sapendo che la prima istruzione è all'indirizzo 08048074_{16} e che occupa cinque byte, si può usare l'indirizzo 08048079_{16} per indicare l'inizio della seconda:

```
(gdb) break *0x08048079 [Invio]
```

```
(gdb) run [Invio]
```

Si vuole verificare che i dati siano dove previsto:

```
(gdb) print (int)*0x08049080[Invio]

  $1 = 77

(gdb) print (int)*0x08049084[Invio]

  $2 = 0
```

Il secondo indirizzo fa riferimento a una memoria non inizializzata che viene posta inizialmente a zero; pertanto il risultato coincide con le previsioni. Si può verificare la presenza dell'intestazione ELF e della descrizione dei segmenti:

```
(gdb) print /x (char[116])*0x08048000[Invio]

  $3 = {0x7f, 0x45, 0x4c, 0x46, 0x1, 0x1, 0x1, 0x0, 0x0, 0x0,
  0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x2, 0x0, 0x3, 0x0, 0x1, 0x0,
  0x0, 0x0, 0x74, 0x80, 0x4, 0x8, 0x34, 0x0, 0x0, 0x0, 0xb0,
  0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x34, 0x0, 0x20, 0x0,
  0x2, 0x0, 0x28, 0x0, 0x7, 0x0, 0x4, 0x0, 0x1, 0x0, 0x0, 0x0,
  0x0, 0x0, 0x0, 0x0, 0x0, 0x80, 0x4, 0x8, 0x0, 0x80, 0x4,
  0x8, 0x80, 0x0, 0x0, 0x0, 0x80, 0x0, 0x0, 0x0, 0x5, 0x0,
  0x0, 0x0, 0x0, 0x10, 0x0, 0x0, 0x1, 0x0, 0x0, 0x0, 0x80,
  0x0, 0x0, 0x0, 0x80, 0x90, 0x4, 0x8, 0x80, 0x90, 0x4, 0x8,
  0x4, 0x0, 0x0, 0x0, 0x8, 0x0, 0x0, 0x0, 0x6, 0x0, 0x0, 0x0,
  0x0, 0x10, 0x0, 0x0}

(gdb) print (char[4])*0x08048000[Invio]

  $4 = "\177ELF"
```

Se si tenta di raggiungere un'area di memoria non allocata, si ottiene un errore:

```
(gdb) print /x (int)*0x0804A000[Invio]
```

```
Cannot access memory at address 0x804a000
```

Il registro ***ESP*** si trova effettivamente in una zona abbastanza profonda della memoria virtuale:

```
(gdb) info registers[Invio]

…
esp            0xbf87a540       0xbf87a540
…
(gdb) quit[Invio]
```

65.3.4 Allineamento dei segmenti in memoria

Riprendendo il rapporto generato da Objdump, va osservato che i segmenti da caricare in memoria sono «allineati»:

```
Program Header:
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x00000080 memsz 0x00000080 flags r-x
    LOAD off    0x00000080 vaddr 0x08049080 paddr 0x08049080 align 2**12
         filesz 0x00000004 memsz 0x00000008 flags rw-
```

È indicato che l'allineamento è da blocchi di 4096 byte (1000_{16}byte) ovvero 2^{12} byte. Ciò comporta un allontanamento significativo del secondo segmento dal primo (da 08048080_{16} che sarebbe il primo byte libero dopo il primo segmento, si salta a 08049080_{16}). Questo distacco lo produce il collegatore, o *link editor* (GNU LD), evidentemente per qualche motivo importante: in un sistema GNU/Linux la memoria virtuale è organizzata in pagine da 4 Kibyte (4096 byte) e non sarebbe possibile distinguere i permessi di accesso se i segmenti occupassero la stessa pagina.

È il caso di osservare che, nell'esempio mostrato, il distacco appare solo negli indirizzi che i segmenti devono prendere in memoria, perché nel file eseguibile, invece, sono collocati uno di seguito all'altro:

```
Program Header:
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x00000080 memsz 0x00000080 flags r-x
    LOAD off    0x00000080 vaddr 0x08049080 paddr 0x08049080 align 2**12
         filesz 0x00000004 memsz 0x00000008 flags rw-
```

65.3.5 Script per il collegamento

GNU LD,[1] ovvero il programma che si usa per collegare i file oggetto rilocabili, consente di definire la struttura del file eseguibile da generare, con un certo grado di dettaglio, attraverso quello che viene definito uno script (precisamente *link script* o *linker script*).

Esiste una configurazione predefinita di come deve essere realizzata la struttura del file eseguibile e la si può consultare con l'opzione '**`--verbose`**':

```
$ ld --verbose [Invio]
```

```
…
OUTPUT_FORMAT("elf32-i386", "elf32-i386",
              "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)
SEARCH_DIR("/usr/i486-linux-gnu/lib32"); SEARCH_DIR("/usr/local/lib32");…
SECTIONS
{
  /* Read-only sections, merged into text segment: */
  PROVIDE (__executable_start = 0x08048000); . = 0x08048000 + SIZEOF_HEADERS;
  .interp         : { *(.interp) }
  .hash           : { *(.hash) }
    …
    …
    …
```

```
    .debug_weaknames 0 : { *(.debug_weaknames) }
    .debug_funcnames 0 : { *(.debug_funcnames) }
    .debug_typenames 0 : { *(.debug_typenames) }
    .debug_varnames  0 : { *(.debug_varnames) }
    /DISCARD/ : { *(.note.GNU-stack) }
  }
```

Con l'opzione '**-T**' è possibile rimpiazzare completamente la configurazione predefinita, indicando lo script da caricare al suo posto. A titolo di esempio viene mostrato uno script molto semplificato che può essere usato con il programma apparso nelle sezioni precedenti, producendo un risultato simile:

```
ENTRY (_start)
SECTIONS {
    . = 0x08048000 + SIZEOF_HEADERS;
    .text . : { *(.text) }
    .data ALIGN (0x1000) : { *(.data) }
    .bss . : {
        *(.bss)
        *(COMMON)
    }
}
```

La direttiva che appare nella prima riga, dichiara il simbolo in corrispondenza del quale associare il punto di inizio; infatti, secondo le convenzioni comuni, il simbolo '**_start**' è quello che in un sorgente in linguaggio assemblatore segnala l'inizio del programma:

```
ENTRY (_start)
```

Successivamente appare un blocco, all'interno del quale si dichiara la configurazione delle sezioni del programma. La prima direttiva di questo blocco definisce l'indirizzo iniziale di riferimento, ottenuto sommando a 08048000_{16} la dimensione dell'intestazione (ov-

vero l'intestazione ELF vera e propria, assieme alla descrizione dei segmenti):

```
    . = 0x08048000 + SIZEOF_HEADERS;
```

Di seguito appare la descrizione delle tre sezioni tipiche: '**.text**', '**.data**' e '**.bss**'. La prima è la più semplice, in quanto si limita a dichiarare di collocare la sezione '**.text**' a partire dalla posizione corrente (quella raggiunta in quel punto), rappresentata da un punto singolo, '.', purché ci siano effettivamente sezioni con quel nome da collocare:

```
    .text . : { *(.text) }
```

In questo tipo di direttiva, il punto che rappresenta la posizione corrente è facoltativo (nel senso che può essere omesso); ciò che appare tra parentesi graffe è il contenuto che la nuova sezione '**.text**' deve avere e in questo caso rappresenta la somma delle sezioni '**.text**' dei file oggetto rilocabili. In particolare, l'asterisco iniziale serve a precisare che in mancanza di tali sezioni nei file oggetto rilocabili, non si deve creare la sezione corrispondente nel file eseguibile.

La sezione '**.data**' viene dichiarata in modo simile, con la differenza che, al posto del punto, viene indicato di spostare in avanti l'indirizzo in modo che sia un multiplo di 1000_{16}, ovvero di 4096_{10}. In questo modo si vuole ottenere che la sezione '**.data**' sia distanziata nel file eseguibile e che così distante sia anche il segmento caricato in memoria. In pratica, l'espressione '**ALIGN (0x1000)**' si traduce nel calcolo di un indirizzo adeguato all'allineamento che si intende ottenere, di 4096 byte:

```
    .data ALIGN (0x1000) : { *(.data) }
```

L'ultima sezione, '**.bss**', è un po' più articolata, in quanto prevede l'inclusione delle sezioni con lo stesso nome provenienti dai file oggetto rilocabili (se ce ne sono), con l'aggiunta di tutto ciò che costituisce dati non inizializzati, rappresentato dall'espressione '***(COMMON)**'.

Riprendendo il programma di esempio già visto nelle sezioni precedenti, ammesso che lo script appena descritto sia contenuto nel file '`config.ld`', il file oggetto '`prg.o`' potrebbe essere elaborato nel modo seguente:

```
$ ld -T config.ld -o prg prg.o[Invio]
```

Ecco cosa si può vedere con Objdump:

```
$ objdump -x prg[Invio]
```

```
prg:     file format elf32-i386
prg
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048074

Program Header:
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x00000080 memsz 0x00000080 flags r-x
    LOAD off    0x00001000 vaddr 0x08049000 paddr 0x08049000 align 2**12
         filesz 0x00000004 memsz 0x00000008 flags rw-

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000000c  08048074  08048074  00000074  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000004  08049000  08049000  00001000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  08049004  08049004  00001004  2**2
                  ALLOC
SYMBOL TABLE:
08048074 l    d  .text  00000000 .text
```

```
08049000 l    d  .data  00000000 .data
08049004 l    d  .bss   00000000 .bss
00000000 l    d  *ABS*  00000000 .shstrtab
00000000 l    d  *ABS*  00000000 .symtab
00000000 l    d  *ABS*  00000000 .strtab
08049000 l       .data  00000000 aaa
08049004 l     O .bss   00000004 bbb
08048074 g       .text  00000000 _start
```

Come si vede, questa volta il segmento riferito alla sezione '**.data**' parte esattamente da 08049000_{16}, ma così vale anche per la posizione della stessa sezione '**.data**' nel file eseguibile. In pratica, ciò comporta che il file eseguibile sia un po' più grande rispetto a prima, mentre l'utilizzo della memoria non cambia in modo sostanziale.

Sempre a titolo di esempio, si può provare a vedere cosa succede se si evita di allineare la sezione '**.data**':

```
ENTRY (_start)
SECTIONS {
    . = 0x08048000 + SIZEOF_HEADERS;
    .text . : { *(.text) }
    .data . : { *(.data) }
    .bss . : {
        *(.bss)
        *(COMMON)
    }
}
```

In questo modo, dato che il contenuto della sezione '**.text**' è molto breve, succede che il contenuto di tutte le sezioni finisce nello stesso segmento, il quale, di conseguenza, deve avere tutti i permessi necessari:

```
$ ld -T config.ld -o prg prg.o[Invio]
```

$ **objdump -x prg** [*Invio*]

```
prg:     file format elf32-i386
prg
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048074

Program Header:
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x00000084 memsz 0x00000088 flags rwx

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000000c  08048074  08048074  00000074  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000004  08048080  08048080  00000080  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  08048084  08048084  00000084  2**2
                  ALLOC
SYMBOL TABLE:
08048074 l    d  .text  00000000 .text
08048080 l    d  .data  00000000 .data
08048084 l    d  .bss   00000000 .bss
00000000 l    d  *ABS*  00000000 .shstrtab
00000000 l    d  *ABS*  00000000 .symtab
00000000 l    d  *ABS*  00000000 .strtab
08048080 l       .data  00000000 aaa
08048084 l     O .bss   00000004 bbb
08048074 g       .text  00000000 _start
```

Naturalmente, anche il file eseguibile torna a essere di dimensioni più piccole.

Sia chiaro che gli esempi di script apparsi qui non possono essere validi in generale, ma servono solo per comprendere a grandi linee il meccanismo. Per utilizzare seriamente questo strumento occorre prima uno studio approfondito del manuale di GNU LD.

65.3.6 Osservazioni sui simboli

I file oggetto, rilocabili o eseguibili, contengono un elenco di simboli, che Objdump raccoglie in una tabella, denominata '**SYMBOL TABLE**'. Vale la pena di confrontare tale tabella nelle varie situazioni descritte qui, come riepilogato nelle figure successive.

Figura 65.49. La tabella dei simboli nel file oggetto rilocabile prodotto dalla compilazione del file sorgente.

```
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l       .data  00000000 aaa
00000000 l     O .bss   00000004 bbb
00000000 g       .text  00000000 _start
```

Figura 65.50. La tabella dei simboli nel file oggetto eseguibile prodotto da GNU LD secondo la configurazione predefinita.

```
08048074 l    d  .text  00000000 .text
08049080 l    d  .data  00000000 .data
08049084 l    d  .bss   00000000 .bss
00000000 l    d  *ABS*  00000000 .shstrtab
00000000 l    d  *ABS*  00000000 .symtab
00000000 l    d  *ABS*  00000000 .strtab
08049080 l       .data  00000000 aaa
08049084 l     O .bss   00000004 bbb
08048074 g       .text  00000000 _start
08049084 g       *ABS*  00000000 __bss_start
08049084 g       *ABS*  00000000 _edata
08049088 g       *ABS*  00000000 _end
```

Figura 65.51. La tabella dei simboli nel file oggetto eseguibile prodotto da GNU LD secondo la configurazione predisposta nella sezione precedente.

```
08048074 l    d  .text  00000000 .text
08048080 l    d  .data  00000000 .data
08048084 l    d  .bss   00000000 .bss
00000000 l    d  *ABS*  00000000 .shstrtab
00000000 l    d  *ABS*  00000000 .symtab
00000000 l    d  *ABS*  00000000 .strtab
08048080 l       .data  00000000 aaa
08048084 l     O .bss   00000004 bbb
08048074 g       .text  00000000 _start
```

Si può osservare che, dopo la compilazione che produce un file oggetto rilocabile, appaiono gli stessi simboli previsti nel sorgente, con l'aggiunta di nomi corrispondenti a quelli delle sezioni. Nella trasformazione standard in file eseguibile, si vede la comparsa di altri simboli, in particolare: '**`.shstrtab`**', '**`.symtab`**',

‘**.strtab**’. Questi rappresentano la collocazione nel file di informazioni amministrative, relative al formato ELF. Inoltre, nel caso specifico dell’eseguibile generato secondo la configurazione predefinita di GNU LD, si vede la comparsa di simboli aggiuntivi che evidentemente dipendono dall’organizzazione della configurazione stessa.

Per comprendere come si possano inserire dei simboli addizionali attraverso lo script per GNU LD, si può riprendere l’esempio già visto nella sezione precedente, ritoccando leggermente la definizione della sezione ‘**.bss**’:

```
ENTRY (_start)
SECTIONS {
    . = 0x08048000 + SIZEOF_HEADERS;
    .text . : { *(.text) }
    .data ALIGN (0x1000) : { *(.data) }
    .bss . : {
        _sbss = .;
        *(.bss)
        *(COMMON)
        _ebss = .;
    }
}
```

I simboli che si vogliono aggiungere sono ‘**_sbss**’ e ‘**_ebss**’, con lo scopo di individuare l’inizio e la fine della nuova sezione ‘**.bss**’.

Figura 65.53. La tabella dei simboli dopo l'introduzione forzata di '**_sbss**' e '**_ebss**'.

```
SYMBOL TABLE:
08048074 l    d  .text  00000000 .text
08049000 l    d  .data  00000000 .data
08049004 l    d  .bss   00000000 .bss
00000000 l    d  *ABS*  00000000 .shstrtab
00000000 l    d  *ABS*  00000000 .symtab
00000000 l    d  *ABS*  00000000 .strtab
08049000 l       .data  00000000 aaa
08049004 l     O .bss   00000004 bbb
08049004 g       .bss   00000000 _sbss
08049008 g       .bss   00000000 _ebss
08048074 g       .text  00000000 _start
```

Eventualmente si può sperimentare cosa cambia nel contenuto dei file oggetto (rilocabili o eseguibili) quando si compila un sorgente con l'opzione '**--gstabs**' di GNU AS o con l'opzione '**-g**' di NASM.

65.3.7 Formati dei file oggetto

«

I file oggetto rilocabili e i file eseguibili possono essere realizzati secondo diversi formati, ma dipende dal sistema operativo qual è la scelta che si deve operare. Negli esempi mostrati, partendo dal presupposto di utilizzare un sistema GNU/Linux, si fa riferimento al formato ELF, in quanto è quello che deve essere usato e gli strumenti comuni sono già configurati per generare file conformi a tale standard.

Il formato del file che si deve produrre condiziona anche i tipi di sezioni che si possono dichiarare nel sorgente in linguaggio assemblatore. Il formato ELF dà molta libertà, comunque prevede una serie numerosa di sezioni con funzioni specifiche, in particolare

‘`.rodata`’ che comporta la creazione di un segmento di memoria con dati inizializzati, ma in sola lettura.

65.4 Formato ELF

Il formato ELF è il contenitore di un programma che non si trova necessariamente nello stato di poter essere eseguito. Il formato ELF si distingue per la presenza di un’intestazione che si trova obbligatoriamente all’inizio del file; quindi, il contenuto del file è affiancato da una serie di tabelle che lo descrivono in base a vari criteri.

65.4.1 Sezioni e segmenti

Per semplificare la descrizione di un formato ELF, lo si può immaginare composto da sezioni, il cui scopo è quello di descrivere tutto ciò che compone il programma, e da segmenti, con i quali si descrive in che modo il programma deve essere rappresentato in memoria ed eseguito. L’informazione relativa alle sezioni è indispensabile quando deve intervenire un «collegatore» (*linker*); l’informazione data dai segmenti riguarda l’avvio del programma.

Se si vuole abbandonare questo tipo di rappresentazione astratta, il formato ELF lo si può vedere come un involucro del codice eseguibile e dei dati inizializzati, contenente un’intestazione di riconoscimento (che si trova obbligatoriamente all’inizio del file) e da una serie di tabelle, più o meno concatenate tra di loro, alcune delle quali possono essere facoltative, in base al contesto per il quale il file oggetto è predisposto.

Tabella 65.54. Componenti principali che descrivono un formato ELF.

Tabella	Descrizione
ELF header	È l'intestazione del file e deve trovarsi necessariamente all'inizio dello stesso. Contiene poi i riferimenti alla tabella dei segmenti (*program header table*) e a quella delle sezioni (*section header table*).
program header table	È la tabella dei segmenti da caricare in memoria, con le informazioni necessarie a procedere in tal senso. La presenza di questa tabella è obbligatoria in un file oggetto eseguibile.
section header table	È la tabella delle sezioni.
string table	È la tabella delle stringhe, a cui fanno riferimento le altre tabelle quando devono indicare una stringa di qualunque tipo.
symbol table	È la tabella dei simboli associati a varie parti del contenuto. La tabella dei simboli, per indicare i nomi dei simboli, deve fare riferimento alla tabella delle stringhe.

65.4.2 Intestazione ELF

«

L'intestazione ELF è il componente più importante del formato, in quanto la sua presenta è obbligatoria. L'intestazione consente di identificare un file ELF come tale e di raggiungere le tabelle delle sezioni e dei segmenti, da cui poi si arriva al contenuto rimanente del file.

Tabella 65.55. Intestazione ELF secondo l'architettura x86, in particolare con le informazioni necessarie a produrre un file eseguibile.

Nome mnemonico	Dimensione x86-32	Dimensione x86-64	Descrizione
`e_ident[0]` `e_ident[1]` `e_ident[2]` `e_ident[3]`	8 bit 8 bit 8 bit 8 bit	8 bit 8 bit 8 bit 8 bit	Impronta di identificazione del formato: deve corrispondere a $7F_{16}$, '**`E`**', '**`L`**', '**`F`**'.
`e_ident[4]`	8 bit	8 bit	Definisce la classe del file: 01_{16} rappresenta un file oggetto a 32 bit; 02_{16} rappresenta invece un file a 64 bit.
`e_ident[5]`	8 bit	8 bit	Definisce la codifica dei dati: 01_{16} rappresenta un formato LSB, ovvero *little endian*; 02_{16} formato MSB, ovvero *big endian*. Sia 01_{16}, sia 02_{16}, si riferiscono a una rappresentazione numerica dei valori negativi attraverso il complemento a due.
`e_ident[6]`	8 bit	8 bit	Definisce la versione dell'intestazione (inizialmente esiste solo la versione 01_{16}).

Nome mnemonico	Dimensione x86-32	Dimensione x86-64	Descrizione
`e_ident[7]`	8 bit	8 bit	Questi byte definiscono informazioni di importanza minore e di solito vengono lasciati a 00_{16}.
`e_ident[8]`	8 bit	8 bit	
`e_ident[9]`	8 bit	8 bit	
`e_ident[10]`	8 bit	8 bit	
`e_ident[11]`	8 bit	8 bit	
`e_ident[12]`	8 bit	8 bit	
`e_ident[13]`	8 bit	8 bit	
`e_ident[14]`			
`e_ident[15]`	8 bit	8 bit	Dichiara la dimensione in byte della sequenza di identificazione. Il valore obbligato per questo byte è 10_{16}, ovvero 16_{10}.
`e_type`	16 bit	16 bit	Definisce il tipo di file oggetto. Un file oggetto rilocabile ha il codice 01_{16}; un file oggetto eseguibile ha il codice 02_{16}.
`e_machine`	16 bit	16 bit	Definisce il tipo di architettura. Il codice 03_{16} si riferisce al tipo Intel.
`e_version`	32 bit	32 bit	Definisce la versione del file oggetto (inizialmente esiste solo la versione 00000001_{16}).

Nome mnemonico	Dimensione x86-32	Dimensione x86-64	Descrizione
`e_entry`	32 bit	64 bit	Contiene l'indirizzo a cui occorre passare il controllo per l'esecuzione del programma.
`e_phoff`	32 bit	64 bit	*program header table offset* Contiene lo scostamento, rispetto all'inizio del file, necessario per raggiungere il primo byte della tabella che descrive i segmenti da caricare in memoria. Tale tabella è nota come *program header table* ed è obbligatoria la sua presenza in un file oggetto eseguibile.
`e_shoff`	32 bit	64 bit	*section header table offset* Contiene lo scostamento, rispetto all'inizio del file, necessario per raggiungere il primo byte della tabella che descrive le sezioni. Tale tabella è nota come *section header table*.
`e_flags`	32 bit	32 bit	Contiene degli indicatori specifici per il tipo di microprocessore. Nel caso dell'architettura x86-32 può contenere semplicemente valori a zero.
`e_ehsize`	16 bit	16 bit	*ELF header size* Contiene la dimensione dell'intestazione ELF.

Nome mnemonico	Dimensione x86-32	Dimensione x86-64	Descrizione
`e_phentsize`	16 bit	16 bit	*program header entry size* Definisce la dimensione di una voce descrittiva di un segmento, nella tabella dei segmenti. Tutte le voci di tale tabella hanno la stessa dimensione.
`e_phnum`	16 bit	16 bit	*program header number* Definisce la quantità di voci contenute nella tabella di descrizione dei segmenti.
`e_shentsize`	16 bit	16 bit	*section header entry size* Definisce la dimensione di una voce descrittiva di una sezione, nella tabella delle sezioni. Tutte le voci di tale tabella hanno la stessa dimensione.
`e_shnum`	16 bit	16 bit	*section header number* Definisce la quantità di voci contenute nella tabella di descrizione delle sezioni.
`e_shstrndx`	16 bit	16 bit	*section header string index* Definisce l'indice, all'interno della tabella delle sezioni, che identifica la voce che fa riferimento alla tabella delle stringhe.

65.4.3 Descrizione dei segmenti

La descrizione dei segmenti, necessaria per mettere in esecuzione un programma, è contenuta nella tabella *program header*, composta da un array di voci, di dimensione uniforme, ognuna delle quali descrive un segmento. Si raggiunge la prima voce di questo array con lo scostamento indicato nell'intestazione ELF ('**e_phoff**'), quindi, sapendo la dimensione di ogni voce ('**e_phentsize**') e la quantità di queste ('**e_phnum**'), è possibile scandire anche le altre.

Tabella 65.56. Descrizione di una voce nella tabella dei segmenti, secondo l'architettura x86-32.

Nome mnemonico	Dimensione	Descrizione
`p_type`	32 bit	Definisce il tipo di operazione da compiere. La situazione più semplice è costituita da codice eseguibile e dati da caricare in memoria: 01_{16}.
`p_offset`	32 bit	Definisce lo scostamento, dall'inizio del file, necessario a raggiungere il primo byte del segmento.
`p_vaddr`	32 bit	Definisce l'indirizzo assoluto, nell'ambito della memoria virtuale, dove il primo byte del segmento deve trovarsi in memoria, una volta caricato.
`p_paddr`	32 bit	Equivale al campo '**p_vaddr**', ma si riferisce alla «memoria fisica». In un sistema GNU/Linux questo valore è sempre uguale a '**p_vaddr**'.

Nome mnemonico	Dimensione	Descrizione
`p_filesz`	32 bit	Definisce la dimensione del segmento nel file e in casi particolari può essere pari a zero.
`p_memsz`	32 bit	Definisce la dimensione del segmento rappresentato in memoria e in casi particolari può essere pari a zero.
`p_flags`	32 bit	Definisce degli indicatori che descrivono i permessi del segmento: 1 = esecuzione; 2 = scrittura; 4 = lettura. Per avere permessi multipli si sommano i permessi elementari. Generalmente, il segmento di una porzione di codice dispone di permessi di accesso in lettura e in esecuzione, mentre quello di un'area di dati, consente normalmente la lettura e la scrittura.
`p_align`	32 bit	Definisce l'allineamento in memoria, a blocchi del valore indicato, il quale a sua volta deve essere una potenza di due.

Tabella 65.57. Descrizione di una voce nella tabella dei segmenti, secondo l'architettura x86-64.

Nome mnemonico	Dimensione	Descrizione
`p_type`	32 bit	Definisce il tipo di operazione da compiere. La situazione più semplice è costituita da codice eseguibile e dati da caricare in memoria: 01_{16}.
`p_flags`	32 bit	Definisce degli indicatori che descrivono i permessi del segmento: 1 = esecuzione; 2 = scrittura; 4 = lettura. Per avere permessi multipli si sommano i permessi elementari. Generalmente, il segmento di una porzione di codice dispone di permessi di accesso in lettura e in esecuzione, mentre quello di un'area di dati, consente normalmente la lettura e la scrittura.
`p_offset`	64 bit	Definisce lo scostamento, dall'inizio del file, necessario a raggiungere il primo byte del segmento.
`p_vaddr`	64 bit	Definisce l'indirizzo assoluto, nell'ambito della memoria virtuale, dove il primo byte del segmento deve trovarsi in memoria, una volta caricato.
`p_paddr`	64 bit	Equivale al campo '**`p_vaddr`**', ma si riferisce alla «memoria fisica». In un sistema GNU/Linux questo valore è sempre uguale a '**`p_vaddr`**'.

Nome mnemonico	Dimensione	Descrizione
`p_filesz`	64 bit	Definisce la dimensione del segmento nel file e in casi particolari può essere pari a zero.
`p_memsz`	64 bit	Definisce la dimensione del segmento rappresentato in memoria e in casi particolari può essere pari a zero.
`p_align`	64 bit	Definisce l'allineamento in memoria, a blocchi del valore indicato, il quale a sua volta deve essere una potenza di due.

65.4.4 Definizione manuale di un formato ELF

«

Un programma eseguibile in formato ELF deve contenere l'intestazione ELF e la descrizione dei segmenti; le sezioni possono anche mancare del tutto. Viene mostrato un esempio di programma banale (non fa altro che restituire il valore 77) in cui tutto, anche ciò che costituisce il formato ELF, viene definito nel sorgente. Il programma in sé trae spunto dal documento *A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux* di Brian Raiter, come annotato alla fine del capitolo. Si osservi che nel sorgente le sezioni non vengono indicate affatto.

```
.code32
.globl _start
#
file_begin:
#
# ELF header.
#
```

```
elf_header_begin:
    .byte  0x7F           # e_ident
    .byte  'E', 'L', 'F'  #
    .byte  1              #
    .byte  1              #
    .byte  1              #
    .byte  0, 0, 0, 0     #
    .byte  0, 0, 0, 0     #
    .byte  16             #
    #
    .short 2      # e_type     2 = executable file
    .short 3      # e_machine  3 = 386
    .int   1      # e_version  1 = current version
    .int   _start # e_entry        start address
    .int   (program_header_begin - file_begin)
                  # e_phoff        program header offset
    .int   0      # e_shoff    0 = no section header table
    .int   0      # e_flags        no flags
    .short (elf_header_end - elf_header_begin)
                  # e_ehsize       ELF header size
    .short (program_header_end - program_header_begin)
                  # e_phentsize    program header entry size
    .short 1      # e_phnum        program header entries
    .short 0      # e_shentsize 0 = no section header table
    .short 0      # e_shnum        section header entries
    .short 0      # e_shstrndx  0 = undefined
elf_header_end:
#
# Program header table, with just one entry.
#
program_header_begin:
    .int   1          # p_type  1 = segment to be loaded
    .int   0          # p_offset    segment's offset
    .int   0x08048000 # p_vaddr     segment's virtual
```

```
                        #               address
    .int    0x08048000 # p_paddr      segment's physical
                        #               address
    .int    (file_end - file_begin)
                        # p_filesz     file image size
    .int    (file_end - file_begin)
                        # p_memsz      memory image size
    .int    5           # p_flags 5 = read + execute
    .int    0x1000      # p_align      segment's memory
                        #               alignment
program_header_end:
#
# Program code.
#
_start:
    mov $77, %ebx
    mov $1, %eax
    int $0x80
#
file_end:
```

Il sorgente scritto nel formato adatto a GNU AS va compilato così:

```
$ as -o elf_test.o elf_test.s[Invio]
```

```
$ ld --oformat binary -o elf_test elf_test.o[Invio]
```

Come si vede, GNU LD viene usato in modo da produrre un formato binario, puro e semplice, ovvero un file privo di formato, dato che è già tutto incluso nella descrizione del programma stesso.

Si mostra anche il sorgente adatto a NASM, dove va annotata anche l'origine, ovvero l'indirizzo in cui tutto deve essere caricato in memoria:

```
    bits 32
    org  0x08048000
;
file_begin:
;
; ELF header.
;
elf_header_begin:
    db 0x7F             ; e_ident
    db 'E', 'L', 'F' ;
    db 1                ;
    db 1                ;
    db 1                ;
    db 0, 0, 0, 0       ;
    db 0, 0, 0, 0       ;
    db 16               ;
    ;
    dw 2            ; e_type      2 = executable file
    dw 3            ; e_machine   3 = 386
    dd 1            ; e_version   1 = current version
    dd _start       ; e_entry         start address
    dd (program_header_begin - file_begin)
                    ; e_phoff         program header offset
    dd 0            ; e_shoff     0 = no section header table
    dd 0            ; e_flags         no flags
    dw (elf_header_end - elf_header_begin)
                    ; e_ehsize        ELF header size
    dw (program_header_end - program_header_begin)
                    ; e_phentsize     program header entry
                    ;                 size
    dw 1            ; e_phnum         program header entries
    dw 0            ; e_shentsize 0 = no section header table
    dw 0            ; e_shnum         section header entries
    dw 0            ; e_shstrndx  0 = undefined
```

```
elf_header_end:
;
; Program header table, with just one entry.
;
program_header_begin:
    dd 1            ; p_type     1 = segment to be loaded
    dd 0            ; p_offset       segment's offset
    dd 0x08048000   ; p_vaddr        segment's virtual
                    ;                address
    dd 0x08048000   ; p_paddr        segment's physical
                    ;                address
    dd (file_end - file_begin)
                    ; p_filesz       file image size
    dd (file_end - file_begin)
                    ; p_memsz        memory image size
    dd 5            ; p_flags    5 = 1 (execute) + 4 (read)
    dd 0x1000       ; p_align        segment's memory
                    ;                 alignment
program_header_end:
;
; Program code.
;
_start:
    mov ebx, 77
    mov eax, 1
    int 0x80
;
file_end:
```

In questo caso, la compilazione non richiede altro che NASM, il quale produce direttamente il formato binario voluto:

```
$ nasm -f bin -o elf_test elf_test.s[Invio]
```

```
$ chmod +x elf_test [Invio]
```

I valori che rappresentano scostamenti e dimensioni del codice, sono calcolati attraverso il compilatore, facendo riferimento ai simboli rappresentati dalle etichette che delimitano le varie porzioni del sorgente. Ecco come si presenta il programma eseguibile dal punto di vista di Objdump:

```
$ objdump -x elf_test [Invio]

  ELF_test:     file format elf32-i386
  ELF_test
  architecture: i386, flags 0x00000102:
  EXEC_P, D_PAGED
  start address 0x08048054

  Program Header:
      LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
           filesz 0x00000060 memsz 0x00000060 flags r-x

  Sections:
  Idx Name          Size      VMA       LMA       File off  Algn
  SYMBOL TABLE:
  no symbols
```

65.4.5 Esempio più complesso

Viene mostrato un esempio più complesso, composto sempre da una sola voce nella tabella dei segmenti; in particolare viene definita una variabile inizializzata, incorporata nel segmento del codice, che nel sorgente appare in fondo. Viene mostrata solo la versione per GNU AS, trattandosi del programma per il calcolo del fattoriale, già descritto in precedenza.

```
.code32
.globl _start
#
```

```
file_begin:
#
# ELF header.
#
elf_header_begin:
    .byte  0x7F           # e_ident
    .byte  'E', 'L', 'F' #
    .byte  1              #
    .byte  1              #
    .byte  1              #
    .byte  0, 0, 0, 0     #
    .byte  0, 0, 0, 0     #
    .byte  16             #
    #
    .short 2        # e_type      2 = executable file
    .short 3        # e_machine   3 = 386
    .int   1        # e_version   1 = current version
    .int   _start   # e_entry         start address
    .int   (program_header_begin - file_begin)
                    # e_phoff         program header offset
    .int   0        # e_shoff     0 = no section header table
    .int   0        # e_flags         no flags
    .short (elf_header_end - elf_header_begin)
                    # e_ehsize        ELF header size
    .short (program_header_end - program_header_begin)
                    # e_phentsize     program header entry
                    #                 size
    .short 1        # e_phnum         program header entries
    .short 0        # e_shentsize 0 = no section header table
    .short 0        # e_shnum         section header entries
    .short 0        # e_shstrndx  0 = undefined
elf_header_end:
#
# Program header table, with just one entry.
```

```
#
program_header_begin:
    .int 1          # p_type  1 = segment to be loaded
    .int 0          # p_offset    segment's offset
    .int 0x08048000 # p_vaddr     segment's virtual address
    .int 0x08048000 # p_paddr     segment's physical address
    .int (file_end - file_begin)
                    # p_filesz    file image size
    .int (file_end - file_begin)
                    # p_memsz     memory image size
    .int 5          # p_flags 5 = read + execute
    .int 0x1000     # p_align     segment's memory alignment
program_header_end:
#
# Program code.
#
_start:
    mov  op1, %esi  # ESI contiene il valore di cui si vuole
                    # calcolare il fattoriale.
    push %esi       # f_fact (ESI) ==> EAX
    call f_fact     #
    add  $4, %esp   #
    mov  %eax, %ebx # Restituisce il valore del fattoriale,
    mov  $1, %eax   # ammesso che sia abbastanza piccolo
    int  $0x80      # da poter essere rappresentato come
                    # valore di uscita.
#
# Fattoriale di un numero senza segno.
# f_fatt (a) ==> EAX
# EAX = a!
#
f_fact:
    enter $4, $0
    pusha
```

```
    #
    mov  8(%ebp), %edi # Valore di cui calcolare il
                       # fattoriale.
    #
    cmp  $1, %edi      # Il fattoriale di 1 è 1.
    jz   f_fact_end_1  #
    #
    mov  %edi, %esi    # ESI contiene il valore di cui si
    dec  %esi          # vuole il fattoriale, ridotto di
                       # una unità.
    #
    push %esi          # f_fact (ESI) ==> EAX
    call f_fact        #
    add  $4, %esp      #
    mul  %edi          # EDX:EAX = EAX*EDI
    mov  %eax, -4(%ebp)# Salva il risultato.
    jmp  f_fact_end_X  # Conclude la funzione.
    #
f_fact_end_1:
    popa               # Conclude la funzione con EAX = 1.
    mov $1, %eax       #
    leave              #
    ret                #
f_fact_end_X:
    popa               # Conclude la funzione con EAX pari
    mov -4(%ebp), %eax # al valore salvato nella variabile
    leave              # locale.
    ret                #
#
# Initialized data.
#
op1:    .int    5
#
file_end:
```

65.5 Programmi completamente autonomi

A volte esiste la necessità di realizzare un programma funzionante in modo autonomo, ovvero *stand alone*, senza il sostegno del sistema operativo. Un lavoro di questo tipo richiede lo studio delle stesse problematiche che riguardano inizialmente la costruzione di un nuovo sistema operativo, ma di norma è meglio fermarsi alla produzione di un programma singolo.[2]

C'è da osservare che l'avvio di un programma «autonomo» in un elaboratore x86 può essere di una complessità mostruosa, a causa di problematiche ereditate dall'architettura originale del microprocessore 8088/8086. La prima difficoltà che si incontra a tale proposito sta nel far sì che il microprocessore si metta a lavorare in «modalità protetta», ovvero in una condizione che consenta di utilizzare in modo ragionevole la memoria centrale. Nel tempo, questa e altre questioni sono diventate di competenza dei programmi che si occupano di avviare un sistema operativo, come è il caso di GRUB 1 e di SYSLINUX, che così predispongono un contesto più confortevole al programma o al kernel da avviare successivamente.

Qui si mostrano esempi che utilizzano anche codice in linguaggio C, il quale viene descritto a partire dal capitolo 66.

65.5.1 Le specifiche «multiboot»

Le specifiche *multiboot* sono definite dal documento *Multiboot specification*, disponibile presso *http://www.gnu.org/software/grub/manual/multiboot/* . Si tratta della definizione di un'interfaccia tra sistema di avvio e sistema operativo (o programma autonomo), inizialmente per un'architettura x86. Il documento citato contiene sia

le specifiche, sia un esempio completo di programma che interagisce con il sistema di avvio secondo le specifiche stesse. Qui si riassumono i concetti principali.

65.5.1.1 Formato del file che deve essere avviato

Il file-immagine che contiene il programma da avviare (programma che potrebbe essere il kernel di un sistema operativo), deve contenere un'intestazione particolare, definita *multiboot header*, costituita in pratica da un'impronta di riconoscimento e da una serie di dati. Attraverso questa intestazione, il sistema di avvio è almeno in grado di riconoscere il file-immagine come qualcosa che deve essere avviato effettivamente e di recepirne le caratteristiche.

Questa intestazione deve trovarsi nella parte iniziale del file-immagine da caricare ed eseguire, ma non è necessario che sia esattamente all'inizio dello stesso, essendo sufficiente che sia contenuta completamente entro i primi 8 Kibyte.

Figura 65.62. La prima parte obbligatoria dell'intestazione.

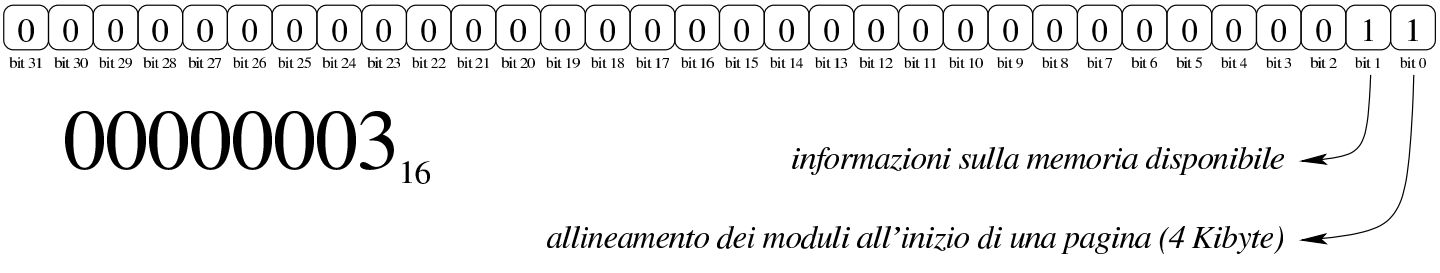
Il primo campo da 32 bit, definito ***magic***, contiene un'impronta di riconoscimento, costituita precisamente dal numero $1BADB002_{16}$. Questa serve al sistema di avvio a individuare la presenza e l'ini-

zio di una tale intestazione. Il secondo campo da 32 bit, definito ***flags***, contiene degli indicatori con i quali si richiede un certo comportamento al sistema di avvio. Il terzo campo da 32 bit, definito ***checksum***, contiene un numero calcolato in modo tale che la somma tra i numeri contenuti nei tre campi da 32 bit porti a ottenere zero, senza considerare i riporti.

I nomi indicati sono quelli definiti dallo standard e, come si vede, il campo ***checksum*** si ottiene calcolando $-(\textbf{\textit{magic}} + \textbf{\textit{flags}})$, dove si deve intendere che i calcoli avvengono con valori interi senza segno e si ignorano i riporti.

Se il file-immagine da avviare è informato ELF, le informazioni che il sistema di avvio necessita per piazzarlo correttamente in memoria e per passare il controllo allo stesso, sono già disponibili e non c'è la necessità di occuparsi di altri campi facoltativi che possono seguire i tre già descritti. Stante questa semplificazione, per quanto riguarda il campo ***flags*** sono importanti i primi due bit, mentre gli altri vanno lasciati a zero.

Figura 65.63. Il campo ***flags*** e il suo utilizzo fondamentale.

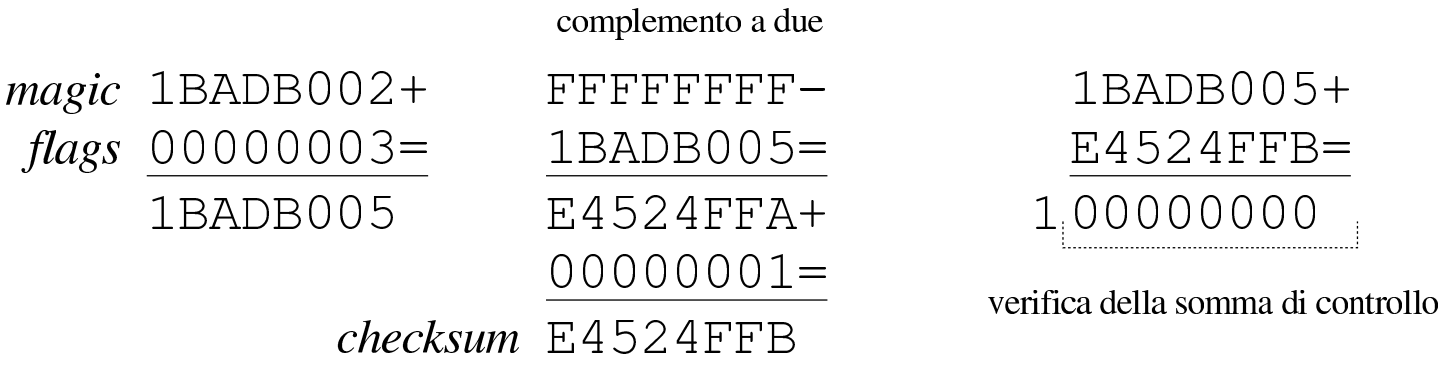


Il bit meno significativo del campo ***flags***, se impostato a uno, serve a richiedere il caricamento in memoria dei moduli eventuali (assieme al file-immagine principale) in modo che risultino allineati all'inizio di una «pagina» (ovvero all'inizio di un blocco da 4 Kibyte). Alcuni

sistemi operativi hanno la necessità di trovare i moduli allineati in questo modo e in generale l'attivazione di tale bit non può creare danno.

Il secondo bit del campo ***flags*** serve a richiedere al sistema di avvio di passare le informazioni disponibili sulla memoria. Queste informazioni vengono rese disponibili a partire da un'area a cui punta inizialmente il registro ***EBX***. In generale si tratta di un'informazione utile (che al massimo può essere ignorata), pertanto conviene attivare anche questo bit.

Figura 65.64. Calcolo del campo ***checksum***.



65.5.1.2 Situazione dopo l'avvio del file-immagine

Quando, dopo il trasferimento in memoria del programma, il sistema di avvio passa il controllo allo stesso, la situazione che questo programma si trova è sostanzialmente quella seguente, dove però sono stati omessi molti dettagli importanti:

- il microprocessore è in modalità protetta;
- il registro ***EAX*** contiene il numero $2BADB002_{16}$ (si osservi che la prima cifra è cambiata, rispetto all'impronta che deve avere il file-immagine da avviare);

- il registro ***EBX*** deve contenere l'indirizzo fisico, a 32 bit, di una serie di campi contenenti informazioni passate dal sistema di avvio (*multiboot information structure*).

Di norma, la prima cosa che fa il programma che è stato avviato in questo modo è di azzerare il registro ***EFLAGS*** e di predisporre uno spazio per la pila dei dati, posizionando il registro ***ESP*** di conseguenza.

65.5.1.3 Informazioni passate dal sistema di avvio al programma

Il sistema di avvio conforme alle specifiche *multiboot* offre una serie di informazioni, collocate in una struttura che parte dall'indirizzo indicato nel registro ***EBX***. Questa struttura ha un certo grado di complessità, in quanto può fare riferimento ad altre strutture. Qui viene descritta brevemente solo una prima porzione di questa struttura; per l'approfondimento occorre consultare le specifiche, pubblicate presso *http://www.gnu.org/software/grub/manual/multiboot/* .

Figura 65.65. Inizio della struttura informativa offerta da un sistema di avvio aderente alle specifiche *multiboot*.

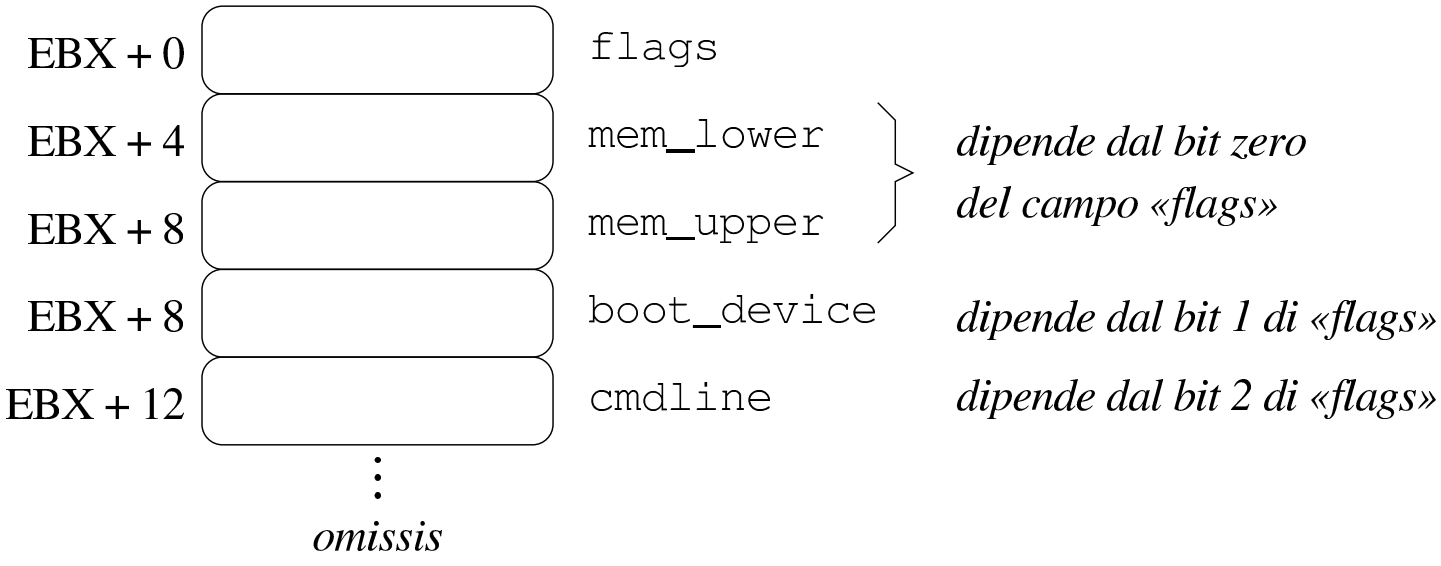


Tabella 65.66. Descrizione dei primi campi della struttura informativa fornita dal sistema di avvio *multiboot*.

Nome mnemonico del campo	bit del campo '**flags**' da cui dipende	Descrizione
`flags`		Il primo campo definisce una serie di indicatori, con i quali si dichiara se una certa informazione, successiva, viene fornita ed è valida.

Nome mnemonico del campo	bit del campo '**flags**' da cui dipende	Descrizione
`mem_lower` `mem_upper`	0	Se è attivo il bit meno significativo del campo '**flags**', i campi '**mem_lower**' e '**mem_upper**' contengono la dimensione della memoria bassa (da zero a un massimo di 640 Kibyte) e della memoria alta (quella che si trova a partire da un mebibyte). La dimensione è da intendersi in kibibyte (simbolo Kibyte) e, per quanto riguarda la memoria alta, viene indicata solo la dimensione continua fino al primo «buco».
`boot_device`	1	Se è attivo il secondo bit, partendo dal lato meno significativo, il campo '**boot_device**' dà informazioni sull'unità di avvio. L'informazione è divisa in quattro byte, come descritto nelle specifiche *multiboot*.
`cmdline`	2	Se è attivo il terzo bit, partendo dal lato meno significativo, il campo '**cmdline**' contiene l'indirizzo iniziale di una stringa che riproduce la riga di comando passata al kernel.

Come si può intuire leggendo la tabella che descrive i primi cinque

campi, il significato dei bit del campo '**flags**' viene attribuito, mano a mano che l'aggiornamento delle specifiche prevede l'espansione della struttura informativa. Per esempio, un campo '**flags**' con il valore 100_2 sta a significare che esistono i campi fino a '**cmdline**' e il contenuto di quelli precedenti non è valido, ma i campi successivi, non esistono affatto. La comprensione di questo concetto dovrebbe rendere un po' più semplice la lettura delle specifiche.

65.5.2 Esempio di programma da avviare secondo le specifiche «multiboot»

«

I listati seguenti mostrano il contenuto dei file necessari a produrre un programma da avviare secondo le specifiche *multiboot*. Per la precisione, il programma non fa alcunché e serve solo come base di partenza per lo sviluppo di qualcosa di più complesso, con l'ausilio del linguaggio C. I file mostrati hanno, nell'ordine di apparizione, i nomi: '`loader.s`', '`kernel.c`', '`linker.ld`' e '`Makefile`'.

Listato 65.67. File '`loader.s`' usato per la prima parte del codice, contenente l'intestazione *multiboot* e la preparazione dell'ambiente minimo di funzionamento, compresa la collocazione della pila dei dati. Il programma chiama la funzione '**_kernel**' presente nel file '`kernel.c`', passando come parametri il codice di riconoscimento del sistema di avvio e il puntatore alle altre informazioni che questo può fornire. Al ritorno dalla chiamata della funzione, il programma tenta di arrestare il microprocessore, ma se non ci riesce si mette in un ciclo senza fine che produce apparentemente lo stesso risultato.

```
.globl  _loader
.extern _kernel
#
```

```
# Dimensione della pila interna al kernel. Qui vengono
# previsti 16384 elementi (0x4000) da 32 bit, pari a 65536
# byte.
#
.equ STACK_SIZE,  0x4000
#
# Si inizia subito con il codice che si mescola con i dati.
#
_loader:
    jmp boot    # Salta all'inizio del codice.
    .align 4    # Fa in modo di riempire lo spazio mancante
                # al completamento di un blocco di 4 byte.
#
# Intestazione «multiboot», poco dopo l'inizio del
# file-immagine.
#
multiboot_header:
    .int 0x1BADB002                     # magic
    .int 0x00000003                     # flags
    .int -(0x1BADB002 + 0x00000003)     # checksum
#
# Inizia il codice di avvio.
#
boot:
    #
    # Regola ESP alla base della pila.
    #
    movl $(stack_max + STACK_SIZE), %esp
    #
    # Azzera gli indicatori (e per questo usa la pila appena
    # sistemata).
    #
    pushl $0
    popf
```

```
    #
    # Chiama il kernel scritto in C, passandogli le
    # informazioni ottenute dal sistema di avvio.
    #
    # void _kernel (unsigned int magic,
    #               void *multiboot_info)
    #
    pushl %ebx   # Puntatore alla struttura contenente le
                 # informazioni passate dal sistema di
                 # avvio.
    pushl %eax   # Codice di riconoscimento del sistema di
                 # avvio.
    #
    call _kernel # Chiama la funzione _kernel()
    #
halt:
    hlt          # Se il kernel termina, ferma il
                 # microprocessore.
    jmp halt     # Se non si è fermato, crea un ciclo
                 # senza fine.
#
# Alla fine del programma, viene collocato lo spazio per la
# pila dei dati, senza inizializzarlo. Per scrupolo si
# allinea ai 4 byte (32 bit).
#
.align 4
.comm stack_max, STACK_SIZE
#
```

Listato 65.68. File ‘`kernel.c`’ che potrebbe contenere idealmente il kernel di un piccolo sistema operativo. In questo caso il programma non fa alcunché e ignora anche la presenza di parametri nella chiamata.

```
void _kernel(void)
{
    ;
}
```

Listato 65.69. File ‘`linker.ld`’, da usare come script per GNU LD. Si può osservare che la sezione ‘**`.data`**’ viene distanziata da ‘**`.text`**’ e ‘**`.rodata`**’, in quanto si deve collocare in una pagina di memoria differente, per poter limitare i permessi di accesso in scrittura ai soli dati variabili.

```
ENTRY (_loader)
SECTIONS {
    . = 0x00100000;
    .text : { *(.text) }
    .rodata : { *(.rodata) }
    .data ALIGN (0x1000) : { *(.data) }
    .bss : {
        _sbss = .;
        *(.bss)
        *(COMMON)
        _ebss = .;
    }
}
```

Listato 65.70. File ‘Makefile’ da usare per la compilazione.

```
all: loader kernel link
#
clean:
        rm *.o
        rm kernel
#
loader:
        as -o loader.o loader.s
#
kernel:
        gcc -Wall -Werror -o kernel.o -c kernel.c \
            -nostdlib -nostartfiles -nodefaultlibs
#
link:
        ld --script=linker.ld -o kernel loader.o kernel.o
```

Per avviare il programma che si ottiene, si può usare GRUB 1, utilizzando le direttive seguenti nel suo file di configurazione:

```
…
title  mio kernel
kernel (fd0)/kernel
…
```

In questo caso si suppone di utilizzare un dischetto per l’avvio e che il file da avviare sia ‘kernel’, contenuto proprio nella radice.

65.5.3 Visualizzazione di messaggi

«

Quando si scrive un programma autonomo, come descritto sinteticamente nella sezione precedente, occorre considerare che il linguaggio C non può essere usato sfruttando le librerie consuete, pertan-

to occorre produrre tutto internamente, anche le funzioni per la visualizzazione dei messaggi. I listati seguenti vanno a sostituire il file ‘`kernel.c`’ della sezione precedente, allo scopo di visualizzare qualcosa sullo schermo (il contenuto dei primi campi della struttura informativa creata dal sistema di avvio), attraverso delle funzioni elementari definite internamente.

Per visualizzare un messaggio sullo schermo di un elaboratore x86 è necessario scrivere in una porzione di memoria che parte dall'indirizzo $B8000_{16}$ (a partire da 736 Kibyte), utilizzando coppie di byte, dove il primo byte è un codice che descrive i colori da usare per il carattere e il suo sfondo, mentre il secondo contiene il carattere da visualizzare. Nel programma il codice in questione è 07_{16} che mostra un carattere bianco su sfondo nero. Ciò che si deve osservare è che il programma tratta la coppia di byte come un numero a 16 bit, nel quale il carattere e il suo codice di visualizzazione sembrano invertiti, a causa del fatto che l'architettura è di tipo *little endian*.

Listato 65.72. File ‘`kernel.c`’ che include automaticamente i file ‘`display.c`’ e ‘`multiboot.c`’.

```
#include "display.c"
#include "multiboot.c"
//
//
//
void
_kernel (unsigned long magic, type_multiboot_info *info)
{
  clear_screen ();
  //
  print_string ("Salve!\n\0");
  //
```

```
  if (magic == 0x2BADB002)
    {
      print_string ("Sono stato avviato attraverso un \0");
      print_string ("sistema di avvio aderente alle \0");
      print_string ("specifiche \n\0");
      print_string ("\"multiboot\". Ecco solo alcune \0");
      print_string ("informazioni:\n\0");
      multiboot_information (info);
    }
  else
    {
      print_string ("Sono stato avviato attraverso un \0");
      print_string ("sistema di avvio che non e` \0");
      print_string ("conforme alle specifiche\n\0");
      print_string ("\"multiboot\".\n\0");
    }
}
```

Listato 65.73. File ‘display.c’, contenente le funzioni necessarie a visualizzare stringhe e numeri in forma di stringa.

```
static unsigned short *Screen = (unsigned short *) 0xB8000;
//
static const unsigned int Rows = 25, Columns = 80;
static       unsigned int Row = 0, Column = 0;
static       unsigned char Attrib = 0x07;
//
//
//
static unsigned short
screen_cell (unsigned char c, unsigned char attrib)
{
  //
  // Assembla i due caratteri in un numero a 16 bit.
  //
```

```
  return (short) c | (((short) attrib) * 0x100);
}
//
//
//
static void
clear_screen (void)
{
  unsigned int i;
  //
  for (i = 0; i < (Rows * Columns) ; i++)
    {
      //
      // Scrive uno spazio nella posizione.
      //
      *(Screen + i) = screen_cell (0x20, Attrib);
    }
}
//
//
//
static void
new_line (void)
{
  int i, j;
  //
  Column = 0;
  Row++;
  //
  if (Row >= Rows)
    {
      //
      // Copia il testo in su.
      //
```

```
      for (i = 0; i < (Rows - 1) * Columns ; i++)
        {
          j = i + Columns;
          //
          // Trascrive la cella della riga successiva.
          //
          *(Screen + i) = *(Screen + j);
        }
      //
      // Mette l'indice di riga nell'ultima posizione.
      //
      Row = Rows - 1;
      //
      // Pulisce la riga alla base dello schermo.
      //
      for (i = ((Rows - 1) * Columns) ;
           i < (Rows * Columns) ;
           i++)
        {
          //
          // Cancella la cella dello schermo.
          //
          *(Screen + i) = screen_cell (0x20, Attrib);
        }
    }
}
//
//
//
static void
print_char (unsigned char c)
{
  //
  // Put the character.
```

```
  //
  if (c == '\n' || c == '\r')
    {
      new_line ();
    }
  else
    {
      *(Screen + (Row * Columns + Column))
       = screen_cell (c, Attrib);
      //
      // Move cursor.
      //
      Column++;
      if (Column >= Columns)
        {
          new_line ();
        }
    }
}
//
//
//
static void
print_string (char *string)
{
  unsigned int i;
  //
  for (i = 0; i < 100000 ; i++)
    {
      if (string[i] != 0)
        {
          print_char (string[i]);
        }
      else
```

```
        {
          break;
        }
    }
}
//
//
//
static void
reverse_string (char *string)
{
  unsigned int i, j;
  unsigned char c;
  //
  // Scandisce la stringa alla ricerca del valore a zero.
  //
  for (i = 0; string[i] != 0; i++)
    {
      ;
    }
  //
  // L'indice "i" punta alla cella a zero.
  // Viene rimesso l'indice "i" in modo da puntare
  // all'ultimo carattere.
  //
  i--;
  //
  // Si inverte l'ordine delle cifre.
  //
  for (j = 0; j < i; j++, i--)
    {
      c = string[i];
      string[i] = string[j];
      string[j] = c;
```

```
    }
  //
}
//
//
//
static void
num_to_string (unsigned long num, unsigned int base,
               char *string)
{
  unsigned int i;
  unsigned char remainder;
  //
  if (num == 0)
    {
      string[0] = '0';
      string[1] = 0;
      return;
    }
  //
  for (i = 0; num != 0; i++)
    {
      remainder = num % base;
      num = num / base;
      //
      if (remainder <= 9)
        {
          string[i] = '0' + remainder;
        }
      else
        {
          string[i] = 'A' + remainder - 10;
        }
    }
```

```
  //
  // Aggiunge la terminazione, tenendo conto che l'indice
  // "i" è già posizionato dopo l'ultima cifra inserita.
  //
  string[i] = 0;
  //
  reverse_string (string);
}
//
//
//
static void
print_num (unsigned long num, char base)
{
  char string[100];
  //
  if (base == 'x')
    {
      num_to_string (num, 16, string);
      print_string ("0x\0");
      print_string (string);
    }
  else if (base == 'o')
    {
      num_to_string (num, 8, string);
      print_string ("0o\0");
      print_string (string);
    }
  else if (base == 'b')
    {
      num_to_string (num, 2, string);
      print_string ("0b\0");
      print_string (string);
    }
```

```
  else
    {
      num_to_string (num, 10, string);
      print_string (string);
    }
}
```

Listato 65.74. File ‘multiboot.c’, contenente la definizione parziale della struttura delle informazioni *multiboot* e la funzione necessaria a visualizzarne il contenuto.

```
//
// The multiboot information.
//
typedef struct multiboot_info
{
  unsigned long flags;
  unsigned long mem_lower;
  unsigned long mem_upper;
  unsigned long boot_device;
            char *cmdline;
} type_multiboot_info;
//
//
//
static void
multiboot_information (type_multiboot_info *info)
{
  print_string ("flags:       \0");
  print_num (info->flags, 'b');
  print_string ("\n\0");
  //
  if ((info->flags & 1) > 0)
    {
      print_string ("mem_lower:   \0");
```

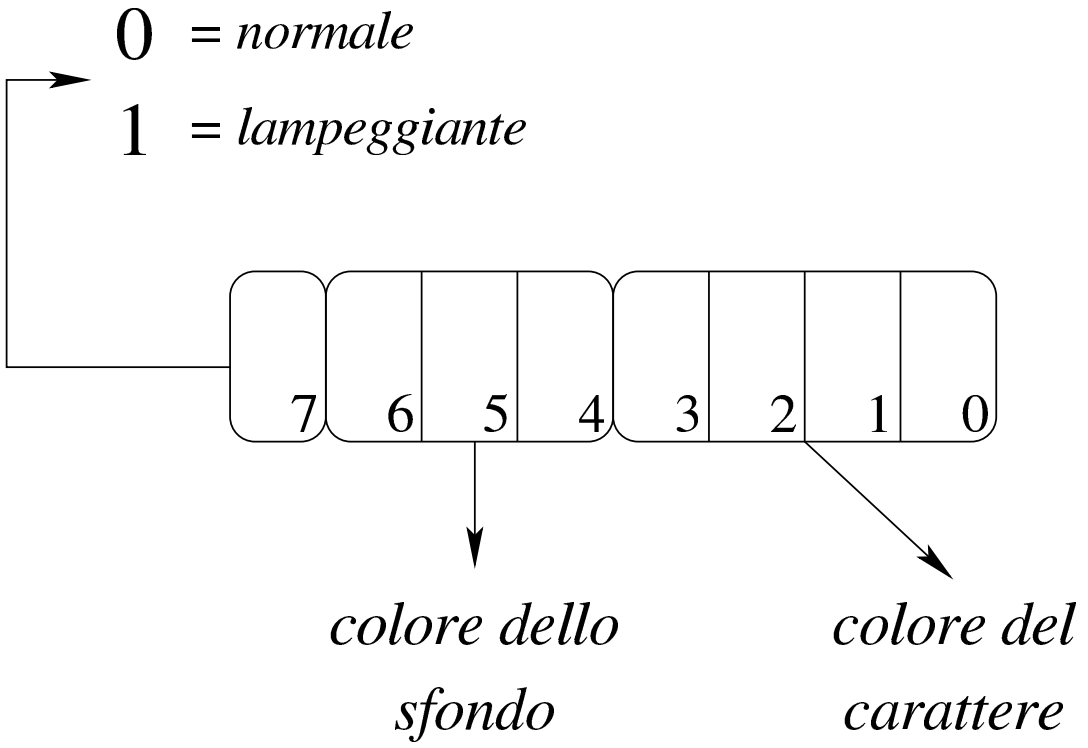
```
      print_num (info->mem_lower, 'x');
      print_string (" \0");
      print_num (info->mem_lower, 'd');
      print_string (" Kibyte\0");
      print_string ("\n\0");
      //
      print_string ("mem_upper:   \0");
      print_num (info->mem_upper, 'x');
      print_string (" \0");
      print_num (info->mem_upper, 'd');
      print_string (" Kibyte\0");
      print_string ("\n\0");
    }
  if ((info->flags & 2) > 0)
    {
      print_string ("boot_device: \0");
      print_num (info->boot_device, 'x');
      print_string ("\n\0");
    }
  if ((info->flags & 4) > 0)
    {
      print_string ("cmdline:     \0");
      print_string (info->cmdline);
      print_string ("\n\0");
    }
}
```

Utilizzando questo programma si potrebbe visualizzare una schermata simile a quella seguente:

```
Salve!
Sono stato avviato attraverso un sistema di avvio aderente
alle specifiche "multiboot". Ecco solo alcune informazioni:
flags:       0b11111100111
mem_lower:   0x27F 639 Kibyte
mem_upper:   0x7C00 31744 Kibyte
boot_device: 0xFFFFFF
cmdline:     (fd0)/kernel
```

65.5.4 Colori dello schermo

Nella sezione precedente si accenna al fatto che a partire dall'indirizzo di memoria $B8000_{16}$, ciò che si scrive serve a ottenere una rappresentazione sullo schermo. Ogni carattere utilizza due byte, in quanto uno dei due contiene il carattere vero e proprio e l'altro l'attributo che ne definisce il colore. Il byte del colore va usato suddividendo i bit nel modo seguente:



Come si vede, se è attivo il bit più significativo si ottiene un carattere lampeggiante, quindi i tre bit successivi descrivono lo sfondo e i quattro bit meno significativi descrivono invece il colore del carattere (in primo piano). Pertanto, i colori dello sfondo sono in quantità minore rispetto a quelli utilizzabili per il primo piano.

Tabella 65.77. Colore associato al primo piano o allo sfondo.

Codice	Sfondo	Primo piano
0_{16}	nero	nero
1_{16}	blu	blu
2_{16}	verde	verde
3_{16}	ciano (azzurro)	ciano (azzurro)
4_{16}	rosso	rosso
5_{16}	magenta (violetto)	magenta (violetto)
6_{16}	marrone	marrone
7_{16}	bianco	bianco
8_{16}	nero con lampeggio	grigio scuro
9_{16}	blu con lampeggio	blu chiaro
A_{16}	verde con lampeggio	verde chiaro
B_{16}	ciano con lampeggio	ciano chiaro
C_{16}	rosso con lampeggio	rosa
D_{16}	magenta con lampeggio	magenta chiaro
E_{16}	marrone con lampeggio	giallo
F_{16}	bianco con lampeggio	bianco luminoso

Per esempio, un colore indicato come 28_{16} genera un testo di colore grigio scuro su sfondo verde, mentre $A0_{16}$ genera un testo lampeggiante nero su sfondo verde.

65.6 Compilazione C dal basso in alto

Il valore del linguaggio C sta nel consentire una programmazione molto vicina a livello del linguaggio macchina, in modo relativamente indipendente dall'architettura. Ma ciò si può comprendere solo se si conosce il contesto operativo del linguaggio assemblatore, in modo particolare per quanto riguarda la gestione della memoria e tanto più per il modo in cui si utilizza la pila dei dati.

Per la compilazione dei programmi di esempio si fa riferimento a GCC[3] (*GNU compiler collection*) e precisamente al programma frontale '**gcc**'.

65.6.1 Compilazione di un programma che non fa uso di librerie

Un programma in linguaggio C che non faccia uso di librerie di alcun tipo, deve seguire alcune regole che riguardano i programmi scritti in linguaggio assemblatore. Il listato seguente contiene la procedura per il calcolo del fattoriale, partendo da un valore già presente in memoria (si calcola precisamente il fattoriale di 5), ma il risultato non viene visualizzato in alcun modo, dal momento che questa sarebbe un'operazione che richiede proprio l'uso di librerie apposite:

```
int x = 5;
int i = 0;
void _start (void)
{
    i = (x - 1);
    while (i > 0)
      {
        x = x * i;
        i--;
```

```
        }
}
```

Se si conoscono i rudimenti del linguaggio C, si può osservare che, al posto della funzione ***main()***, appare invece ***_start()***, come si fa in un programma scritto in linguaggio assemblatore.

Supponendo che il file che contiene quanto mostrato si chiami '`fact.c`', la compilazione potrebbe iniziare dalla trasformazione in linguaggio assemblatore:

```
$ gcc -Wall -Werror -S -o fact.s fact.c ↩
↪     -nostdlib -nostartfiles -nodefaultlibs[Invio]
```

Si otterrebbe il file '`fact.s`', in linguaggio assemblatore, che, a seconda della versione di GCC, potrebbe essere molto simile al listato seguente:

```
        .file   "fact.c"
.globl x
        .data
        .align 4
        .type   x, @object
        .size   x, 4
x:
        .long   5
.globl i
        .bss
        .align 4
        .type   i, @object
        .size   i, 4
i:
        .zero   4
        .text
.globl _start
```

```
        .type   _start, @function
_start:
        pushl   %ebp
        movl    %esp, %ebp
        movl    x, %eax
        decl    %eax
        movl    %eax, i
        jmp     .L2
.L3:
        movl    x, %edx
        movl    i, %eax
        imull   %edx, %eax
        movl    %eax, x
        movl    i, %eax
        decl    %eax
        movl    %eax, i
.L2:
        movl    i, %eax
        testl   %eax, %eax
        jg      .L3
        popl    %ebp
        ret
        .size   _start, .-_start
        .ident  "GCC: (GNU) 4.1.2 20061115 ↩
↪(prerelease) (Debian 4.1.1-21)"
        .section        .note.GNU-stack,"",@progbits
```

Tale file in linguaggio assemblatore può essere compilato con GNU AS e GNU LD nel modo consueto:

```
$ as -o fact.o fact.s[Invio]
```

```
$ ld -o fact fact.o[Invio]
```

Si può ispezionare il programma ottenuto con Objdump:

$ **objdump -x fact** [*Invio*]

```
fact:     file format elf32-i386
fact
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048094

Program Header:
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x000000cd memsz 0x000000cd flags r-x
    LOAD off    0x000000d0 vaddr 0x080490d0 paddr 0x080490d0 align 2**12
         filesz 0x00000004 memsz 0x00000008 flags rw-
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**2
         filesz 0x00000000 memsz 0x00000000 flags rw-

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000039  08048094  08048094  00000094  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000004  080490d0  080490d0  000000d0  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  080490d4  080490d4  000000d4  2**2
                  ALLOC
  3 .comment      0000003a  00000000  00000000  000000d4  2**0
                  CONTENTS, READONLY
SYMBOL TABLE:
08048094 l    d  .text  00000000 .text
080490d0 l    d  .data  00000000 .data
080490d4 l    d  .bss   00000000 .bss
00000000 l    d  .comment       00000000 .comment
00000000 l    d  *ABS*  00000000 .shstrtab
00000000 l    d  *ABS*  00000000 .symtab
00000000 l    d  *ABS*  00000000 .strtab
00000000 l    df *ABS*  00000000 fact.c
080490d0 g     O .data  00000004 x
080490d4 g     O .bss   00000004 i
08048094 g     F .text  00000039 _start
080490d4 g       *ABS*  00000000 __bss_start
080490d4 g       *ABS*  00000000 _edata
080490d8 g       *ABS*  00000000 _end
```

È possibile fare in modo che GCC interpelli automaticamente GNU AS, in modo da generare un file oggetto senza mostrare la creazione del file in linguaggio assemblatore (la trasformazione in linguaggio assemblatore avviene ugualmente, in un file temporaneo che poi viene cancellato in modo automatico). Pertanto, la compilazione si ridurrebbe ai due comandi seguenti:

```
$ gcc -Wall -Werror -c -o fact.o fact.c↩
↪     -nostdlib -nostartfiles -nodefaultlibs[Invio]

$ ld -o fact fact.o[Invio]
```

È il caso di osservare che il programma eseguibile ottenuto dal sorgente mostrato, produce un errore, dal momento che manca la chiamata della funzione del sistema operativo che ne conclude l'attività.

65.6.2 Uso di GDB e di DDD

Per poter sfruttare programmi come GDB, allo scopo di analizzare il funzionamento del programma, è necessario inserire delle informazioni aggiuntive durante la fase di trasformazione nel formato del linguaggio assemblatore. In pratica, si tratta di utilizzare l'opzione '**-gstabs**', o altre simili, nella riga di comando di GCC. Riprendendo l'esempio della sezione precedente, la compilazione verrebbe eseguita con il comando seguente:

```
$ gcc -Wall -Werror -gstabs -S -o fact.s fact.c ↩
↪     -nostdlib -nostartfiles -nodefaultlibs[Invio]
```

In questo caso, nel file in linguaggio assemblatore si troverebbero delle informazioni in più:

```
        .file   "fact.c"
        .stabs  "fact.c",100,0,2,.Ltext0
        .text
.Ltext0:
        .stabs  "gcc2_compiled.",60,0,0,0
        .stabs  "int:t(0,1)=r(0,1);-2147483648;2147483647;",128,0,0,0
        .stabs  "char:t(0,2)=r(0,2);0;127;",128,0,0,0
        .stabs  "long int:t(0,3)=r(0,3);-2147483648;2147483647;",128,0,0,0
        .stabs  "unsigned int:t(0,4)=r(0,4);0;4294967295;",128,0,0,0
        .stabs  "long unsigned int:t(0,5)=r(0,5);0;4294967295;",128,0,0,0
        .stabs  "long long int:t(0,6)=r(0,6);-0;4294967295;",128,0,0,0
        .stabs  "long long unsigned int:t(0,7)=r(0,7);0;-1;",128,0,0,0
        .stabs  "short int:t(0,8)=r(0,8);-32768;32767;",128,0,0,0
        .stabs  "short unsigned int:t(0,9)=r(0,9);0;65535;",128,0,0,0
        .stabs  "signed char:t(0,10)=r(0,10);-128;127;",128,0,0,0
        .stabs  "unsigned char:t(0,11)=r(0,11);0;255;",128,0,0,0
        .stabs  "float:t(0,12)=r(0,1);4;0;",128,0,0,0
        .stabs  "double:t(0,13)=r(0,1);8;0;",128,0,0,0
        .stabs  "long double:t(0,14)=r(0,1);12;0;",128,0,0,0
        .stabs  "void:t(0,15)=(0,15)",128,0,0,0
.globl x
        .data
        .align 4
        .type   x, @object
        .size   x, 4
x:
        .long   5
.globl i
        .bss
        .align 4
        .type   i, @object
        .size   i, 4
i:
        .zero   4
        .text
        .stabs  "_start:F(0,15)",36,0,0,_start
.globl _start
        .type   _start, @function
_start:
```

```
        .stabn  68,0,4,.LM0-_start
.LM0:
        pushl   %ebp
        movl    %esp, %ebp
        .stabn  68,0,5,.LM1-_start
.LM1:
        movl    x, %eax
        decl    %eax
        movl    %eax, i
        .stabn  68,0,6,.LM2-_start
.LM2:
        jmp     .L2
.L3:
        .stabn  68,0,8,.LM3-_start
.LM3:
        movl    x, %edx
        movl    i, %eax
        imull   %edx, %eax
        movl    %eax, x
        .stabn  68,0,9,.LM4-_start
.LM4:
        movl    i, %eax
        decl    %eax
        movl    %eax, i
.L2:
        .stabn  68,0,6,.LM5-_start
.LM5:
        movl    i, %eax
        testl   %eax, %eax
        jg      .L3
        .stabn  68,0,11,.LM6-_start
.LM6:
        popl    %ebp
        ret
        .size   _start, .-_start
.Lscope0:
        .stabs  "x:G(0,1)",32,0,0,0
        .stabs  "i:G(0,1)",32,0,0,0
        .stabs  "",100,0,0,.Letext0
.Letext0:
        .ident  "GCC: (GNU) 4.1.2 20061115 (prerelease) (Debian 4.1.1-21)"
```

```
        .section        .note.GNU-stack,"",@progbits
```

Per la compilazione successiva non ci sono cambiamenti; va quindi osservato che non è più compito di GNU AS l'inserimento di tali informazioni:

```
$ as -o fact.o fact.s[Invio]

$ ld -o fact fact.o[Invio]
```

Per utilizzare GDB o DDD si procede come nel caso di un programma che parte direttamente da un sorgente in linguaggio assemblatore:

```
$ gdb fact[Invio]

(gdb) break _start[Invio]

 Breakpoint 1 at 0x8048097: file fact.c, line 5.


(gdb) run[Invio]

 Breakpoint 1, _start () at fact.c:5
 5           i = (x - 1);


(gdb) stepi[Invio]

 0x0804809c      5           i = (x - 1);


(gdb) stepi[Invio]

 0x0804809d in _start () at fact.c:5
 5           i = (x - 1);


(gdb) stepi[Invio]
```

```
6           while (i > 0)
```

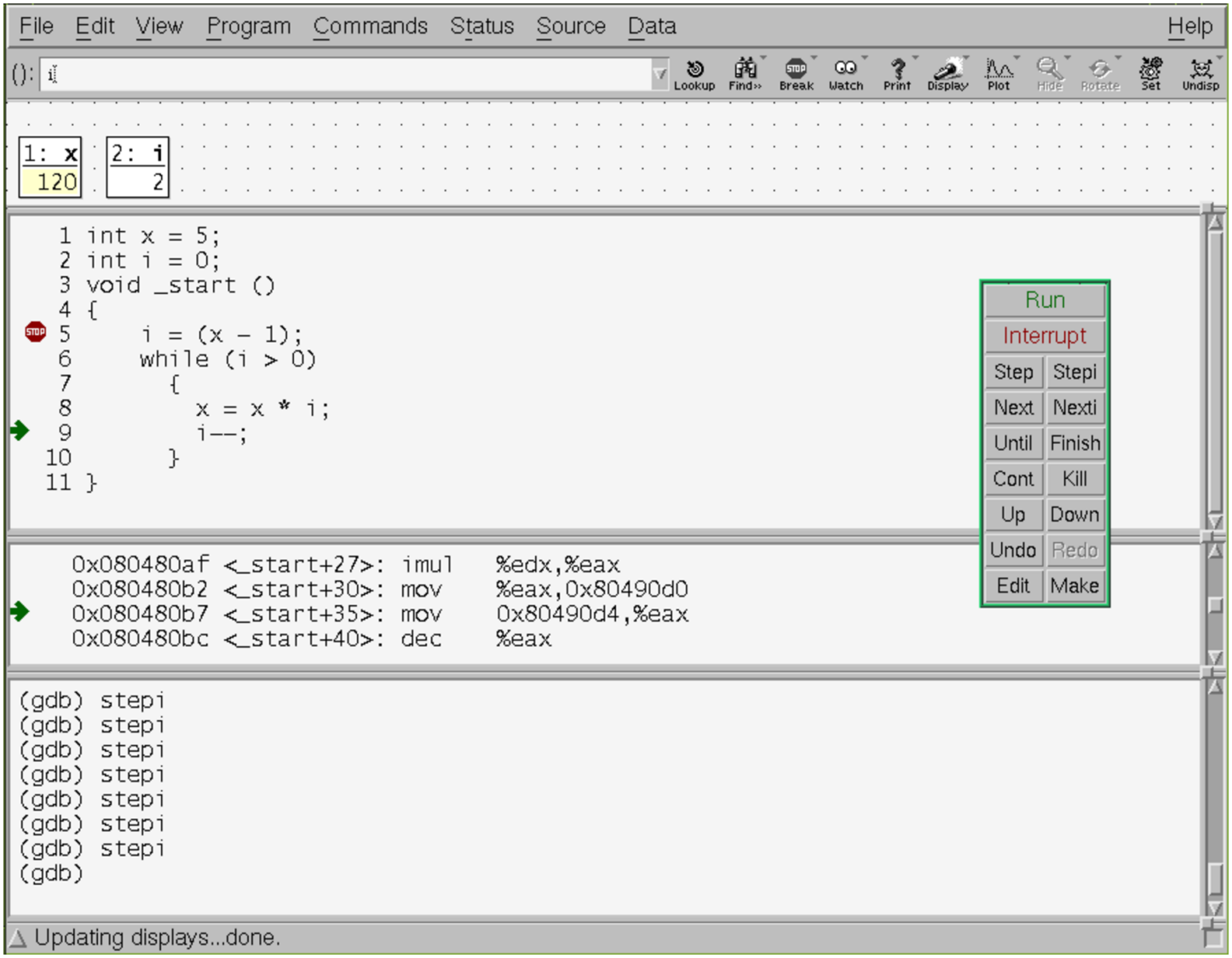
Come si può osservare, occorrono più comandi di avanzamento per passare alla riga successiva del codice originale, perché in realtà si fa riferimento alle istruzioni in linguaggio macchina.

```
(gdb) print i[Invio]

  $1 = 4

(gdb) print x[Invio]

  $2 = 5

(gdb) quit[Invio]
```

Naturalmente, se si può utilizzare DDD, tutto diventa più semplice:

```
$ ddd fact[Invio]
```

Figura 65.89. DDD che mette in evidenza lo stato di due variabili (si attiva la loro visualizzazione facendo un clic sul pulsante a icona denominato DISPLAY) durante il funzionamento, passo passo, del programma.



65.6.3 Da «_start» a «main»

Per fare in modo che un programma in linguaggio C inizi dalla funzione ***main()***, così come si prevede sia, si può istruire il collegatore (*linker*), attraverso uno script apposito che, in un sistema GNU/Linux, potrebbe essere come quello seguente:

```
ENTRY (main)
SECTIONS {
    . = 0x08048000 + SIZEOF_HEADERS;
    .text . : { *(.text) }
    .data ALIGN (0x1000) : { *(.data) }
    .bss . : {
        _sbss = .;
        *(.bss)
        *(COMMON)
        _ebss = .;
    }
}
```

Il nuovo sorgente C:

```
int x = 5;
int i = 0;
int main ()
{
    i = (x - 1);
    while (i > 0)
      {
        x = x * i;
        i--;
      }
    return x;
}
```

Per la compilazione, i passaggi sarebbero quelli seguenti, supponendo che lo script per GNU LD sia contenuto nel file '`config.ld`':

```
$ gcc -Wall -Werror -S -o fact.s fact.c ↩
↪    -nostdlib -nostartfiles -nodefaultlibs[Invio]
```

```
$ as -o fact.o fact.s[Invio]

$ ld -T config.ld -o fact fact.o[Invio]
```

Tuttavia, rimane ancora il problema della conclusione del programma che non avviene in modo grazioso. Se si osserva la nuova versione del programma, la funzione (che ora si chiama ***main()***) restituisce un valore intero, corrispondente al risultato del calcolo eseguito, solo che non è stato chiarito in che modo quel valore debba essere acquisito dal sistema operativo. Si può quindi procedere in un modo diverso, creando un piccolo programma in linguaggio assemblatore, da associare a quello in linguaggio C:

```
.section .text
.globl _start
.extern main
_start:
    call  main
    mov   %eax, %ebx
    mov   $1, %eax
    int   $0x80
```

Supponendo che questo file si chiami '**start.s**', la compilazione complessiva potrebbe essere svolta nel modo seguente:

```
$ gcc -Wall -Werror -gstabs -S -o fact.s fact.c ↩
↪    -nostdlib -nostartfiles -nodefaultlibs[Invio]

$ as -o fact.o fact.s[Invio]

$ as --gstabs -o start.o start.s[Invio]

$ ld -o fact start.o fact.o[Invio]
```

Come si vede sono state aggiunte le opzioni '**-gstabs**' e

‘**--gstabs**’, dove appropriato; inoltre non serve più lo script per GNU LD. Se si avvia il programma, questo si arresta correttamente restituendo il fattoriale di 5:

```
$ ./fact ; echo $?[Invio]

  120
```

65.6.4 Compilazione naturale di un programma in linguaggio C

Quando non si utilizzano le opzioni ‘**-nostdlibs**’, ‘**-nostartfiles**’ e ‘**-nodefaultlibs**’, la compilazione attraverso GCC avviene in modo più intuitivo, con l’inclusione automatica di tutto quello che è necessario per far sì che il programma parta dalla propria funzione ***main()***; inoltre, se non si specifica il nome che si vuole produrre, si ottiene direttamente un file eseguibile con il nome ‘`a.out`’, secondo la tradizione.

In condizioni normali vengono inclusi nella compilazione alcuni file-oggetto che hanno un nome corrispondente al modello ‘`crt*.o`’ e la libreria Libc. All’interno di uno di quei file-oggetto si trova la funzione ***_start()***, dalla quale si arriva poi alla chiamata di ***main()*** in modo analogo a quanto mostrato nella sezione precedente, ma questi file potrebbero coinvolgere anche la libreria Libc.

L’opzione ‘**-nostartfiles**’ serve a impedire che vengano incorporati automaticamente i file che contengono la funzione ***_start()*** e tutto ciò che altrimenti si prevede di far fare al programma prima di entrare nella funzione ***main()***. L’opzione ‘**-nodefaultlibs**’ serve a impedire l’inclusione automatica della libreria Libc. L’op-

zione '**-nostdlibs**' richiede entrambe le cose ed è stata usata negli esempi in modo ridondante.

Ecco la classica compilazione che produce direttamente il file eseguibile con il nome '`a.out`':

```
$ gcc -Wall -Werror -gstabs fact.c[Invio]
```

Per buona abitudine è bene usare sempre l'opzione '**-Wall**' e possibilmente anche '**-Werror**'; inoltre, l'uso di '**-gstabs**' diventa essenziale per potersi avvalere di programmi come '**GDB**'.

Si può verificare che questo basta per arrivare al risultato voluto:

```
$ ./a.out ; echo $?[Invio]

  120
```

Se poi si vogliono usare comandi tradizionali, da '**gcc**' occorre passare a '**cc**', ma in un sistema GNU si tratta normalmente di un collegamento simbolico a '**gcc**' stesso.

Tabella 65.95. Riepilogo delle opzioni utilizzate con '**gcc**' nel corso del capitolo.

Opzione	Descrizione
`-S`	Genera un file in linguaggio assemblatore (prevale sull'opzione '**-c**').
`-o` ***nome_file***	Dichiara il nome del file che si vuole ottenere.

Opzione	Descrizione
`-c`	Fa sì che la compilazione salti la fase di collegamento (*link*). In condizioni normali serve a generare solo i file-oggetto. Se si usa questa opzione, ma non si specifica l'opzione '**`-o`**', il file-oggetto ha un nome con la stessa radice del file sorgente e l'estensione '`.o`'.
`-Wall`	Richiede di mostrare tutti i messaggi che avvertono dell'uso imperfetto del linguaggio (*warning*).
`-Werror`	Fa sì che tutte le segnalazioni di avvertimento siano trattate come errori e portino al fallimento della compilazione.
`-gstabs`	Inserisce delle annotazioni, con le quali i programmi come GDB possono abbinare il sorgente originale all'esecuzione controllata del programma.

65.7 Compilazione C dall'alto in basso

Tradizionalmente, la compilazione di un programma scritto in linguaggio C avviene utilizzando il comando '**`cc`**' come nell'esempio seguente, sapendo che se non si usa l'opzione '**`-o`**' si ottiene il file '`a.out`':

```
$ cc mio.c[Invio]
```

Tuttavia, l'elaborazione del file in linguaggio C richiede diversi passaggi, prima di arrivare al file eseguibile finale; passaggi che è bene tenere in considerazione.

In un sistema GNU il compilatore standard è GCC (*GNU compiler collection*) che si usa sia per il C, sia per altri linguaggi. Nel caso del linguaggio C, il programma frontale è precisamente '**gcc**', al quale corrisponde comunque il collegamento '**cc**'.

Qui non si esaurisce il problema e si accenna soltanto alle situazioni più comuni. Per un approfondimento si vedano i documenti citati nella bibliografia che conclude il capitolo.

65.7.1 Le fasi della compilazione

«

La compilazione di un programma scritto in linguaggio C prevede diverse fasi: precompilazione, trasformazione in linguaggio assemblatore, trasformazione in file-oggetto, collegamento (*link*) di uno o più file-oggetto in un file eseguibile. Per conservare i file intermedi della compilazione si può usare l'opzione '**-save-temps**' di '**gcc**', come nell'esempio seguente:

```
$ gcc -save-temps mio.c[Invio]
```

In questo caso si ottengono i file '`mio.i`', '`mio.s`' e '`mio.o`', contenenti rispettivamente il risultato elaborato dal precompilatore, la trasformazione in linguaggio assemblatore e il file-oggetto finale. Se poi il programma contenuto nel file sorgente è completo, si ottiene anche il file '`a.out`' che costituisce il programma eseguibile.

Eventualmente, alcune opzioni di '**gcc**' consentono di fermare l'elaborazione a uno stadio prestabilito: '**-E**' serve a ottenere solo l'elaborazione da parte del precompilatore; '**-S**' serve a ottenere il sorgente in linguaggio assemblatore; '**-c**' serve a compilare, ma senza eseguire il collegamento finale (pertanto si ottiene il file-oggetto rilocabile).

65.7.2 Precompilatore

Ogni compilatore C «standard» prevede che il file sorgente venga elaborato, prima della compilazione vera e propria, attraverso un precompilatore, il quale elabora il sorgente e genera un altro sorgente ottenuto dall'interpretazione delle istruzioni di «precompilazione». Queste istruzioni di precompilazione costituiscono un linguaggio indipendente dal C vero e proprio. Il precompilatore di GCC è '**cpp**' e di norma viene chiamato automaticamente da '**gcc**' stesso, come già accennato nella sezione precedente.

```
#include <stdio.h>
int main (void)
{
    printf ("Ciao a tutti!\n");
    return 0;
}
```

Nell'esempio mostrato, l'istruzione '**#include <stdio.h>**' riguarda il precompilatore e richiede l'inclusione del file '`stdio.h`' in quella posizione (il file si deve trovare all'interno di una directory prestabilita). Con l'opzione '**-E**' di '**gcc**' (oppure anche con '**-save-temps**') si può vedere il risultato della precompilazione:

```
$ gcc -E -o mio.i mio.c[Invio]
```

Il file '`mio.i`' che si genera dall'elaborazione ha un aspetto simile al pezzo che si vede nel listato successivo:

```
# 1 "mio_file.c"
# 1 "<built-in>"
# 1 "<command line>"
...
...
```

```
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;
…
…
extern void funlockfile (FILE *__stream)
            __attribute__ ((__nothrow__));
# 834 "/usr/include/stdio.h" 3 4

# 2 "mio_file.c" 2
int main (void)
{
    printf ("Ciao a tutti!\n");
    return 0;
}
```

65.7.3 Compilazione dei file intermedi

«

Di norma, ogni compilatore tradizionale del linguaggio C si prende cura di tutte le fasi della compilazione, chiamando a sua volta i programmi necessari. Pertanto, con lo stesso programma frontale è possibile avviare manualmente la compilazione da fasi successive. Per esempio:

1. `$` **`cc mio_file.i`**[*Invio*]
2. `$` **`cc mio_file.s`**[*Invio*]
3. `$` **`cc mio_file.o`**[*Invio*]

Negli esempi si mostra l'uso del comando '**`cc`**', ma '**`gcc`**' è perfettamente conforme a questa convenzione tradizionale. Come si può

intuire, dall'estensione del nome del file il programma frontale determina quali azioni deve intraprendere: nel primo caso avvia la compilazione saltando solo la fase iniziale dell'analisi del precompilatore; nel secondo caso avvia l'assemblatore (e quindi continua con il collegatore); nell'ultimo caso avvia soltanto il collegatore (*linker*).

Naturalmente è possibile mescolare file differenti assieme, se la somma di questi deve portare a un solo file-eseguibile finale. Per esempio, si può compilare un programma composto dai file '`uno.c`', '`due.i`', '`tre.s`' e '`quattro.o`', dove ognuno viene elaborato in base alle proprie esigenze e alla fine il tutto viene collegato assieme:

```
$ cc uno.c due.i tre.s quattro.o [Invio]
```

65.7.4 L'uso di librerie

In generale, la compilazione di un programma scritto secondo il linguaggio C implica automaticamente l'utilizzo della libreria Libc e il collegamento (*link*) con dei file-oggetto predefiniti, che contengono il codice necessario a preparare il programma prima di passare all'esecuzione della funzione ***main()***.

Con '**gcc**', per escludere l'utilizzo di qualunque libreria predefinita vanno usate le opzioni '**-nostartfiles**' e '**-nodefaultlibs**'; eventualmente l'opzione '**-nostdlibs**' dovrebbe valere per entrambe queste opzioni e può essere usata assieme a loro, benché sia ridondante.

Quando si fa uso di funzioni che non sono state dichiarate nel proprio programma, si tratta sempre di qualcosa che è contenuto in una libre-

ria, di solito quella predefinita (Libc), ma per usarle correttamente è indispensabile che sia inserita all'inizio del file la dichiarazione del loro prototipo. Per questo, a seconda delle funzioni che si utilizzano, si includono i file che contengono i prototipi necessari; nel caso della funzione ***printf()*** si include comunemente il file '`stdio.h`'.

Se si utilizza una funzione che appartiene a una libreria prevista nella compilazione, della quale però non si dichiara il prototipo, si può anche ottenere una compilazione «corretta», ma non è detto che, durante il funzionamento del programma, il passaggio degli argomenti attraverso i parametri della funzione avvenga in modo altrettanto corretto. In pratica, è molto probabile che la chiamata di tali funzioni produca risultati errati.

65.7.5 Librerie statiche e librerie dinamiche

«

Le librerie statiche sono file-oggetto raccolti in archivi generati con il programma '**`ar`**', dove i nomi dei file di tali archivi hanno estensione '`.a`'. L'uso di queste librerie implica l'incorporazione del codice utilizzato nel programma finale.

Per compilare un programma che utilizza delle librerie statiche è sufficiente indicare i nomi dei file che le contengono, assieme agli altri file del programma:

```
$ gcc mio.c /usr/lib/libncurses.a [Invio]
```

In alternativa, secondo la modalità normale, quando i file di tali librerie si trovano nelle directory previste, si può usare l'opzione '**`-l`**', a cui si attacca il nome della libreria, ottenuto dal nome del file to-

gliendo l'estensione e il prefisso '`lib`'. Pertanto, l'esempio appena mostrato andrebbe trasformato così:

```
$ gcc -static mio.c -lncurses[Invio]
```

Le librerie dinamiche sono realizzate in modo differente rispetto a quelle statiche e sono contenute normalmente in file con estensione '`.so`'. La compilazione con l'uso di librerie dinamiche avviene in modo analogo a quanto visto per quelle statiche:

```
$ gcc mio.c /usr/lib/libncurses.so[Invio]
```

Oppure:

```
$ gcc -dynamic mio.c -lncurses[Invio]
```

Come si può intuire dagli esempi mostrati, se una stessa libreria è fornita sia in versione statica, sia in versione dinamica, le opzioni '**`-static`**' e '**`-dynamic`**' servono a precisare che tipo di compilazione si vuole. Se però si omette di specificarlo, in generale vengono utilizzate le librerie dinamiche.

L'opzione '**`-l`**' implica una ricerca dei file delle librerie all'interno di directory prestabilite, ma può succedere che sia necessario esplicitarlo nella riga di comando. In tal caso si può usare l'opzione '**`-L`**':

```
$ gcc mio.c -L/opt/mia/lib -lmia[Invio]
```

Nell'esempio appena mostrato, la compilazione richiede l'uso della libreria '**`mia`**' ('`libmia.so`' o '`libmia.a`') che va cercata prima nella directory '`/opt/mia/lib/`'.

Dal momento che l'uso delle librerie si affianca all'inclusione dei file che ne contengono il prototipo, conviene ricordare anche l'op-

zione '**-I**', con la quale si richiede di cercare i file da includere a cominciare dalla directory specificata:

```
$ gcc mio.c -I/opt/mia/include -L/opt/mia/lib -lmia[Invio]
```

In questo nuovo esempio, si specifica anche che i file da includere vanno cercati a cominciare dalla directory '`/opt/mia/include/`'.

Naturalmente, il problema dei percorsi di ricerca per i file da includere riguarda solo quelli che nel sorgente si indicano tra parentesi angolari, come in questo esempio:

```
#include <stdio.h>
```

Diversamente, se il nome fosse messo tra apici doppi, il file verrebbe cercato nel percorso indicato esplicitamente nel sorgente stesso.

A ogni modo, quando la compilazione manifesta dei problemi che non sembrano dovuti a errori sintattici, conviene usare l'opzione '**-v**', con la quale si vede esattamente cosa tenta di fare il programma frontale e dove si interrompe la compilazione. Ciò può essere molto utile per capire, per esempio, quando il problema deriva da file mancanti (librerie o altro).

Per il procedimento necessario alla produzione di una libreria, statica o dinamica, si veda la sezione 65.2.

65.7.6 L'ordine dei file e delle librerie nella compilazione

La compilazione corretta richiede che i file e le librerie siano indicati nella riga di comando secondo un ordine logico: prima il file che contiene la funzione ***main()***, poi i file o le librerie contenenti le funzioni chiamate dal primo file, poi i file o le librerie contenenti le funzioni chiamate dai predecessori e così di seguito. Per esempio, se il file '`uno.c`' contiene la funzione ***main()*** e a sua volta chiama la funzione ***due()*** contenuta nel file '`due.s`', la riga di comando per la compilazione deve avere l'aspetto seguente:

```
$ gcc uno.c due.s …
```

Se poi la funzione ***due()*** si avvale della funzione ***tre()***, contenuta nella libreria '`libtre.a`', la riga di comando si sviluppa così:

```
$ gcc uno.c due.s -ltre …
```

Naturalmente, anche la funzione ***tre()*** potrebbe avvalersi di una funzione contenuta in una seconda libreria. Per esempio potrebbe usare la funzione ***quattro()*** della libreria '`libquattro.so`':

```
$ gcc uno.c due.s -ltre -lquattro …
```

Questa è una regola generale da considerare in fase di collegamento (*link*). Si osservi che GNU LD (ovvero il programma usato automaticamente da '**`gcc`**' per questo scopo) non richiede necessariamente tale accorgimento, ma ugualmente è meglio curarsi di rispettare il principio.

65.7.7 Prevenzione e ricerca degli errori

«

Il linguaggio C può essere usato «bene» o «male», così come ogni altro linguaggio. Nel caso particolare del C, certi modi leciti di scrivere un programma possono essere facilmente motivo di errori banali, evitabili se si chiede al compilatore di segnalare anche le piccole mancanze. In pratica, con '**`gcc`**' è bene usare sempre l'opzione '**`-Wall`**' per ottenere la segnalazione di una serie numerosa di avvertimenti; eventualmente a questa opzione si può aggiungere '**`-Werror`**', con la quale si trasformano gli avvertimenti in errori, così da evitare che in loro presenza la compilazione vada a buon fine.

Per analizzare il funzionamento del programma con GDB o altri analizzatori simili, conviene aggiungere l'opzione '**`-gstabs`**', oppure un'altra opzione che inizi per '**`-g`**...', in base alle caratteristiche del programma usato per l'analisi.

Infine, disponendo di un sistema GNU, o di un altro sistema compatibile con il modello di Unix, è bene abilitare lo scarico dell'immagine dei processi elaborativi in un file (*core dump*). Così facendo, quando durante il funzionamento un programma tenta di eseguire un'azione che il sistema impedisce, questo programma viene fermato e scaricato in un file '`core`' che può essere analizzato successivamente con GDB. A titolo di esempio viene mostrato un sorgente che produce un errore del genere:

```
int main (void)
{
    int a;
    a = 1 / 0;
    return a;
}
```

Se si compila il programma con l'accortezza di aggiungere l'opzione '**-Wall**' si viene avvisati del problema, ma in questo caso si preferisce ignorarlo:

```
$ gcc -Wall -gstabs errore.c[Invio]

  errore.c: In function ‘main’:
  errore.c:4: warning: division by zero
```

Prima di proseguire, ci si assicura che lo scarico dell'immagine del processo elaborativo sia abilitata:[4]

```
$ ulimit -c unlimited[Invio]
```

Si avvia il programma difettoso:

```
$ ./a.out[Invio]

  /bin/sh: line 1: 12134 Floating point exception↩
 ↪(core dumped) ./a.out
```

Il messaggio della shell avvisa di avere «scaricato la memoria», ovvero di avere creato il file '`core`'. Con GDB si può procedere alla ricerca di cosa è stato a causare l'errore:

```
$ gdb a.out core[Invio]

  …
  Core was generated by `./a.out’.
  Program terminated with signal 8, Arithmetic exception.
  #0  0x08048344 in main () at errore.c:4
  4           a = 1 / 0;
```

65.7.8 Problemi con l'ottimizzazione

«

Il compilatore '**gcc**' consente di utilizzare diverse opzioni per ottenere un risultato più o meno ottimizzato. L'ottimizzazione richiede una potenza elaborativa maggiore, al crescere del livello di ottimizzazione richiesto. In situazioni particolari, può succedere che la compilazione non vada a buon fine a causa di questo problema, interrompendosi con segnalazioni più o meno oscure, riferite alla scarsità di risorse. In particolare potrebbe essere rilevato un uso eccessivo della memoria virtuale, per arrivare fino allo scarico della memoria (*core dump*).

È evidente che in queste situazioni diventa necessario diminuire il livello di ottimizzazione richiesto, modificando opportunamente le opzioni relative. L'opzione in questione è '**-O*n***', come descritto nella tabella 65.102. In generale, l'assenza di tale opzione implica la compilazione normale senza ottimizzazione, mentre l'uso dell'opzione '**-O0**' può essere utile alla fine della serie di opzioni, per garantire l'azzeramento delle richieste di ottimizzazione precedenti.

Tabella 65.102. Opzioni di ottimizzazione per '**gcc**'.

Opzione	Descrizione
`-O` `-O1`	Ottimizzazione minima.
`-O2`	Ottimizzazione media.
`-O3`	Ottimizzazione massima.
`-O0`	Annullamento delle richieste precedenti di ottimizzazione.

Alle volte, compilando un programma, può succedere che a causa del livello eccessivo di ottimizzazione prestabilito, non si riesca a produrre alcun risultato. In questi casi, può essere utile ritoccare lo script di Make, dopo l'uso del comando '**`configure`**'; per la precisione si deve ricercare un'opzione che inizia per '**`-O`**'. Purtroppo, il problema sta nel fatto che spesso si tratta di più di uno script, in base all'articolazione dei file che compongono il sorgente.

Ammesso che si tratti dei file '`Makefile`', si potrebbe usare il comando seguente per attuare la ricerca:

```
$ find . -name Makefile ↩
↪        -exec echo \{\} \; ↩
↪        -exec grep \\-O \{\} \; [Invio]
```

Il risultato potrebbe essere simile a quello che si vede qui di seguito:

```
./doc/Makefile
./backend/Makefile
CFLAGS = -g -O2 -W -Wall -DSCSIBUFFERSIZE=131072
./frontend/Makefile
CFLAGS = -g -O2 -W -Wall -DSCSIBUFFERSIZE=131072
./include/Makefile
./japi/Makefile
CFLAGS = -g -O2 -W -Wall -DSCSIBUFFERSIZE=131072
./lib/Makefile
CFLAGS = -g -O2 -W -Wall -DSCSIBUFFERSIZE=131072
./sanei/Makefile
CFLAGS = -g -O2 -W -Wall -DSCSIBUFFERSIZE=131072
./tools/Makefile
CFLAGS = -g -O2 -W -Wall -DSCSIBUFFERSIZE=131072
./Makefile
```

In questo caso, si può osservare che i file '`./doc/Makefile`', '`./include/Makefile`' e '`Makefile`', non contengono tale stringa.

Tabella 65.104. Riepilogo delle altre opzioni utilizzate con '**gcc**' nel corso del capitolo.

Opzione	Descrizione
`-E`	Elabora il file solo con il precompilatore.
`-S`	Genera un file in linguaggio assemblatore (prevale sull'opzione '**-c**').
`-c`	Fa sì che la compilazione salti la fase di collegamento (*link*). In condizioni normali serve a generare solo i file-oggetto. Se si usa questa opzione, ma non si specifica l'opzione '**-o**', il file-oggetto ha un nome con la stessa radice del file sorgente e l'estensione '`.o`'.
`-o` ***nome_file***	Dichiara il nome del file che si vuole ottenere.
`-static` `-dynamic`	Richiede espressamente di compilare utilizzando le librerie statiche o dinamiche.
`-l`***libreria***	Indica il nome di una libreria da utilizzare. Il nome del file che la contiene può essere '`lib`***libreria*****`.a`**' o '`lib`***libreria*****`.so`**', a seconda che si tratti di una libreria statica o dinamica.
`-L`***percorso***	Indica un percorso in cui ricercare i file delle librerie, che prende la precedenza sugli altri già considerati.
`-I`***percorso***	Indica un percorso in cui ricercare i file da includere, che prende la precedenza sugli altri già considerati.

Opzione	Descrizione
`-Wall`	Richiede di mostrare tutti i messaggi che avvertono dell'uso imperfetto del linguaggio (*warning*).
`-Werror`	Fa sì che tutte le segnalazioni di avvertimento siano trattate come errori e portino al fallimento della compilazione.
`-gstabs`	Inserisce delle annotazioni, con le quali i programmi come GDB possono abbinare il sorgente originale all'esecuzione controllata del programma.

65.8 Compilazione guidata con Make

La compilazione di un programma, in qualunque linguaggio sia scritto, può essere un'operazione molto laboriosa, soprattutto se si tratta di aggregare un sorgente suddiviso in più parti, o peggio, se si tratta di un progetto costituito da più programmi. Per semplificare la procedura si potrebbe predisporre uno script che esegue sequenzialmente tutte le operazioni necessarie, ma la tradizione richiede di utilizzare il programma Make.

Uno dei vantaggi più appariscenti nell'uso di Make sta nella possibilità di evitare che vengano rielaborati i file che non sono stati modificati, abbreviando quindi il tempo di compilazione necessario quando si procede a una serie di modifiche limitate.

Make viene usato normalmente assieme a uno script, denominato comunemente '`Makefile`',[5] scritto in un modo che dovrebbe risultare molto semplice da interpretare; tuttavia, è comunque possibile fare il contrario, specialmente con le versioni più evolute di tale pro-

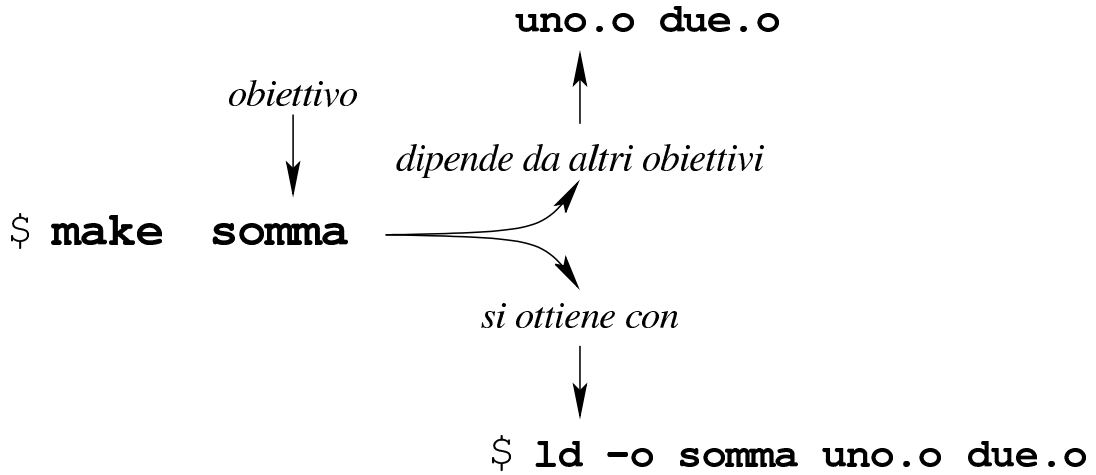
gramma. Evidentemente, Make è utile quando lo si utilizza con moderazione, ovvero con uno script semplice e lineare, altrimenti uno script di shell è sicuramente più appropriato al caso.

Gli esempi che qui mostrano script di Make non contengono commenti, pertanto è bene chiarire subito che le righe bianche o vuote vengono ignorate, così come si ignora il testo che appare alla destra del simbolo '**#**'.

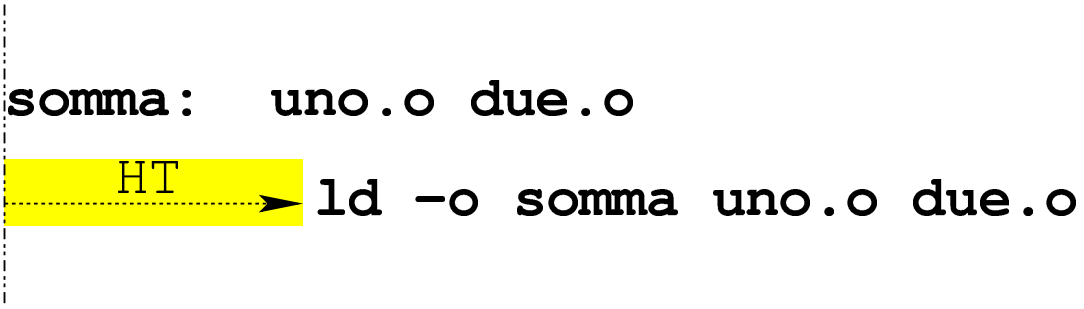
65.8.1 Obiettivo, dipendenze e comandi

Make viene usato per realizzare un ***obiettivo*** attraverso uno o più comandi da impartire alla shell (precisamente '`/bin/sh`'), dopo che sono stati soddisfatti altri eventuali obiettivi da cui quello finale dipende. In linea di principio, **l'obiettivo è rappresentato dal nome di un file che deve essere generato**.

Per esempio, volendo produrre il programma '**somma**' che si ottiene dalla compilazione dei file '`uno.c`' e '`due.c`', l'obiettivo «somma» che si ottiene con il comando '**ld -o somma uno.o due.o**', dipende dagli obiettivi «uno.o» e «due.o», in quanto i file '`uno.o`' e '`due.o`' devono essere presenti per poter eseguire il collegamento con il programma '**ld**'.



Nello script di Make, l'obiettivo di esempio mostrato si descrive come si vede nella figura successiva, dove il tratteggio verticale a sinistra rappresenta l'inizio della prima colonna. Ciò che descrive un obiettivo è, nel suo complesso, una ***regola***:



Si deve tenere a mente che la riga che definisce l'obiettivo e le dipendenze deve iniziare dalla prima colonna, mentre le righe contenenti dei comandi devono trovarsi rientrate con un carattere di tabulazione orizzontale (<*HT*>); al contrario, degli spazi veri e propri come rientro non sono ammissibili.

L'esempio introdotto è incompleto, perché non esplicita in che modo ottenere gli obiettivi '**`uno.o`**' e '**`due.o`**'. Ecco come potrebbe essere composto lo script completo delle regole che descrivono tutte le dipendenze:

```
somma: uno.o due.o
        ld -o somma uno.o due.o

uno.o: uno.c mate.h
        cc -c -o uno.o uno.c

due.o: due.c
        cc -c -o due.o due.c
```

Per comprendere l'esempio va chiarito che per ottenere il file '`uno.o`' è necessario il file '`uno.c`' che a sua volta include il file '`mate.`

h’.

Il vantaggio di usare Make sta nel fatto che questo tiene conto della data di modifica dei file, nel momento in cui valuta le dipendenze. Nel caso dell’esempio, per eseguire il collegamento (*link*) dei file oggetto nel file eseguibile ‘**somma**’, è necessario disporre di tali file oggetto, ma se il file eseguibile esiste e se questi file oggetto esistono e hanno una data di modifica antecedente a quella del file eseguibile, allora sarebbe da intendere che tale operazione non sia necessaria. Tuttavia, i file ‘uno.o’ e ‘due.o’ sono indicati come obiettivi da ottenere attraverso altri file: nel caso di ‘uno.o’ è stabilito che dipende dai file ‘uno.c’ e ‘mate.h’; nel caso di ‘due.o’ è stabilito che dipende solo dal file ‘due.c’ (si osservi che per i file ‘uno.c’, ‘mate.h’ e ‘due.c’ non sono state dichiarate altre dipendenze). A questo punto è logico attendersi che anche la data dei file di partenza conti. In pratica, le date di modifica di ‘uno.c’ e ‘mate.h’ devono essere antecedenti a qualla di ‘uno.o’ e così deve essere antecedente anche quella di ‘due.c’ rispetto a quella di ‘due.o’. Se a un certo punto si modifica il file ‘mate.h’ (e quindi la data di modifica viene aggiornata dal sistema operativo), la dipendenza che riguarda il file ‘uno.o’ richiede la ripetizione dei comandi relativi; quindi viene ricompilato il file ‘uno.o’ e viene eseguito nuovamente il collegamento che genera il file eseguibile ‘**somma**’.

Figura 65.108. Modello sintattico di una regola, per la definizione di un obiettivo in uno script di Make.

obiettivo … : [***dipendenza*** …]
<HT>***comando***
…

Per ottenere lo stesso risultato pratico dell'esempio mostrato, si può modificare il modo in cui si indica la dipendenza dovuta al file 'mate.h':

```
…
uno.o: uno.c
        cc -c -o uno.o uno.c

uno.c: mate.h
        touch uno.c mate.h
…
```

Ciò che appare nel pezzo mostrato indica che il file 'uno.o' dipende da 'uno.c' soltanto, ma il file 'uno.c' dipende dal file 'mate.h'. Se il file 'mate.h' si trova ad avere una data più recente di 'uno.c', le date vengono rese uguali e viene rifatta la compilazione.

65.8.2 Obiettivi fittizi

In generale, un obiettivo di Make viene raggiunto con la creazione o l'aggiornamento di un file che ha lo stesso nome dell'obiettivo, attraverso dei comandi stabiliti. In pratica, l'obiettivo è quel file da generare o aggiornare. Tuttavia, spesso si definiscono obiettivi che non implicano la creazione di un file con tale nome; pertanto servono per essere eseguiti sempre, assicurando che le dipendenze eventuali siano rispettate.

```
all: somma moltiplicazione

somma: …
        …
moltiplicazione: …
        …
…
```

L'esempio mostra una situazione tipica in cui si utilizza un obiettivo fittizio, in questo caso denominato '**all**'. Questo obiettivo ha il solo scopo di richiamare automaticamente gli obiettivi '**somma**' e '**moltiplicazione**' (ma nell'esempio, questi ulteriori obiettivi non vengono descritti). C'è da osservare però una cosa importante: **se per qualunque ragione dovesse esistere un file con lo stesso nome dell'obiettivo, avente una data di modifica successiva a quella dei file degli obiettivi da cui dipende, l'operazione non verrebbe eseguita**, salve naturalmente altre ipotesi riferite alle dipendenze degli obiettivi precedenti.[6]

Per ovviare all'inconveniente dovuto alla possibilità che esista un file con lo stesso nome di un obiettivo fittizio, non correlato a tale file, si può usare uno strattagemma consolidato:

```
clean: FORCE
        rm *.o core

FORCE:
```

In questo caso, l'obiettivo '**FORCE**' (usato comunemente per questo scopo), non ha dipendenze, non ha comandi, inoltre si dà per certo che non possa esistere un file con lo stesso nome; pertanto l'obiettivo risulta sempre da raggiungere. L'obiettivo '**clean**' che ha evidentemente lo scopo di eliminare alcuni file non più necessari, dipendendo dall'obiettivo '**FORCE**', viene eseguito in ogni caso, anche se esistesse un file '`clean`', perché la dipendenza non è mai soddisfatta.[7]

65.8.3 Scelta dell'obiettivo

Make è costituito generalmente dal programma eseguibile '**make**' e si usa solitamente secondo la sintassi seguente:

```
make [opzioni] [obiettivi]
```

Per esempio, il comando seguente richiede a Make di «raggiungere» l'obiettivo '**somma**':

```
$ make somma [Invio]
```

Se però non si specifica l'obiettivo, questo viene determinato in modo predefinito:

```
$ make [Invio]
```

Ammesso che nella directory corrente sia presente lo script di Make (per convenzione deve trattarsi del file '`Makefile`'), l'obiettivo viene cercato al suo interno e se non è stato definito si intende il primo che appare nel file.[8]

È comunque possibile utilizzare Make anche senza script, ma in tal caso l'indicazione dell'obiettivo nella riga di comando è obbligatoria. L'utilizzo di Make senza uno script dipende da quelle che sono definite ***regole implicite***. In pratica, quando si richiede un obiettivo non previsto espressamente, Make cerca di fare la cosa più logica, partendo dal presupposto che il contesto sia relativo alla compilazione di un programma. Si osservi l'esempio seguente:

```
$ make prova [Invio]
```

Se non è stato definito l'obiettivo '**prova**', Make considera il contenuto della directory corrente e cerca qualcosa che sia ragionevol-

mente trasformabile nel file ‘`prova`’. Per esempio, se trova il file ‘`prova.c`’ esegue automaticamente il comando ‘**`cc -o prova prova.c`**’. Questa proprietà di Make consente di omettere la descrizione delle regole degli obiettivi «ovvi». Questo sistema di regole implicite serve anche per semplificare il lavoro di stesura di uno script di Make, quando si descrive un obiettivo finale e non si stabiliscono le regole per ottenere le dipendenze:

```
somma: uno.o due.o
        ld -o somma uno.o due.o

uno.c: mate.h
        touch mate.h uno.c
```

Questo esempio richiama quanto già mostrato in precedenza: dato che la costruzione dei file ‘`uno.o`’ e ‘`due.o`’ richiede dipendenze prevedibili, non è necessario descriverne le regole.

65.8.4 Interpretazione dei comandi che portano a un obiettivo

«

In condizioni normali, i comandi che devono essere eseguiti per il raggiungimento di un certo obiettivo, vengono passati alla shell ‘`/bin/sh`’, indipendentemente dalla shell utilizzata dall’utente che avvia il programma ‘**`make`**’.

I comandi troppo lunghi possono essere spezzati e ripresi nella riga successiva, se alla fine della riga interrotta appare il simbolo ‘`\`’, esattamente come sarebbe in uno script per una shell Bourne. C’è però da osservare che, in questo caso, il comando passato alla shell comprende letteralmente sia ‘`\`’, sia il codice di interruzione di riga successivo, ma questo fatto, di norma, non ha conseguenze nel

risultato.

Sul problema dell'interruzione e proseguimento delle righe dei comandi occorre soffermarsi su un fatto: nella riga che viene ripresa, il carattere di tabulazione iniziale viene omesso automaticamente, nel momento in cui viene chiesto alla shell di eseguire il comando. L'esempio seguente rappresenta il contenuto di uno script che dovrebbe chiarire il meccanismo. Per ora si sorvoli sulla presenza della chiocciola all'inizio dei comandi:

```
esempio:
        @echo "supercalifragilisti\
        chespiralidoso"
        @echo "supercalifragilisti \
        chespiralidoso"
```

Ecco cosa succede:

```
$ make esempio[Invio]

  supercalifragilistichespiralidoso
  supercalifragilisti chespiralidoso
```

Si può osservare che nel primo caso la parola è rimasta unita, mentre nel secondo è separata perché uno spazio è stato inserito prima della segnalazione dell'interruzione.

I comandi di una regola sono eseguiti uno alla volta, ma Make tiene conto del risultato. Se il comando eseguito restituisce zero, ovvero se risulta eseguito correttamente, allora Make avvia il successivo, altrimenti interrompe l'operazione segnalando il fallimento dell'obiettivo e di quelli che da lui dipendono.[9] Pertanto, se i comandi possono restituire un errore anche se ciò non pregiudica il raggiungimento dell'obiettivo previsto, occorre provvedere in qualche modo.

Per esempio così:

```
obiettivo: …
        …
        mkdir ciao ; true
        …
```

In questo caso, la regola che descrive l'obiettivo contiene un comando che serve a garantire la presenza di una certa directory. Il comando in questione potrebbe fallire se la directory esiste già, senza per questo pregiudicare il resto del procedimento, così si unisce al comando '**`true`**' che complessivamente fa sì che l'esito sia sempre «corretto». È comunque possibile usare un prefisso che informa Make di ignorare gli errori; si tratta del segno '**`-`**', pertanto l'esempio appena apparso può essere modificato così:

```
obiettivo: …
        …
        -mkdir ciao
        …
```

In generale, prima di avviare ogni comando, Make lo visualizza, in modo da far capire ciò che accade all'utente. In alcune situazioni, però, ciò può essere spiacevole, pertanto è possibile utilizzare il prefisso '**`@`**' che evita tale comportamento:

```
mio: …
        @echo "sto per eseguire la compilazione, bla bla..."
        cc -o mio mio.c
```

Come si vede nell'esempio, si vuole fare in modo che il comando '**`echo`**' non sia «descritto», dato che già serve a mostrare qualcosa.

Tabella 65.118. Alcuni prefissi da usare nelle righe che contengono comandi.

Prefisso	Significato
`-`	fa in modo che gli errori vengano ignorati;
`+`	fa in modo che il comando venga eseguito sempre;
`@`	fa in modo che il testo del comando non venga mostrato.

65.8.5 Variabili o «macro»

All'interno di uno script di Make è possibile definire delle variabili, altrimenti note come «macro». Le variabili si dichiarano attraverso direttive espresse nella forma seguente:

```
nome = stringa
```

In particolare, la stringa non deve essere delimitata e l'ordine della dichiarazione delle variabili non viene tenuto in considerazione, come dimostrato poco più avanti. L'espansione di una variabile si indica attraverso due modi possibili:

```
$(nome)
```

Oppure:

```
${nome}
```

Si osservi l'esempio seguente, in particolare a proposito del fatto che l'ordine di dichiarazione delle variabili non è significativo:

```
bindir = $(exec_prefix)/bin
prefix = /usr/local
sbindir = $(exec_prefix)/sbin
exec_prefix = $(prefix)

all:
        @echo "prefix = $(prefix)"
        @echo "exec_prefix = $(exec_prefix)"
        @echo "bindir = $(bindir)"
        @echo "sbindir = $(sbindir)"
```

Ammesso che questo sia lo script di Make contenuto nella directory corrente:

```
$ make [Invio]

  prefix = /usr/local
  exec_prefix = /usr/local
  bindir = /usr/local/bin
  sbindir = /usr/local/sbin
```

Il fatto che l'ordine nella dichiarazione nelle variabili non conti, implica che l'assegnamento a una variabile del proprio stesso contenuto produca un circolo vizioso. In pratica, una cosa come la dichiarazione seguente **non è ammissibile**:

```
opzioni = -c
opzioni = -gstabs $(opzioni)
```

Invece di agire così, per aggiungere qualcosa a una variabile occorre una direttiva differente:

nome `+=` ***stringa***

Si osservi l'esempio seguente e ciò che succede provando a usare '**`make`**':

```
opzioni = -c
opzioni += -gstabs

all:
        @echo "opzioni = $(opzioni)"
```

`$` **`make`** [*Invio*]

```
  opzioni = -c -gstabs
```

Come si può intendere, le variabili di Make che appaiono all'interno dei comandi, vengono espanse prima dell'esecuzione dei comandi stessi; di conseguenza, se si vuole usare il dollaro ('**`$`**') in modo che la shell lo recepisca, occorre raddoppiarlo:

```
obiettivo: …
        …
        NUM=3 ; echo $$NUM
        …
```

GNU Make recepisce le variabili di ambiente e le assimila tra le proprie variabili, ma se nel proprio script vengono ridefinite, ciò prevale sul valore ottenuto dall'esterno.

Make prevede delle variabili predefinite, il cui scopo principale è controllare il funzionamento delle regole implicite, ma che spesso

vengono usate per coerenza anche nei comandi di obiettivi dichiarati esplicitamente. La tabella 65.127 ne elenca alcune e l'esempio successivo, riprendendone un altro già apparso, mostra in che modo potrebbero essere usate:

```
somma: uno.o due.o
        $(LD) $(LDFLAGS) -o somma uno.o due.o

uno.o: uno.c mate.h
        $(CC) -c $(CFLAGS) -o uno.o uno.c

due.o: due.c
        $(CC) -c $(CFLAGS) -o due.o due.c
```

Trattandosi di variabili conosciute, se utilizzate correttamente si facilita la lettura dello script, consentendo di precisare, se ce ne fosse bisogno, il nome del compilatore e le opzioni da dare:

```
LD = ld
CC = gcc
CFLAGS = -gstabs

somma: uno.o due.o
        $(LD) $(LDFLAGS) -o somma uno.o due.o

uno.o: uno.c mate.h
        $(CC) -c $(CPPFLAGS) $(CFLAGS) -o uno.o uno.c

due.o: due.c
        $(CC) -c $(CPPFLAGS) $(CFLAGS) -o due.o due.c
```

Tabella 65.127. Elenco di alcune variabili predefinite di Make.

Nome	Contenuto usuale	Annotazioni
`MAKE`	`make`	Il nome del programma stesso. Di solito viene usata questa informazione per l'avvio di altri Make in sottodirectory con un proprio script.
`SHELL`	`/bin/sh`	La shell che deve eseguire i comandi: è bene evitare di cambiare il valore di questa variabile, ovvero, se dichiarata, è bene confermarlo.
`AR` `ARFLAGS`	`ar` `rw`	Il programma di archiviazione usato per creare le librerie statiche e le sue opzioni consuete.
`LD` `LDFLAGS`	`ld`	Il programma usato per collegare i file-oggetto in un file eseguibile e le sue opzioni consuete. Di norma non sono previste opzioni particolari.
`AS` `ASFLAGS`	`as`	Il programma usato per compilare un file in linguaggio assemblatore e le sue opzioni consuete. Di norma non sono previste opzioni particolari.
`CPP` `CPPFLAGS`	`cpp`	Il precompilatore e le sue opzioni consuete. Di norma non sono previste opzioni particolari.

Nome	Contenuto usuale	Annotazioni
`CC` `CFLAGS`	`cc`	Il programma usato per compilare un file in linguaggio C e le sue opzioni consuete. Di norma non sono previste opzioni particolari.
`PC` `PFLAGS`	`pc`	Il programma usato per compilare un file in linguaggio Pascal e le sue opzioni consuete. Di norma non sono previste opzioni particolari.

65.8.6 Utilizzo oculato delle variabili

«

Dal momento che le variabili possono essere espanse in ogni posizione di uno script di Make, le definizioni ripetitive possono essere semplificate. Nell'esempio successivo si dichiara la variabile '**`obj`**', contenente l'elenco dei file-oggetto coinvolti nella produzione di un certo file eseguibile:

```
obj = aaa.o bbb.o ccc.o ddd.o \
      eee.o fff.o ggg.o
…
prog: $(obj)
    ld -o prog $(obj)
…
```

Generalmente, i nomi delle variabili sono scritti utilizzando solo lettere maiuscole, ma non c'è un obbligo in tal senso. Di solito, l'utilizzo di lettere maiuscole per le variabili vuole indicare la possibilità di modificarne il contenuto per qualunque adattamento possa essere necessario; per questo, se invece si utilizzano nomi di variabili con

lettere minuscole (come nell'esempio mostrato), di solito lo si fa per quelle cose che è bene non modificare.

Come già accennato, GNU Make eredita le variabili di ambiente come proprie variabili-macro, anche se poi queste possono essere ridefinite nello script. In ogni caso, il modo «normale» di assegnare un valore a una variabile, nel momento dell'avvio del programma '**make**', è quello di usare la riga di comando, per esempio, così:

```
$ make "CFLAGS = -O" "LDFLAGS = -s" obiettivo [Invio]
```

Si supponga di avere uno script come quello seguente e si osservi cosa succede con il comando appena mostrato:

```
CFLAGS = -gstabs
LDFLAGS = -S

all:
        @echo "CFLAGS:  $(CFLAGS)"
        @echo "LDFLAGS: $(LDFLAGS)"
```

```
$ make "CFLAGS = -O" "LDFLAGS = -s" obiettivo [Invio]

  CFLAGS:  -O
  LDFLAGS: -s
```

Pertanto, questo modo di passare il valore alle variabili di Make prevale sulla dichiarazione interna di uno script.

65.8.7 Espansione e continuazione al di fuori dei comandi

Il testo di uno script di Make, quando non costituisce un comando da passare alla shell e non si tratta nemmeno di un commento, può essere espanso, sia a causa dell'uso di variabili, sia per la presenza di

caratteri che si espandono in nomi di file, come si fa comunemente per le shell POSIX (quindi si possono usare l'asterisco, il punto interrogativo e le parentesi quadre, con lo stesso significato che hanno per una shell POSIX e si possono anche proteggere i simboli, contro l'espansione, facendoli precedere da una barra obliqua inversa: '`\`'). Inoltre, è possibile continuare il testo su più righe, usando il simbolo '`\`' alla fine della riga che deve continuare. L'esempio che appare sotto serve a mostrare l'effetto dell'espansione, ma non è un modello da seguire, perché in pratica si creerebbero delle complicazioni:

```
prog: *.o
        $(LD) $(LDFLAGS) -o prog *.o
```

In questo caso, la realizzazione del file '`prog`' dipende da tutti i file-oggetto presenti nella directory corrente. L'esempio non è utile in generale, perché se tali file sono assenti viene meno la realizzazione dell'obiettivo. È comunque interessante osservare che l'espansione di '`*.o`' nell'elenco delle dipendenze avviene per opera di Make, mentre ciò che appare nel comando viene espanso dalla shell.

65.8.8 Variabili automatiche

«

Alcune variabili non possono essere dichiarate e nemmeno modificate nel loro contenuto. Si tratta delle ***variabili automatiche***, composte da un solo carattere. Per esempio, la variabile '`@`' rappresenta l'obiettivo attuale, ma per espandere il suo contenuto è sufficiente scrivere '`$@`', senza bisogno di parentesi.

Per espandere una variabile si possono sempre evitare le parentesi se il nome di questa è composto da un solo carattere; tuttavia, si preferisce rinunciare alle parentesi solo quando si tratta precisamente di variabili automatiche.

Tabella 65.132. Alcune variabili automatiche.

Variabile automatica	Significato
`*`	Il nome dell'obiettivo attuale, ma senza suffisso; se l'obiettivo non ha suffisso, la variabile risulta vuota.
`@`	L'obiettivo attuale, completo.
`<`	La voce che costituisce la prima dipendenza (quella più a sinistra).
`?`	L'elenco delle dipendenze associate a file che sono più recenti di quello che rappresenta l'obiettivo.
`^`	L'elenco di tutte le dipendenze previste.
`$`	Si espande semplicemente nel simbolo '**`$`**' e, come per tutte le variabili automatiche, va usato aggiungendo un altro dollaro: '**`$$`**'. In pratica, nei comandi, le variabili di ambiente vanno annotate raddoppiando il simbolo dollaro; per esempio: '**`$$HOME`**'.

Per comprendere meglio il significato della descrizione fatta nella tabella precedente, si consideri di disporre dei file seguenti: '`uno.c`', '`due.c`', '`somma.o`'. Inoltre, si suppone che solo '`due.c`' abbia una data di modifica successiva a quella di '`somma.o`'. A tale proposito, si consideri lo script seguente:

```
somma.o: uno.c due.c
        @echo \$$\* = $*
        @echo \$$@ = $@
        @echo \$$\< = $<
        @echo \$$\? = $?
        @echo \$$\^ = $^
```

Se viene avviato, si può leggere lo stato delle variabili automatiche:

```
$ make [Invio]

  $* = somma
  $@ = somma.o
  $< = uno.c
  $? = due.c
  $^ = uno.c due.c
```

Come si può vedere, in questo caso la variabile automatica '**?**' consentirebbe di individuare le dipendenze per le quali si richiede una nuova compilazione.

È importante notare che le variabili automatiche possono essere usate solo all'interno di comandi, perché il loro contenuto si definisce dopo la dichiarazione dell'obiettivo e delle sue dipendenze.

65.8.9 Regole implicite

«

Le regole implicite sono quelle che descrivono degli obiettivi predefiniti, nel modo più logico possibile. Come già accennato altrove, queste regole definiscono i comandi attraverso delle variabili che è possibile controllare.

Tabella 65.135. Comandi comuni di regole implicite.

Comando	Condizione di utilizzo
`$(CC) -c $(CPPFLAGS) $(CFLAGS)` ↩ ↪***nome*`.c`	Se si richiede la realizzazione dell'obiettivo denominato ***nome.o*** ed esiste il file '***nome*`.c`'.
`$(CXX) -c $(CPPFLAGS) $(CXXFLAGS)` ↩ ↪***nome*`.cc` `$(CXX) -c $(CPPFLAGS) $(CXXFLAGS)` ↩ ↪***nome*`.cpp`	Se si richiede la realizzazione dell'obiettivo denominato ***nome.o*** ed esiste il file '***nome*`.cc`' o '***nome*`.cpp`'.
`$(AS) $(ASFLAGS)` ***nome*`.s`**	Se si richiede la realizzazione dell'obiettivo denominato ***nome.o*** ed esiste il file '***nome*`.s`'. Se il file '***nome*`.s`**' è assente ma al suo posto esiste '***nome*`.S`**', allora il primo viene generato dal comando successivo.
`$(CPP) $(CPPFLAGS)` ***nome*`.S`**	Se si richiede la realizzazione dell'obiettivo denominato ***nome.s*** ed esiste il file '***nome*`.S`**'.
`$(CC) $(LDFLAGS)` ***nome*`.o`** ↩ ↪`$(LOADLIBES) $(LDLIBS)`	Se si richiede la realizzazione dell'obiettivo denominato ***nome*** ed esiste il file '***nome*`.o`**'.

Naturalmente, le regole implicite si concatenano tra di loro. Per esempio, si suppone di disporre del file '`prova.c`' e di volerlo compilare utilizzando Make nel modo seguente:

```
$ make prova [Invio]
```

Si sta facendo riferimento all'obiettivo '**prova**' che si intende non sia stato dichiarato nello script di Make. Pertanto, per realizzare questo obiettivo, Make deve cercare una regola implicita appropriata e in questo caso è quella che serve a collegare un file oggetto 'prova.o':

```
prova: prova.o
        $(CC) $(LDFLAGS) prova.o $(LOADLIBES) $(LDLIBS)
```

Questa regola implicita, evidentemente, dipende da un'altra regola che descrive in che modo viene ottenuto il file 'prova.o'. Dal momento che Make trova il file 'prova.c', la regola è questa:

```
prova.o: prova.c
        $(CC) -c $(CPPFLAGS) $(CFLAGS) prova.c
```

Di conseguenza viene eseguita la compilazione.

65.8.10 Uno script per ogni sottodirectory

«

Di solito si predispone uno script di Make per ogni sottodirectory che contenga qualcosa da costruire; poi, in una o in alcune directory si colloca uno script realizzato in modo da avviare lo stesso programma '**make**' nelle sottodirectory inferiori.

A titolo di esempio, si suppone di avere un progetto suddiviso in tre sottodirectory: 'mele/', 'arance/' e 'limoni/'. All'interno di ogni sottodirectory c'è un file 'Makefile'. Nella directory che contiene queste sottodirectory c'è un file 'Makefile' con una regola per avviare sequenzialmente gli altri file equivalenti delle sottodirectory:

```
sub = mele arance limoni

all:
        for d in $(sub) ; do cd $$d ; $(MAKE) ; cd .. ; done
```

Viene usato un ciclo per la scansione delle sottodirectory che la shell interpreta così:

```
for d in mele arance limoni
do
    cd $d
    make
    cd ..
done
```

Ogni volta che si usa Make in questo modo, si dovrebbe vedere un avvertimento come quello seguente:

```
make[1]: Entering directory ‘/home/tizio/mele’
…
make[1]: Leaving directory ‘/home/tizio/mele’
make[1]: Entering directory ‘/home/tizio/arance’
…
make[1]: Leaving directory ‘/home/tizio/arance’
make[1]: Entering directory ‘/home/tizio/limoni’
…
make[1]: Leaving directory ‘/home/tizio/limoni’
```

Per lasciare a Make il controllo del ciclo di avvii nelle sottodirectory, si può usare un meccanismo differente, come quello che si vede nel listato successivo:

```
sub = mele arance limoni

all: $(sub)

$(sub): FORCE
        cd $@ && $(MAKE)

FORCE:
```

Si può vedere che l'obiettivo '**all**' (evidentemente un obiettivo fittizio), dipende dai nomi delle sottodirectory. Successivamente è dichiarata una regola con obiettivo multiplo, ovvero una regola che vale indifferentemente per i tre obiettivi di ogni sottodirectory (la variabile automatica '**$@**' si espande nel nome dell'obiettivo preso in considerazione effettivamente). Tale regola dipende però da un altro obiettivo, senza dipendenze e senza comandi, per il quale si è certi che non possa esistere un file con lo stesso nome.

Come mostrato in questi esempi, invece di scrivere il nome del programma eseguibile '**make**', è stata usata la variabile '**MAKE**', la quale riproduce il nome del comando usato per avviare l'interpretazione dello script. Per esempio, se per qualunque motivo il programma '**make**' fosse nominato in maniera differente o fosse usato al di fuori dei percorsi di ricerca per gli eseguibili, con la variabile '**MAKE**' si garantisce sempre di trovare lo stesso programma che risulta già in funzione.

65.8.11 Una regola per più obiettivi

«

La sezione precedente introduce il concetto di obiettivo multiplo, che però può risultare difficile da intendere con l'esempio mostrato, dal momento che si utilizza una variabile per esprimere l'elenco di

obiettivi. Lo stesso esempio può essere tradotto così, senza l'uso di variabili:

```
all: mele arance limoni

mele arance limoni: FORCE
        cd $@ && $(MAKE)

FORCE:
```

Senza usare una regola del genere, occorrerebbe suddividere la stessa in tre (una per ogni singolo obiettivo):

```
…
mele: FORCE
        cd mele && $(MAKE)

arance: FORCE
        cd arance && $(MAKE)

limoni: FORCE
        cd limoni && $(MAKE)
…
```

Questo dovrebbe chiarire anche l'utilità della variabile automatica '**$@**', per individuare l'obiettivo preso effettivamente in considerazione in un dato momento.

65.8.12 Regole fittizie tipiche

Generalmente si organizza uno script di Make in modo da avere alcuni obiettivi fittizi, con cui eseguire le operazioni più comuni in modo complessivo. Tra questi, l'obiettivo più importante in assoluto è quello che deve essere eseguito in modo predefinito (ovvero il

primo) e generalmente viene chiamato '**all**'. Tale obiettivo serve di norma per indicare soltanto delle dipendenze da soddisfare.

Tabella 65.144. Obiettivi fittizi comuni.

Obiettivo	Significato comune
`all`	Le azioni da compiere quando non si indica alcun obiettivo in modo esplicito.
`clean`	I comandi da eseguire per cancellare i file oggetto, i binari già compilati ed eventualmente altri file temporanei.
`install`	I comandi necessari a installare i programmi dopo la compilazione.

Stando alla tabella appena mostrata, si può ricordare che le fasi tipiche di un'installazione di un programma distribuito in forma sorgente sono appunto quelle seguenti:

1. `#` **`make`**[*Invio*]

   Richiama automaticamente l'obiettivo '**all**', coincidente con i comandi necessari per la compilazione del programma.

2. `#` **`make install`**[*Invio*]

   Provvede a installare gli eseguibili compilati nella loro destinazione prevista.

Supponendo di avere realizzato un programma, denominato '`mio_prog.c`', il cui eseguibile debba essere installato nella directory '`/usr/local/bin/`', si potrebbe utilizzare uno script composto come l'esempio seguente, dove l'obiettivo '**all**' richiama la dipendenza dal programma '**mio_prog**' che viene soddisfatta in modo implicito:

```
all: mio_prog

clean:
        rm -f core *.o mio_prog

install:
        cp mio_prog /usr/local/bin
```

65.8.13 Variabili per l'installazione

In uno script di Make realizzato per la compilazione di un programma (composto eventualmente da uno o più file eseguibili) e per la sua installazione successiva, sono presenti normalmente alcune variabili, più o meno standardizzate, per descrivere la collocazione finale dei file. Alcune di queste variabili sono elencate nella tabella successiva.

Tabella 65.146. Alcune variabili usate comunemente per definire l'installazione.

Variabile	Utilizzo
`prefix`	La variabile '**`prefix`**' viene usata normalmente come punto di partenza da cui traggono origine altre variabili più specifiche. Di solito, il valore dato a questa variabile per la distribuzione di un pacchetto sorgente è '`/usr/local/`', dove poi chi compila e installa deve attribuire un valore che per sé possa essere più appropriato.
`exec_prefix`	La variabile '**`exec_prefix`**' rappresenta il punto di riferimento iniziale per l'installazione dei file eseguibili e di solito corrisponde al contenuto di '**`prefix`**'.

Variabile	Utilizzo
`bindir`	Rappresenta la directory in cui vanno installati i file eseguibili a disposizione di tutti gli utenti e corrisponde normalmente alla directory '`bin/`' successiva al contenuto di '**`exec_prefix`**'. Pertanto, se '**`prefix`**' e '**`exec_prefix`**' indicano '`/usr/local/`', '**`bindir`**' indica normalmente '`/usr/local/bin/`'.
`sbindir`	Rappresenta la directory in cui vanno installati i file eseguibili utili per l'amministrazione e corrisponde normalmente alla directory '`sbin/`' successiva al contenuto di '**`exec_prefix`**'.

Uno script che usa queste variabili potrebbe essere realizzato così:

```
prefix = /usr/local
exec_prefix = $(prefix)
bindir=$(exec_prefix)/bin

all: mio_prog

clean:
        rm -f core *.o mio_prog

install:
        cp mio_prog $(bindir)
```

65.8.14 Definizione della shell

«

In linea di principio, la shell usata per eseguire i comandi contenuti nelle regole di uno script di Make è '`/bin/sh`'. Tuttavia, il nome e il percorso esatto possono essere controllati attraverso la variabile '**`SHELL`**'. In generale può essere conveniente aggiungere nello script la dichiarazione seguente, in modo da non avere sorprese:

```
SHELL = /bin/sh
```

Per esempio, utilizzando GNU Make occorre considerare che le variabili di ambiente sono ereditate e la presenza eventuale di una variabile '**SHELL**' potrebbe creare problemi: è per questo che la dichiarazione suggerita può essere conveniente.

Evidentemente, nello stesso modo descritto è possibile cambiare la shell che interpreta i comandi, ma ciò è sicuramente sconsigliabile, nell'ottica della creazione di script «standard».

65.8.15 Installazione dei programmi

È il caso di osservare che, normalmente, l'installazione dei programmi, ovvero la loro copia nella destinazione finale, dopo la compilazione, si esegue con il programma '**install**' (eventualmente si veda la sezione 20.11.7). Il motivo di questa scelta sta normalmente nella facilità con cui, assieme alla copia, si definiscono i permessi e la proprietà del file nella destinazione. Lo script seguente riprende gli ultimi esempi e concetti già visti:

```
SHELL = /bin/sh
prefix = /usr/local
exec_prefix = $(prefix)
bindir=$(exec_prefix)/bin

all: mio_prog

clean:
        rm -f core *.o mio_prog

install:
```

```
install -o root -g bin -m 755 mio_prog $(bindir)
```

65.9 Riferimenti

«

- *YoLinux Tutorial - Static, Shared Dynamic and Loadable Linux Libraries*, http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html
- Steve Chamberlain, Ian Lance Taylor, *Using LD, the GNU linker*, http://www.zemris.fer.hr/~leonardo/oszur/tehnicki.dokumenti/gnu-linker.pdf
- *a.out (file format)*, http://en.wikipedia.org/wiki/A.out_(file_format)
- *COFF*, http://en.wikipedia.org/wiki/COFF
- *Portable Executable*, http://en.wikipedia.org/wiki/Portable_Executable
- *DWARF*, http://en.wikipedia.org/wiki/DWARF
- *Mach-O*, http://en.wikipedia.org/wiki/Mach-O
- *Executable and linkable format* http://en.wikipedia.org/wiki/Executable_and_Linkable_Format
- *format of executable and linking format (ELF) files*, pagina di manuale *elf(5)*
- Brian Raiter, *A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux*, http://www.muppetlabs.com/~breadbox/software/tiny/teensy.html

- *Multiboot specification*, http://www.gnu.org/software/grub/manual/multiboot/
- osdev.org wiki, *Tutorial: bare bones*, http://wiki.osdev.org/Tutorial:Bare_bones
- Gergor Brunmar: *The booting process*, *The world of Protected mode*, *Mixing Assembly and C-code*, http://www.osdever.net/tutorials/pdf/gb_booting.pdf , http://www.osdever.net/tutorials/pdf/gb_pmode.pdf , http://www.osdever.net/tutorials/pdf/gb_asm_and_c.pdf
- mr. xsism, *Xosdev*, http://www.osdever.net/tutorials/pdf/ch01.pdf , http://www.osdever.net/tutorials/pdf/ch02.pdf
- Tim Robinson, *Writing a Kernel in C*, http://www.osdever.net/tutorials/pdf/ckernel.pdf
- Joachim Nock, K.J., *Making a Simple C kernel with Basic printf and clearscreen Functions*, http://www.osdever.net/tutorials/view/writing-a-simple-c-kernel
- OSDevWiki, *Category: Tutorials*, http://wiki.osdev.org/Category:Tutorials
- OSDevWiki, *Category: Babystep*, http://wiki.osdev.org/Category:Babystep
- Daniel Rowell Faulkner, *Hello World Boot Loader*, http://www.osdever.net/tutorials/view/hello-world-boot-loader
- OSRC, *the Operating System resource center*, http://www.nondot.org/sabre/os/articles
- SigOps, *How to Write an Operating System*, http://www.acm.uiuc.edu/sigops/roll_your_own/ , http://www.acm.uiuc.edu/

sigops/roll_your_own/hardware/kb.html , *http://www.acm.uiuc.edu/sigops/roll_your_own/hardware/text.html*

- *GNU Compiler Collection*, *http://en.wikipedia.org/wiki/GNU_Compiler_Collection*
- *Using the GNU compiler collection (GCC)*, *http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/* , *http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc.pdf*
- Brian Gough, *An Introduction to GCC*, *http://www.network-theory.co.uk/docs/gccintro/*
- Richard M. Stallman, Roland McGrath and Paul D. Smith, *GNU Make: A Program for Directing Recompilation*, ISBN 1-882114-83-3, *http://www.gnu.org/software/make/manual/*

[1] **GNU Binutils** GNU GPL

[2] Sono pochi i sistemi operativi affermati, mentre esistono una miriade di progetti più o meno abbandonati; quindi: prima di pensare di scrivere il proprio sistema converrebbe dare un'occhiata a quanti lavori del genere sono stati catalogati.

[3] **GCC** GNU GPL

[4] Di norma, il comando '**`ulimit`**' è gestito internamente dalla shell; in questo esempio si fa riferimento a una shell POSIX.

[5] Lo script di Make potrebbe essere nominato anche in altri modi, per esempio senza l'iniziale maiuscola (quindi solo '`makefile`') ma in generale conviene attenersi al suggerimento di usare il nome '`Makefile`' che, in un elenco ordinato per nome dei file di una directory, ha il vantaggio di apparire prima di altri a causa dell'iniziale maiuscola.

[6] Nel caso di GNU Make esiste una direttiva apposita per dichiarare quali obiettivi sono fittizi (*phoney*).

[7] GNU Make offre un meccanismo più accurato per impedire che un obiettivo fittizio sia bloccato da un file, ma il metodo mostrato è valido in generale.

[8] Nel caso di GNU Make, nella ricerca del primo obiettivo si escludono i nomi che iniziano con un punto, dal momento che a quelli viene dato un significato particolare.

[9] Di solito il fallimento di un obiettivo può comportare la cancellazione contestuale del file corrispondente all'obiettivo mancato, anche se si tratta di una versione precedente.

Capitolo 66

Introduzione al linguaggio C

66.1 Nozioni minime 563
66.1.1 Struttura fondamentale 564
66.1.2 Ciao mondo! 566
66.1.3 Variabili e tipi 569
66.1.4 Operatori ed espressioni 586
66.1.5 Strutture di controllo di flusso 597
66.1.6 Funzioni 607
66.1.7 Vincoli nei nomi 611
66.1.8 I/O elementare 612
66.1.9 Restituzione di un valore 615
66.1.10 Attributi per GNU C 617
66.2 Istruzioni del precompilatore 619
66.2.1 Linguaggio a sé stante 620
66.2.2 Direttiva «#include» 621
66.2.3 Direttiva «#define» 623
66.2.4 Direttiva «#define» con parametri 627
66.2.5 Direttive «#if», «#else», «#elif» e «#endif» . . 634
66.2.6 Direttive «#if defined», «#if !defined», «#ifdef» e «#ifndef» 636
66.2.7 Direttiva «#undef» 640
66.2.8 Direttiva «#line» 640
66.2.9 Direttiva «#error» 645

66.2.10 Macro predefinite 646
66.2.11 Pragma 649
66.3 Dal campo di azione alla compilazione 650
66.3.1 Il punto di vista del «collegatore» 650
66.3.2 Campo di azione legato al file sorgente 651
66.3.3 Semplificazione dovuta all'uso comune dei file di intestazione 655
66.3.4 Campo di azione interno alle funzioni 657
66.3.5 Campo di azione interno ai raggruppamenti di istruzioni 661
66.3.6 Funzioni annidate 662
66.3.7 Visibilità, accessibilità, staticità 663
66.3.8 Compilazione di un progetto composto da più file . 666
66.3.9 Osservazioni sulla vita delle costanti letterali 668
66.3.10 Libreria standard e file di intestazione 669
66.4 Annotazioni sulla terminologia 670
66.4.1 Parametri e argomenti 670
66.4.2 Byte e caratteri 671
66.4.3 Unità di traduzione 672
66.4.4 «Linkage» 672
66.4.5 Durata di memorizzazione 673
66.4.6 «Lvalue» e «rvalue» 674
66.4.7 «Digraph» e «Trigraph» 676
66.5 Puntatori, array, stringhe e allocazione dinamica della memoria 677

66.5.1 Espressioni a cui si assegnano dei valori 678
66.5.2 Puntatori 679
66.5.3 Array 687
66.5.4 Array multidimensionali 693
66.5.5 Natura dell'array 696
66.5.6 Puntatori costanti 701
66.5.7 Array e funzioni 702
66.5.8 Aritmetica dei puntatori 704
66.5.9 Osservazioni sui puntatori 709
66.5.10 Stringhe 710
66.5.11 Parametri della funzione main() 720
66.5.12 Puntatori a puntatori 723
66.5.13 Puntatori a più dimensioni 725
66.5.14 Puntatori e funzioni 730
66.5.15 Puntatori a variabili distrutte 734
66.5.16 Puntatore nullo 735
66.5.17 Utilizzo della memoria in modo dinamico . 737
66.5.18 Puntatori «ristretti» 740

66.6 Le funzioni 743

66.6.1 Pila dei dati 743
66.6.2 Dichiarazione e chiamata di una funzione 745
66.6.3 Elenco indefinito di parametri 746
66.6.4 Annotazioni su «printf()» e altre funzioni simili . . 753
66.6.5 Costante predefinita «__func__» 754

66.7 Struttura, unione, campo, enumerazione, costante composta 755

66.7.1 Enumerazioni 755
66.7.2 Strutture 759
66.7.3 Assegnamento, inizializzazione, campo di azione e puntatori delle strutture 761
66.7.4 Scostamento all'interno delle strutture 767
66.7.5 Unioni 768
66.7.6 Campi 770
66.7.7 Istruzione «typedef» 772
66.7.8 Costanti letterali composte 773

66.8 Tipi di dati speciali, di uso comune 777

66.8.1 Tipo «_Bool» 778
66.8.2 Tipo «void» 779
66.8.3 Tipo «size_t» 781
66.8.4 Tipo «ptrdiff_t» 783
66.8.5 Tipo «va_list» 784
66.8.6 Tipo «wchar_t» 786
66.8.7 Tipo «wint_t» 787
66.8.8 Tipo «time_t» 787
66.8.9 Tipo «struct tm» 787
66.8.10 Tipo «FILE» 788
66.8.11 Tipo «fpos_t» 788

66.9 Configurazione locale 789

66.9.1 Configurazione locale nei sistemi Unix e simili ... 789
66.9.2 Configurazione locale nel linguaggio C 790
66.9.3 Caratteri multibyte e caratteri estesi 792
66.9.4 Concatenamento eterogeneo 796
66.9.5 Conversione tra caratteri multibyte e caratteri estesi 796
66.10 Organizzazione dei file sorgenti 800
66.10.1 File di intestazione 801
66.10.2 Funzioni pubbliche 801
66.10.3 Funzioni e variabili private 802
66.10.4 Esempio di «stdlib.h» 802
66.10.5 Parametri delle macroistruzioni 806
66.10.6 Compilazione 807
66.11 K&R 808
66.11.1 Prototipi e chiamate delle funzioni 809
66.11.2 Dichiarazione delle funzioni 811
66.11.3 Operatori composti di assegnamento 812
66.11.4 Tipi numerici 813
66.11.5 Tipo «void *» 814
66.11.6 Direttive del preprocessore 814
66.11.7 Altre osservazioni su K&R 814
66.11.8 Unproto 815
66.12 Riferimenti 816

! 586 590 != 586 590 * 586 588 679 ** 723 725 *** 723
*...const 701 *= 586 588 *& 709 + 586 588 ++ 586 588 += 586

588 . 759 761 / 586 588 /*...*/ 564 // 564 /= 586 588 0... 575 0x... 575 ; 564 = 586 588 == 586 590 ? : 586 590 argc 720 argv 720 auto 657 bool 778 break 599 602 604 calloc() 737 case 599 *cast* 594 char 571 const 583 584 const...* 701 const volatile 584 continue 602 604 cpp 619 default 599 do 603 double 571 else 598 enum 755 exit() 615 extern 651 657 extern const volatile 584 F 575 FILE 788 float 571 for 604 fpos_t 788 free() 737 if 598 int 571 L 575 575 LL 575 locale.h 790 long 571 long long 571 L"..." 792 L'...' 792 main() 720 malloc() 737 NULL 735 offsetof 767 printf() 568 753 prototipo di funzione 608 ptrdiff_t 783 realloc() 737 register 657 restrict 740 return 609 short 571 signed 571 sizeof 687 size_t 781 static 651 657 stdarg.h 746 stdlib.h 737 strcat() 714 strchr() 714 strcmp() 714 strcoll() 714 strcpy() 714 strcspn() 714 string.h 714 strlen() 714 strncat() 714 strncmp() 714 strncpy() 714 strpbrk() 714 strrchr() 714 strspn() 714 struct 759 struct tm 787 switch 599 time_t 787 typedef 772 U 575 UL 575 ULL 575 union 768 unsigned 571 va_arg 746 va_end 746 va_list 746 784 va_start 746 void 586 608 779 volatile 584 wchar_t 786 792 while 602 wint_t 787 # 564 #define 623 #define() 627 #define()...# 627 #define()...## 627 #define()...__VA_ARGS__ 627 #define...## 623 #elif 634 #else 634 #endif 634 #error 645 #if 634 #ifdef 636 #ifndef 636 #if !defined 636 #if defined 636 #include 621 #line 640 #pragma 649 #undef 640 & 586 592 679 &* 709 &= 586 592 && 586 590 ^ 586 592 ^= 586 592 ~

586 592 ~= 586 592 \... 575 \0 575 \? 575 \a 575 \b 575 \f 575 \n 575 \r 575 \t 575 \v 575 \x... 575 \" 575 \\ 575 \' 575 | 586 592 |= 586 592 || 586 590 {...} 564 _Bool 778 _Pragma 649 __DATE__ 646 __FILE__ 646 __func__ 754 __LINE__ 646 __STDC_HOSTED__ 646 __STDC_IEC_559__ 646 __STDC_IEC_COMPLEX__ 646 __STDC_ISO_10646__ 646 __STDC_VERSION__ 646 __STDC__ 646 __TIME__ 646 __VA_ARGS__ 627 '...' 575 , 596 - 586 588 -= 586 588 -- 586 588 -> 761 < 586 590 <= 586 590 << 586 592 <<= 586 592 > 586 590 >= 586 590 >> 586 592 >>= 586 592 % 586 588 %= 586 588

66.1 Nozioni minime

Il linguaggio C è il fondamento dei sistemi Unix. Un minimo di conoscenza di questo linguaggio è importante per districarsi tra i programmi distribuiti in forma sorgente, pur senza volerli modificare.

Il linguaggio C richiede la presenza di un compilatore per generare un file eseguibile (o interpretabile) dal kernel. Se si dispone di un sistema GNU con i cosiddetti «strumenti di sviluppo», intendendo con questo ciò che serve a ricompilare il kernel, si dovrebbe disporre di tutto quello che è necessario per provare gli esempi di questi capitoli.

Il linguaggio C standard deve disporre di una libreria applicabile a ogni tipo di architettura e di sistema operativo; pertanto le funzionalità di tale libreria è molto limitata. In questi capitoli dedicati al linguaggio C, quando si vuole fare riferimento a funzioni che sono definite al di fuori dello standard minimo, ciò viene annota-

to espressamente; in particolare, nel caso di estensioni che riguardano lo standard POSIX, può apparire anche una nota a margine simbolica.

66.1.1 Struttura fondamentale

Il contenuto di un sorgente in linguaggio C può essere suddiviso in tre parti: commenti, direttive del precompilatore e istruzioni C. I commenti vanno aperti e chiusi attraverso l'uso dei simboli '`/*`' e '`*/`'; se poi il compilatore è conforme a standard più recenti, è ammissibile anche l'uso di '`//`' per introdurre un commento che termina alla fine della riga.

```
/* Questo è un commento che continua
   su più righe e finisce qui. */

// Qui inizia un altro commento che termina alla fine della
// riga; pertanto, per ogni riga va ripetuta la sequenza
// "//" di apertura.
```

Le direttive del precompilatore rappresentano un linguaggio che guida alla compilazione del codice vero e proprio. L'uso più comune di queste direttive viene fatto per includere porzioni di codice sorgente esterne al file. È importante fare attenzione a non confondersi, dal momento che tali istruzioni iniziano con il simbolo '`#`': non si tratta di commenti.

Il programma C tipico richiede l'inclusione di codice esterno composto da file che terminano con l'estensione '`.h`'. La porzione di libreria che viene utilizzata più frequentemente è quella necessaria alla gestione dei flussi di standard input, standard output e standard error; si dichiara implicitamente il suo utilizzo includendo il file di intestazione '`stdio.h`' nel modo seguente:

```
#include <stdio.h>
```

Le istruzioni C terminano con un punto e virgola (‘`;`’) e i raggruppamenti di queste (noti come «istruzioni composte») si fanno utilizzando le parentesi graffe (‘`{ }`’).[1]

***istruzione* ;**

{ ***istruzione* ;** ***istruzione* ;** ***istruzione* ;** }

Generalmente, un’istruzione può essere interrotta e ripresa nella riga successiva, dal momento che la sua conclusione è dichiarata chiaramente dal punto e virgola finale. L’istruzione nulla viene rappresentata utilizzando un punto e virgola da solo.

I nomi scelti per identificare ciò che si utilizza all’interno del programma devono seguire regole determinate, definite dal compilatore C a disposizione. Ma per cercare di scrivere codice portabile in altre piattaforme, conviene evitare di sfruttare caratteristiche speciali del proprio ambiente. In particolare:

- un nome può iniziare con una lettera alfabetica e continuare con altre lettere, cifre numeriche e il trattino basso;
- i nomi potrebbero iniziare anche con il trattino basso, ma ciò è preferibile evitarlo, se non ci sono motivi validi per questo;[2]
- nei nomi si distinguono le lettere minuscole da quelle maiuscole (pertanto, ‘**`Nome`**’ è diverso da ‘**`nome`**’ e da tante altre combinazioni di minuscole e maiuscole).

La lunghezza dei nomi può essere un elemento critico; generalmente la dimensione massima dovrebbe essere di 32 caratteri, ma ci sono versioni di C che ne possono accettare solo una quantità inferiore. In particolare, il compilatore GNU ne accetta molti di più di 32. In ogni caso, il compilatore non rifiuta i nomi troppo lunghi, semplicemente non ne distingue più la differenza oltre un certo punto.

Il codice di un programma C è scomposto in funzioni, dove normalmente l'esecuzione del programma corrisponde alla chiamata della funzione ***main()***. Questa funzione può essere dichiarata senza parametri, '**`int main (void)`**', oppure con due parametri precisi: '**`int main (int argc, char *argv[])`**'.[3]

66.1.2 Ciao mondo!

«

Come sempre, il modo migliore per introdurre a un linguaggio di programmazione è di proporre un esempio banale, ma funzionante. Al solito si tratta del programma che emette un messaggio e poi termina la sua esecuzione.

```
/*
 *      Ciao mondo!
 */

#include <stdio.h>

/* La funzione main() viene eseguita automaticamente
   all'avvio. */
int main (void)
{
  /* Si limita a emettere un messaggio. */
  printf ("Ciao mondo!\n");
}
```

Nel programma sono state inserite alcune righe di commento. In particolare, all'inizio, l'asterisco che si trova nella seconda riga ha soltanto un significato estetico, per guidare la vista verso la conclusione del commento stesso.

Il programma si limita a emettere la stringa «Ciao Mondo!» seguita da un codice di interruzione di riga, rappresentato dal simbolo '**\n**'.

66.1.2.1 Compilazione

Per compilare un programma scritto in C si utilizza generalmente il comando '**cc**', anche se di solito si tratta di un collegamento simbolico al vero compilatore che si ha a disposizione. Supponendo di avere salvato il file dell'esempio con il nome '`ciao.c`', il comando per la sua compilazione è il seguente:

```
$ cc ciao.c[Invio]
```

Quello che si ottiene è il file '`a.out`' che dovrebbe già avere i permessi di esecuzione.

```
$ ./a.out[Invio]

  Ciao mondo!
```

Se si desidera compilare il programma definendo un nome diverso per il codice eseguibile finale, si può utilizzare l'opzione standard '**-o**'.

```
$ cc -o ciao ciao.c[Invio]
```

Con questo comando, si ottiene l'eseguibile '**ciao**'.

```
$ ./ciao[Invio]

  Ciao mondo!
```

In generale, se ciò è possibile, conviene chiedere al compilatore di mostrare gli avvertimenti (*warning*), senza limitarsi ai soli errori. Pertanto, nel caso il compilatore sia GNU C, è bene usare l'opzione '**-Wall**':

```
$ cc -Wall -o ciao ciao.c[Invio]
```

66.1.2.2 Emissione dati attraverso «printf()»

L'esempio di programma presentato sopra si avvale della funzione ***printf()***[4] per emettere il messaggio attraverso lo standard output. Questa funzione è più sofisticata di quanto possa apparire dall'esempio, in quanto permette di comporre il risultato da emettere. Negli esempi più semplici di codice C appare immancabilmente questa funzione, per cui è necessario descrivere subito, almeno in parte, il suo funzionamento.

```
int printf (stringa_di_formato [, espressione] ...);
```

La funzione ***printf()*** emette attraverso lo standard output la stringa che costituisce il primo parametro, dopo averla rielaborata in base alla presenza di ***specificatori di conversione*** riferiti alle eventuali espressioni che compongono gli argomenti successivi; inoltre restituisce il numero di caratteri emessi.

L'utilizzo più semplice di ***printf()*** è quello che è già stato visto, cioè l'emissione di una stringa senza specificatori di conversione (il codice '**\n**' rappresenta un carattere preciso e non è uno specificatore, piuttosto si tratta di una cosiddetta sequenza di escape).

```
   printf ("Ciao mondo!\n");
```

La stringa può contenere degli specificatori di conversione del tipo '**%i**', '**%c**', '**%f**',... e questi fanno ordinatamente riferimento agli argomenti successivi. L'esempio seguente fa in modo che la stringa incorpori il valore del secondo argomento nella posizione in cui appare '**%i**':

```
printf ("Totale fatturato: %i\n", 12345);
```

Lo specificatore di conversione '**%i**' stabilisce anche che il valore in questione deve essere trasformato secondo una rappresentazione decimale intera. Per cui, il risultato diviene esattamente quello che ci si aspetta.

```
Totale fatturato: 12345
```

66.1.3 Variabili e tipi

I tipi di dati elementari gestiti dal linguaggio C dipendono dall'architettura dell'elaboratore sottostante. In questo senso, volendo fare un discorso generale, è difficile definire la dimensione delle variabili numeriche; si possono dare solo delle definizioni relative. Solitamente, il riferimento è dato dal tipo numerico intero ('**int**') la cui dimensione in bit corrisponde a quella della ***parola***, ovvero dalla capacità dell'unità aritmetico-logica del microprocessore, oppure a qualunque altra entità che il microprocessore sia in grado di gestire con la massima efficienza. In pratica, con l'architettura x86 a 32 bit, la dimensione di un intero normale è di 32 bit, ma rimane la stessa anche con l'architettura x86 a 64 bit.

I documenti che descrivono lo standard del linguaggio C, definiscono la «dimensione» di una variabile come ***rango*** (*rank*).

66.1.3.1 Bit, byte e caratteri

«

A proposito della gestione delle variabili, esistono pochi concetti che sembrano rimanere stabili nel tempo. Il riferimento più importante in assoluto è il byte, che per il linguaggio C è almeno di 8 bit, ma potrebbe essere più grande.[5] Dal punto di vista del linguaggio C, il byte è l'elemento più piccolo che si possa indirizzare nella memoria centrale, questo anche quando la memoria fosse organizzata effettivamente a parole di dimensione maggiore del byte. Per esempio, in un elaboratore che suddivide la memoria in blocchi da 36 bit, si potrebbero avere byte da 9, 12, 18 bit o addirittura 36 bit.[6]

Una volta definito il byte, si considera che il linguaggio C rappresenti ogni variabile scalare come una sequenza continua di byte; pertanto, tutte le variabili scalari sono rappresentate come multipli di byte; di conseguenza anche le variabili strutturate lo sono, con la differenza che in tal caso potrebbero inserirsi dei «buchi» (in byte), dovuti alla necessità di allineare i dati in qualche modo.

Il tipo '**`char`**' (carattere), indifferentemente se si considera o meno il segno, rappresenta tradizionalmente una variabile numerica che occupa esattamente un byte, pertanto, spesso si confondono i termini «carattere» e «byte», nei documenti che descrivono il linguaggio C.

A causa della capacità limitata che può avere una variabile di tipo '**`char`**', il linguaggio C distingue tra un insieme di caratteri «minimo» e un insieme «esteso», da rappresentare però in altra forma.

66.1.3.2 Tipi primitivi

I tipi di dati primitivi rappresentano un valore **numerico** singolo, nel senso che anche il tipo '**char**' viene trattato come un numero. Il loro elenco essenziale si trova nella tabella 66.9.

Tabella 66.9. Elenco dei tipi comuni di dati primitivi elementari in C.

Tipo	Descrizione
`char`	Carattere (generalmente di 8 bit).
`int`	Intero normale.
`float`	Virgola mobile a precisione singola.
`double`	Virgola mobile a precisione doppia.

Come già accennato, non si può stabilire in modo generale quali siano le dimensioni esatte in bit dei vari tipi di dati, ovvero il rango, in quanto l'elemento certo è solo la relazione tra loro.

$$\text{char} \leq \text{int} \leq \text{float} \leq \text{double}$$

Questi tipi primitivi possono essere estesi attraverso l'uso di alcuni qualificatori: '**short**', '**long**', '**long long**', '**signed**'[7] e '**unsigned**'.[8] I primi tre si riferiscono al rango, mentre gli altri modificano il modo di valutare il contenuto di alcune variabili. La tabella 66.11 riassume i vari tipi primitivi con le combinazioni ammissibili dei qualificatori.

Tabella 66.11. Elenco dei tipi comuni di dati primitivi in C assieme ai qualificatori usuali.

Tipo	Abbreviazione	Descrizione
`char`		Tipo '**`char`**' per il quale non conta sapere se il segno viene considerato o meno.
`signed char`		Tipo '**`char`**' usato numericamente con segno.
`unsigned char`		Tipo '**`char`**' usato numericamente senza segno.
`short int` `signed short int`	`short` `signed short`	Intero più breve di '**`int`**', con segno.
`unsigned short int`	`unsigned short`	Tipo '**`short`**' senza segno.
`int` `signed int`		Intero normale, con segno.
`unsigned int`	`unsigned`	Tipo '**`int`**' senza segno.
`long int` `signed long int`	`long` `signed long`	Intero più lungo di '**`int`**', con segno.
`unsigned long int`	`unsigned long`	Tipo '**`long`**' senza segno.
`long long int` `signed long long int`	`long long` `signed long long`	Intero più lungo di '**`long int`**', con segno.

Tipo	Abbreviazione	Descrizione
`unsigned long long int`	`unsigned long long`	Tipo '**`long long`**' senza segno.
`float`		Tipo a virgola mobile a precisione singola.
`double`		Tipo a virgola mobile a precisione doppia.
`long double`		Tipo a virgola mobile «più lungo» di '**`double`**'.

Così, il problema di stabilire le relazioni di rango si complica:

$$\text{char} \leq \text{short} \leq \text{int} \leq \text{long} \leq \text{long long}$$

$$\text{float} \leq \text{double} \leq \text{long double}$$

I tipi '**`long`**' e '**`float`**' potrebbero avere un rango uguale, altrimenti non è detto quale dei due sia più grande.

Il programma seguente, potrebbe essere utile per determinare il rango dei vari tipi primitivi nella propria piattaforma.[9]

Listato 66.13. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/NxyS6KVy* , *http://ideone.com/uSVC3* .

```
#include <stdio.h>

int main (void)
{
    printf ("char           %i\n", (int) sizeof (char));
    printf ("short int      %i\n", (int) sizeof (short int));
    printf ("int            %i\n", (int) sizeof (int));
    printf ("long int       %i\n", (int) sizeof (long int));
    printf ("long long int  %i\n", (int) sizeof (long long int));
    printf ("float          %i\n", (int) sizeof (float));
    printf ("double         %i\n", (int) sizeof (double));
```

```
    printf ("long double   %i\n", (int) sizeof (long double));
    return 0;
}
```

Il risultato potrebbe essere simile a quello seguente:

```
char           1
short int      2
int            4
long int       4
long long int  8
float          4
double         8
long double    12
```

I numeri rappresentano la quantità di caratteri, nel senso di valori '**char**', per cui il tipo '**char**' dovrebbe sempre avere una dimensione unitaria.[10]

I tipi primitivi di variabili mostrati sono tutti utili alla memorizzazione di valori numerici, a vario titolo. A seconda che il valore in questione sia trattato con segno o senza segno, varia lo spettro di valori che possono essere contenuti.

Nel caso di interi ('**char**', '**short**', '**int**', '**long**' e '**long long**'), la variabile può essere utilizzata per tutta la sua estensione a contenere un numero binario. Pertanto, quando la rappresentazione è senza segno, il massimo valore ottenibile è $(2^{n})-1$, dove n rappresenta il numero di bit a disposizione. Quando invece si vuole trattare il dato come un numero con segno, il valore numerico massimo ottenibile è circa la metà (se si usa la rappresentazione dei valori negativi in complemento a due, l'intervallo di valori va da $(2^{n-1})-1$ a $-(2^{n-1})$)

Nel caso di variabili a virgola mobile non c'è più la possibilità di rappresentare esclusivamente valori senza segno; inoltre, più che esserci un limite nella grandezza rappresentabile (che comunque esiste), c'è soprattutto un limite nel grado di approssimazione.

Le variabili '**char**' sono fatte, in linea di principio, per contenere il codice di rappresentazione di un carattere, secondo la codifica utilizzata nel sistema. Generalmente si tratta di un dato di 8 bit, ma non è detto che debba sempre essere così. A ogni modo, il fatto che questa variabile possa essere gestita in modo numerico, permette una facile conversione da lettera a codice numerico corrispondente.

Un tipo di valore che non è stato ancora visto è quello logico: *Vero* è rappresentato da un qualsiasi valore numerico intero diverso da zero, mentre *Falso* corrisponde a zero.

66.1.3.3 Costanti letterali comuni

Quasi tutti i tipi di dati primitivi hanno la possibilità di essere rappresentati in forma di costante letterale. In particolare, si distingue tra:

- costanti carattere, rappresentate da un carattere alfanumerico racchiuso tra apici singoli, come '**'A'**', '**'B'**',...;
- costanti intere, rappresentate da un numero senza decimali, e a seconda delle dimensioni può trattarsi di uno dei vari tipi di interi (escluso '**char**');
- costanti con virgola, rappresentate da un numero con decimali (un punto seguito da altre cifre, anche se si tratta solo di zeri)

che, indipendentemente dalle dimensioni, di norma sono di tipo '**`double`**'.

Per esempio, 123 è generalmente una costante '**`int`**', mentre 123.0 è una costante '**`double`**'.

Le costanti che esprimono valori interi possono essere rappresentate con diverse basi di numerazione, attraverso l'indicazione di un prefisso: '**`0`*n***', dove ***n*** contiene esclusivamente cifre da zero a sette, viene inteso come un numero in base otto; '**`0x`*n***' o '**`0X`*n***', dove ***n*** può contenere le cifre numeriche consuete, oltre alle lettere da «A» a «F» (minuscole o maiuscole, indifferentemente) viene trattato come un numero in base sedici; negli altri casi, un numero composto con cifre da zero a nove è interpretato in base dieci.

Per quanto riguarda le costanti che rappresentano numeri con virgola, oltre alla notazione '***intero*`.`*decimali***' si può usare la «notazione scientifica». Per esempio, '**`7e+15`**' rappresenta l'equivalente di $7 \cdot (10^{15})$, cioè un sette con 15 zeri. Nello stesso modo, '**`7e-5`**', rappresenta l'equivalente di $7 \cdot (10^{-5})$, cioè 0,00007.

Il tipo di rappresentazione delle costanti numeriche, intere o con virgola, può essere specificato aggiungendo un suffisso, costituito da una o più lettere, come si vede nelle tabelle successive. Per esempio, '**`123UL`**' è un numero di tipo '**`unsigned long int`**', mentre '**`123.0F`**' è un tipo '**`float`**'. Si osservi che il suffisso può essere composto, indifferentemente, con lettere minuscole o maiuscole.

Tabella 66.15. Suffissi per le costanti che esprimono valori interi.

Suffisso	Descrizione
assente	In tal caso si tratta di un intero «normale» o più grande, se necessario.
`U`	Tipo senza segno (‘**`unsigned`**’).
`L`	Intero più grande della dimensione normale (‘**`long`**’).
`LL`	Intero molto più grande della dimensione normale (‘**`long long`**’).
`UL`	Intero senza segno, più grande della dimensione normale (‘**`unsigned long`**’).
`ULL`	Intero senza segno, molto più grande della dimensione normale (‘**`unsigned long long`**’).

Tabella 66.16. Suffissi per le costanti che esprimono valori con virgola.

Suffisso	Descrizione
assente	Tipo ‘**`double`**’.
`F`	Tipo ‘**`float`**’.
`L`	Tipo ‘**`long double`**’.

È possibile rappresentare anche le stringhe in forma di costante attraverso l’uso degli apici doppi, ma la stringa non è un tipo di dati primitivo, trattandosi piuttosto di un array di caratteri. Per il momento è importante fare attenzione a non confondere il tipo ‘**`char`**’ con la stringa. Per esempio, ‘**`'F'`**’ è un carattere (con un proprio valore numerico), mentre ‘**`"F"`**’ è una stringa, ma la differenza tra i due è

notevole. Le stringhe vengono descritte nella sezione 66.5.

I caratteri privi di rappresentazione grafica possono essere indicati, principalmente, attraverso tre tipi di notazione: ottale, esadecimale e simbolica. In tutti i casi si utilizza la barra obliqua inversa (‘****’) come carattere di escape, cioè come simbolo per annunciare che ciò che segue immediatamente deve essere interpretato in modo particolare.

La notazione ottale usa la forma ‘***ooo***’, dove ogni lettera ***o*** rappresenta una cifra ottale. A questo proposito, è opportuno notare che se la dimensione di un carattere fosse superiore ai fatidici 8 bit, occorrerebbero probabilmente più cifre (una cifra ottale rappresenta un gruppo di 3 bit).

La notazione esadecimale usa la forma ‘**\x*hh***’, dove ***h*** rappresenta una cifra esadecimale. Anche in questo caso vale la considerazione per cui ci vogliono più di due cifre esadecimali per rappresentare un carattere più lungo di 8 bit.

Dovrebbe essere logico, ma è il caso di osservare che la corrispondenza dei caratteri con i rispettivi codici numerici dipende dalla codifica utilizzata. Generalmente si utilizza la codifica ASCII, riportata anche nella sezione 47.7.5 (in questa fase introduttiva si omette di trattare la rappresentazione dell’insieme di caratteri universale).

La notazione simbolica permette di fare riferimento facilmente a codici di uso comune, quali <*CR*>, <*HT*>,... Inoltre, questa notazione permette anche di indicare caratteri che altrimenti verrebbero interpretati in maniera differente dal compilatore. La tabella 66.17 riporta i vari tipi di rappresentazione delle costanti carattere attraverso codici di escape.

Tabella 66.17. Elenco dei modi di rappresentazione delle costanti carattere attraverso codici di escape.

Codice	ASCII	Altra codifica
`\`***ooo***	Notazione ottale in base alla codifica.	idem
`\x`***hh***	Notazione esadecimale in base alla codifica.	idem
`\\`	Una singola barra obliqua inversa (‘****’).	idem
`\'`	Un apice singolo destro.	idem
`\"`	Un apice doppio.	idem
`\?`	Un punto interrogativo (per impedire che venga inteso come parte di una sequenza triplice, o *trigraph*).	idem
`\0`	Il codice <*NUL*>.	Il carattere nullo (con tutti i bit a zero).
`\a`	Il codice <*BEL*> (*bell*).	Il codice che, rappresentato sullo schermo o sulla stampante, produce un segnale acustico (*alert*).
`\b`	Il codice <*BS*> (*backspace*).	Il codice che fa arretrare il cursore di una posizione nella riga (*backspace*).
`\f`	Il codice <*FF*> (*form feed*).	Il codice che fa avanzare il cursore all’inizio della prossima pagina logica (*form feed*).

Codice	ASCII	Altra codifica
`\n`	Il codice <*LF*> (*line feed*).	Il codice che fa avanzare il cursore all'inizio della prossima riga logica (*new line*).
`\r`	Il codice <*CR*> (*carriage return*).	Il codice che porta il cursore all'inizio della riga attuale (*carriage return*).
`\t`	Una tabulazione orizzontale (<*HT*>).	Il codice che porta il cursore all'inizio della prossima tabulazione orizzontale (*horizontal tab*).
`\v`	Una tabulazione verticale (<*VT*>).	Il codice che porta il cursore all'inizio della prossima tabulazione verticale (*vertical tab*).

A parte i casi di '***ooo***' e '**\x*hh***', le altre sequenze esprimono un concetto, piuttosto di un codice numerico preciso. All'origine del linguaggio C, tutte le altre sequenze corrispondono a un solo carattere non stampabile, ma attualmente non è più garantito che sia così. In particolare, la sequenza '**\n**', nota come *new-line*, potrebbe essere espressa in modo molto diverso rispetto al codice <*LF*> tradizionale. Questo concetto viene comunque approfondito a proposito della gestione dei flussi di file.

In varie situazioni, il linguaggio C standard ammette l'uso di sequenze composte da due o tre caratteri, note come *digraph* e *trigraph* rispettivamente; ciò in sostituzione di simboli la cui rappresentazione, in quel contesto, può essere impossibile. In un sistema che ammetta almeno l'uso della codifica ASCII per scrivere il file sorgente, con l'ausilio di una tastiera comune, non c'è alcun bisogno di usare tali artifici, i quali, se usati, renderebbero estremamente complessa la lettura del sorgente. Pertanto, è bene sapere che esistono queste cose, ma è meglio non usarle mai. Tuttavia, siccome le sequenze a tre caratteri (*trigraph*) iniziano con una coppia di punti interrogativi, se in una stringa si vuole rappresentare una sequenza del genere, per evitare che il compilatore la traduca diversamente, è bene usare la sequenza '**\?\?**', come suggerisce la tabella 66.17.

Nell'esempio introduttivo appare già la notazione '**\n**' per rappresentare l'inserzione di un codice di interruzione di riga alla fine del messaggio di saluto:

```
...
  printf ("Ciao mondo!\n");
...
```

Senza di questo, il cursore resterebbe a destra del messaggio alla fine dell'esecuzione di quel programma, ponendo lì l'invito.

66.1.3.4 Valore numerico delle costanti carattere

Il linguaggio C distingue tra i caratteri di un insieme fondamentale e ridotto, da quelli dell'insieme di caratteri universale (ISO 10646). Il gruppo di caratteri ridotto deve essere rappresentabile in una variabi-

le ‘**char**’ (descritta nelle sezioni successive) e può essere gestito direttamente in forma numerica, se si conosce il codice corrispondente a ogni simbolo (di solito si tratta della codifica ASCII).

Se si può essere certi che nella codifica le lettere dell’alfabeto latino siano disposte esattamente in sequenza (come avviene proprio nella codifica ASCII), si potrebbe scrivere ‘**’A’+1**’ e ottenere l’equivalente di ‘**’B’**’. Tuttavia, lo standard prescrive che sia garantito il funzionamento solo per le cifre numeriche. Pertanto, per esempio, ‘**’0’+3**’ (zero espresso come carattere, sommato a un tre numerico) deve essere equivalente a ‘**’3’**’ (ovvero un «tre» espresso come carattere).

Listato 66.19. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/yrc2S7Xv* , *http://ideone.com/HCvsD* .

```
#include <stdio.h>

int main (void)
{
  char c;
  for (c = '0'; c <= 'Z'; c++)
    {
      printf ("%c", c);
    }
  printf ("\n");
  return 0;
}
```

Il programma di esempio che si vede nel listato appena mostrato, se prodotto per un ambiente in cui si utilizza la codifica ASCII, genera il risultato seguente:

```
0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

66.1.3.5 Campo di azione delle variabili

Il campo di azione delle variabili in C viene determinato dalla posizione in cui queste vengono dichiarate e dall'uso di qualificatori particolari. Nella fase iniziale dello studio del linguaggio basta considerare, approssimativamente, che quanto dichiarato all'interno di una funzione ha valore locale per la funzione stessa, mentre quanto dichiarato al di fuori, ha valore globale per tutto il file. Pertanto, in questo capitolo si usano genericamente le definizioni di «variabile locale» e «variabile globale», senza affrontare altre questioni. Nella sezione 66.3 viene trattato questo argomento con maggiore dettaglio.

66.1.3.6 Dichiarazione delle variabili

La dichiarazione di una variabile avviene specificando il tipo e il nome della variabile, come nell'esempio seguente dove viene creata la variabile ***numero*** di tipo intero:

```
int numero;
```

La variabile può anche essere inizializzata contestualmente, assegnandole un valore, come nell'esempio seguente in cui viene dichiarata la stessa variabile ***numero*** con il valore iniziale di 1000:

```
int numero = 1000;
```

Una costante è qualcosa che non varia e generalmente si rappresenta attraverso una notazione che ne definisce il valore, ovvero attraverso una costante letterale. Tuttavia, a volte può essere più comodo definire una costante in modo simbolico, come se fosse una variabile,

per facilitarne l'utilizzo e la sua identificazione all'interno del programma. Si ottiene questo con il modificatore '**const**'. Ovviamente, è obbligatorio inizializzala contestualmente alla sua dichiarazione. L'esempio seguente dichiara la costante simbolica ***pi*** con il valore del π:

```
const float pi = 3.14159265;
```

Le costanti simboliche di questo tipo, sono delle variabili per le quali il compilatore non concede che avvengano delle modifiche; pertanto, il programma eseguibile che si ottiene potrebbe essere organizzato in modo tale da caricare questi dati in segmenti di memoria a cui viene lasciato poi il solo permesso di lettura.

Tradizionalmente, l'uso di costanti simboliche di questo tipo è stato limitato, preferendo delle ***macro-variabili*** definite e gestite attraverso il precompilatore (come viene descritto nella sezione 66.2). Tuttavia, un compilatore ottimizzato è in grado di gestire al meglio le costanti definite nel modo illustrato dall'esempio, utilizzando anche dei valori costanti letterali nella trasformazione in linguaggio assemblatore, rendendo così indifferente, dal punto di vista del risultato, l'alternativa delle macro-variabili. Pertanto, la stessa guida *GNU coding standards* chiede di definire le costanti come variabili-costanti, attraverso il modificatore '**const**'.

66.1.3.7 Variabili costanti e variabili volatili

«

Come già descritto nella sezione precedente, una variabile può essere dichiarata con il modificatore '**const**' per sottolineare al compilatore che non deve essere modificata nel corso del programma, salva la possibilità di inizializzarla contestualmente alla sua dichiarazione.

```
const float pi = 3.14159265;
```

All'opposto della costante si può considerare un'area di memoria a cui accedono programmi differenti, in modo asincrono, ognuno con la facoltà di modificarla a proprio piacimento, oppure un'area che viene modificata direttamente dall'hardware. In questi casi, ovvero quando il compilatore non deve attuare delle semplificazioni che partano dalla presunzione del contenuto di una certa variabile, si usa il modificatore '**volatile**'. Si osservi l'esempio seguente:

```
…
volatile int i;
…
i = 1;
if (i > 0)
  {
    …
  }
else
  {
    …
  }
…
```

Anche se alla variabile ***i*** viene assegnato il valore uno, il compilatore non può escludere che nel momento della verifica della variabile questa abbia invece un valore differente. In altri termini, se la variabile ***i*** venisse dichiarata in modo normale, un compilatore ottimizzato potrebbe escludere le istruzioni sotto il controllo della parola chiave '**else**'.

Quando l'area di memoria che viene considerata «volatile», deve essere modificata da un processo estraneo, mentre il programma si

limita semplicemente a leggerne il contenuto prendendo atto del valore che ha, la variabile può essere dichiarata simultaneamente con i modificatori '**const**' e '**volatile**', come nell'esempio seguente, dove, tra l'altro, si presume che la variabile in questione sia definita in un altro file-oggetto:

```
extern const volatile int variabile;
…
```

66.1.3.8 Il tipo indefinito: «void»

«

Lo standard del linguaggio C definisce un tipo particolare di variabile, individuato dalla parola chiave '**void**': si tratta di un tipo che rappresenta una variabile di rango nullo; la quale, come tale, non può contenere alcun valore.

66.1.4 Operatori ed espressioni

«

L'operatore è qualcosa che esegue un qualche tipo di funzione, su uno o più operandi, restituendo un valore.[11] Il valore restituito è di tipo diverso a seconda degli operandi utilizzati. Per esempio, la somma di due interi genera un risultato intero. Gli operandi descritti di seguito sono quelli più comuni e importanti.

Le espressioni sono formate spesso dalla valutazione di sottoespressioni (espressioni più piccole). Va osservato che ci sono circostanze in cui il contesto non impone che ci sia un solo ordine possibile nella valutazione delle sottoespressioni, ma il programmatore deve tenere conto di questa possibilità, per evitare che il risultato dipenda dalle scelte non prevedibili del compilatore.

Tabella 66.27. Ordine di precedenza tra gli operatori previsti nel linguaggio C. Gli operatori sono raggruppati a livelli di priorità equivalente, partendo dall'alto con la priorità maggiore, scendendo progressivamente alla priorità minore. Le variabili ***a***, ***b*** e ***c*** rappresentano la collocazione delle sottoespressioni da considerare ed esprimono l'ordine di associatività: prima ***a***, poi ***b***, poi ***c***.

Operatori	Annotazioni
`(`***a***`)` `[`***a***`]` ***a*`->`***b*** ***a*`.`***b***	Le parentesi tonde usate per raggruppare una porzione di espressione hanno la precedenza su ogni altro operatore. Le parentesi quadre riguardano gli array; gli operatori '**->**' e '**.**', riguardano le strutture e le unioni.
`!`***a*** `~`***a*** `++`***a*** `--`***a*** `+`***a*** `-`***a*** `*`***a*** `&`***a*** `(`***tipo***`)` `sizeof` ***a***	Gli operatori '**+**' e '**-**' di questo livello sono da intendersi come «unari», ovvero si riferiscono al segno di quanto appare alla loro destra. Gli operatori '*****' e '**&**' di questo livello riguardano la gestione dei puntatori; le parentesi tonde si riferiscono al cast.
a*`*`b*** ***a*`/`***b*** ***a*`%`***b***	Moltiplicazione, divisione e resto della divisione intera.
a*`+`b*** ***a*`-`***b***	Somma e sottrazione.
a*`<<`b*** ***a*`>>`***b***	Scorrimento binario.
a*`<`b*** ***a*`<=`***b*** ***a*`>`***b*** ***a*`=>`***b***	Confronto.
a*`==`b*** ***a*`!=`***b***	Confronto.
a*`&`b***	AND bit per bit.

Operatori	Annotazioni
a^*b*	XOR bit per bit.
a|*b*	OR bit per bit.
a&&*b*	AND nelle espressioni logiche.
a||*b*	OR nelle espressioni logiche.
c?*b*:*a*	Operatore condizionale
b=*a* *b*+=*a* *b*-=*a* *b**=*a* *b*/=*a* *b*%=*a* *b*&=*a* *b*^=*a* *b*|=*a* *b*<<=*a* *b*>>=*a*	Operatori di assegnamento.
a, *b*	Sequenza di espressioni (espressione multipla).

66.1.4.1 Operatori aritmetici

«

Gli operatori che intervengono su valori numerici sono elencati nella tabella 66.28. Per dare un significato alle descrizioni della tabella, occorre tenere presenta una caratteristica importante del linguaggio, per la quale, la maggior parte delle espressioni restituisce un valore. Per esempio, '`b = a = 1`' fa sì che la variabile ***a*** ottenga il valore 1 e che, successivamente, la variabile ***b*** ottenga il valore di ***a***. In questo senso, al problema dell'ordine di precedenza dei vari operatori si aggiunge anche l'ordine in cui le espressioni restituiscono un valo-

re. Per esempio, '`d = e++`' comporta l'incremento di una unità del contenuto della variabile ***e***, ma ciò solo **dopo** averne restituito il valore che viene assegnato alla variabile ***d***. Pertanto, se inizialmente la variabile ***e*** contiene il valore 1, dopo l'elaborazione dell'espressione completa, la variabile ***d*** contiene il valore 1, mentre la variabile ***e*** contiene il valore 2.

Tabella 66.28. Elenco degli operatori aritmetici e di quelli di assegnamento relativi a valori numerici.

Operatore e operandi	Descrizione
$++op$	Incrementa di un'unità l'operando prima che venga restituito il suo valore.
$op++$	Incrementa di un'unità l'operando dopo averne restituito il suo valore.
$--op$	Decrementa di un'unità l'operando prima che venga restituito il suo valore.
$op--$	Decrementa di un'unità l'operando dopo averne restituito il suo valore.
$+op$	Non ha alcun effetto.
$-op$	Inverte il segno dell'operando (prima di restituirne il valore).
op1 + *op2*	Somma i due operandi.
op1 - *op2*	Sottrae dal primo il secondo operando.
op1 * *op2*	Moltiplica i due operandi.
op1 / *op2*	Divide il primo operando per il secondo.
op1 % *op2*	Calcola il resto della divisione tra il primo e il secondo operando, i quali devono essere costituiti da valori interi.

Operatore e operandi	Descrizione
var = *valore*	Assegna alla variabile il valore alla destra.
op1 += *op2*	*op1* = (*op1* + *op2*)
op1 -= *op2*	*op1* = (*op1* - *op2*)
op1 *= *op2*	*op1* = (*op1* * *op2*)
op1 /= *op2*	*op1* = (*op1* / *op2*)
op1 %= *op2*	*op1* = (*op1* % *op2*)

66.1.4.2 Operatori di confronto e operatori logici

«

Gli operatori di confronto determinano la relazione tra due operandi. Il risultato dell'espressione composta da due operandi posti a confronto è un numero intero ('**int**') e precisamente si ottiene uno se il confronto è valido e zero in caso contrario. Gli operatori di confronto sono elencati nella tabella 66.29.

Il linguaggio C non ha una rappresentazione specifica per i valori booleani *Vero* e *Falso*,[12] ma si limita a interpretare un valore pari a zero come *Falso* e un valore diverso da zero come *Vero*. Va osservato, quindi, che il numero usato come valore booleano, può essere espresso anche in virgola mobile, benché sia preferibile di gran lunga un intero normale.

Tabella 66.29. Elenco degli operatori di confronto. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
op1 `==` ***op2***	*Vero* se gli operandi si equivalgono.
op1 `!=` ***op2***	*Vero* se gli operandi sono differenti.
op1 `<` ***op2***	*Vero* se il primo operando è minore del secondo.
op1 `>` ***op2***	*Vero* se il primo operando è maggiore del secondo.
op1 `<=` ***op2***	*Vero* se il primo operando è minore o uguale al secondo.
op1 `>=` ***op2***	*Vero* se il primo operando è maggiore o uguale al secondo.

Quando si vogliono combinare assieme diverse espressioni logiche, comprendendo in queste anche delle variabili che contengono un valore booleano, si utilizzano gli operatori logici (noti normalmente come: AND, OR, NOT, ecc.). Il risultato di un'espressione logica complessa è quello dell'ultima espressione elementare valutata effettivamente, in quanto le sottoespressioni che non possono cambiare l'esito della condizione complessiva non vengono valutate. Gli operatori logici sono elencati nella tabella 66.30.

Tabella 66.30. Elenco degli operatori logici. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
`!` ***op***	Inverte il risultato logico dell'operando.
op1 `&&` ***op2***	Se il risultato del primo operando è *Falso* non valuta il secondo.
op1 `||` ***op2***	Se il risultato del primo operando è *Vero* non valuta il secondo.

Un tipo particolare di operatore logico è l'operatore condizionale, il quale permette di eseguire espressioni diverse in relazione al risultato di una condizione. La sua sintassi si esprime nel modo seguente:

```
condizione ? espressione1 : espressione2
```

In pratica, se l'espressione che rappresenta la condizione si avvera, viene eseguita la prima espressione che segue il punto interrogativo, altrimenti viene eseguita quella che segue i due punti.

66.1.4.3 Operatori binari

Il linguaggio C consente di eseguire alcune operazioni binarie, sui **valori interi**, come spesso è possibile fare con un linguaggio assemblatore, anche se non è possibile interrogare degli indicatori (*flag*) che informino sull'esito delle azioni eseguite. Sono disponibili le operazioni elencate nella tabella 66.31.

Tabella 66.31. Elenco degli operatori binari. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
op1 `&` ***op2***	AND bit per bit.
op1 `|` ***op2***	OR bit per bit.
op1 `^` ***op2***	XOR bit per bit (OR esclusivo).
op1 `<<` ***op2***	Scorrimento a sinistra di ***op2*** bit (con ***op2*** che rappresenta un valore positivo o senza segno). A destra vengono aggiunti bit a zero
op1 `>>` ***op2***	Scorrimento a destra di ***op2*** bit (con ***op2*** che rappresenta un valore positivo o senza segno). Il valore dei bit aggiunti a sinistra potrebbe tenere conto del segno.
`~`***op1***	Complemento a uno.
op1 `&=` ***op2***	***op1*** `=` `(`***op1*** `&` ***op2***`)`
op1 `|=` ***op2***	***op1*** `=` `(`***op1*** `|` ***op2***`)`
op1 `^=` ***op2***	***op1*** `=` `(`***op1*** `^` ***op2***`)`
op1 `<<=` ***op2***	***op1*** `=` `(`***op1*** `<<` ***op2***`)`
op1 `>>=` ***op2***	***op1*** `=` `(`***op1*** `>>` ***op2***`)`
op1 `~=` ***op2***	***op1*** `=` `~`***op2***

A seconda del compilatore e della piattaforma, lo scorrimento a destra potrebbe essere di tipo aritmetico, ovvero potrebbe tenere conto

del segno del valore che viene fatto scorrere. Pertanto, non potendo fare affidamento su questa ipotesi, è bene che i valori di cui si fa lo scorrimento a destra siano sempre senza segno, o comunque positivi.

Per aiutare a comprendere l'uso degli operatori binari vengono mostrati alcuni esempi. In particolare si utilizzano due operandi di tipo '**`char`**' (a 8 bit) senza segno: ***a*** contenente il valore 42, pari a 00101010_2; ***b*** contenente il valore 51, pari a 00110011_2.

`c = a & b`

00101010_2	(42_{10})	AND
00110011_2	(51_{10})	=
00100010_2	(34_{10})	

`c = a | b`

00101010_2	(42_{10})	OR
00110011_2	(51_{10})	=
00111011_2	(59_{10})	

`c = a ^ b`

00101010_2	(42_{10})	XOR
00110011_2	(51_{10})	=
00011001_2	(25_{10})	

Lo scorrimento, invece, viene mostrato sempre solo per una singola unità: ***a*** contenente il valore 42; ***b*** contenente il valore 1.

`c = a << b`

00101010_2	(42_{10})	<<
00000001_2	(1_{10})	=
01010100_2	(84_{10})	

`c = a >> b`

00101010_2	(42_{10})	>>
00000001_2	(1_{10})	=
00010101_2	(21_{10})	

`c = ~a`

00101010_2	(42_{10})
11010101_2	(213_{10})

66.1.4.4 Conversione di tipo

Quando si assegna un valore a una variabile, nella maggior parte dei casi, il contesto stabilisce il tipo di questo valore in modo corretto. Di fatto, è il tipo della variabile ricevente che stabilisce la conversione necessaria. Tuttavia, il problema si pone anche durante la valutazione di un'espressione.

Per esempio, '**5/4**' viene considerata la divisione di due interi e, di conseguenza, l'espressione restituisce un valore intero, cioè 1. Diverso sarebbe se si scrivesse '**5.0/4.0**', perché in questo caso si tratterebbe della divisione tra due numeri a virgola mobile (per la precisione, di tipo '**double**') e il risultato è un numero a virgola mobile.

Quando si pone il problema di risolvere l'ambiguità si utilizza esplicitamente la conversione del tipo, attraverso un ***cast***:

```
(tipo) espressione
```

In pratica, si deve indicare tra parentesi tonde il nome del tipo di dati in cui deve essere convertita l'espressione che segue. Il problema sta nella precedenza che ha il cast nell'insieme degli altri operatori e in generale conviene utilizzare altre parentesi per chiarire la relazione che ci deve essere.

```
int x = 10;
double y;
…
y = (double) x/9;
```

In questo caso, la variabile intera ***x*** viene convertita nel tipo '**double**' (a virgola mobile) prima di eseguire la divisione. Dal momento che il cast ha precedenza sull'operazione di divisione, non si pongono problemi, inoltre, la divisione avviene trasformando implicitamente il 9 intero in un 9,0 di tipo '**double**'. In pratica, l'operazione avviene utilizzando valori '**double**' e restituendo un risultato '**double**'.

66.1.4.5 Espressioni multiple

«

Un'istruzione, cioè qualcosa che termina con un punto e virgola, può contenere diverse espressioni separate da una virgola. Tenendo presente che in C l'assegnamento di una variabile è anche un'espressione, la quale restituisce il valore assegnato, si veda l'esempio seguente:

```
int x;
int y;
…
y = 10, x = 20, y = x*2;
```

L'esempio mostra un'istruzione contenente tre espressioni: la prima assegna a ***y*** il valore 10, la seconda assegna a ***x*** il valore 20 e la terza sovrascrive ***y*** assegnandole il risultato del prodotto ***x***·2. In pratica, alla fine la variabile ***y*** contiene il valore 40 e ***x*** contiene 20.

Un'espressione multipla, come quella dell'esempio, restituisce il valore dell'ultima a essere eseguita. Tornando all'esempio appena visto, gli si può apportare una piccola modifica per comprendere il concetto:

```
int x;
int y;
int z;
…
z = (y = 10, x = 20, y = x*2);
```

La variabile ***z*** si trova a ricevere il valore dell'espressione '***y = x*2***', perché è quella che viene eseguita per ultima nel gruppo raccolto tra parentesi.

A proposito di «espressioni multiple» vale la pena di ricordare ciò

che accade con gli assegnamenti multipli, con l'esempio seguente:

```
y = x = 10;
```

Qui si vede l'assegnamento alla variabile ***y*** dello stesso valore che viene assegnato alla variabile ***x***. In pratica, sia ***x***, sia ***y***, contengono alla fine il numero 10, perché le precedenze sono tali che è come se fosse scritto: '***y = (x = 10)***'.

66.1.5 Strutture di controllo di flusso

Il linguaggio C gestisce praticamente tutte le strutture di controllo di flusso degli altri linguaggi di programmazione, compreso *go-to* che comunque è sempre meglio non utilizzare e qui, volutamente, non viene presentato.

Le strutture di controllo permettono di sottoporre l'esecuzione di una parte di codice alla verifica di una condizione, oppure permettono di eseguire dei cicli, sempre sotto il controllo di una condizione. La parte di codice che viene sottoposta a questo controllo, può essere una singola istruzione, oppure un gruppo di istruzioni (precisamente si chiamerebbe istruzione composta). Nel secondo caso, è necessario delimitare questo gruppo attraverso l'uso delle parentesi graffe.

Dal momento che è comunque consentito di realizzare un gruppo di istruzioni che in realtà ne contiene una sola, probabilmente è meglio utilizzare sempre le parentesi graffe, in modo da evitare equivoci nella lettura del codice. Dato che le parentesi graffe sono usate nel codice C, se queste appaiono nei modelli sintattici indicati, significa che fanno parte delle istruzioni e non della sintassi.

Negli esempi, i rientri delle parentesi graffe seguono le indicazioni della guida *GNU coding standards*.

66.1.5.1 Struttura condizionale: «if»

La struttura condizionale è il sistema di controllo fondamentale dell'andamento del flusso delle istruzioni.

```
if (condizione) istruzione
```

```
if (condizione) istruzione else istruzione
```

Se la condizione si verifica, viene eseguita l'istruzione o il gruppo di istruzioni che segue; quindi il controllo passa alle istruzioni successive alla struttura. Se viene utilizzata la sotto-struttura che si articola a partire dalla parola chiave '**else**', nel caso non si verifichi la condizione, viene eseguita l'istruzione che ne dipende. Sotto vengono mostrati alcuni esempi.

```
int i_importo;
…
if (i_importo > 10000000) printf ("L'offerta è vantaggiosa\n");
```

```
int i_importo;
int i_memorizza;
…
if (i_importo > 10000000)
  {
    i_memorizza = i_importo;
    printf ("L'offerta è vantaggiosa\n");
  }
else
  {
    printf ("Lascia perdere\n");
  }
```

L'esempio successivo, in particolare, mostra un modo grazioso per allineare le sottocondizioni, senza eccedere negli annidamenti:

```
int i_importo;
int i_memorizza;
…
if (i_importo > 10000000)
  {
    i_memorizza = i_importo;
    printf ("L'offerta è vantaggiosa\n");
  }
else if (i_importo > 5000000)
  {
    i_memorizza = i_importo;
    printf ("L'offerta è accettabile\n");
  }
else
  {
    printf ("Lascia perdere\n");
  }
```

66.1.5.2 Struttura di selezione: «switch»

La struttura di selezione che si attua con l'istruzione '**`switch`**', è un po' troppo complessa per essere rappresentata facilmente attraverso uno schema sintattico. In generale, questa struttura permette di **saltare** a una certa posizione della struttura, in base al risultato di un'espressione. L'esempio seguente mostra la visualizzazione del nome del mese, in base al valore di un intero.

```
int i_mese;
…
switch (i_mese)
  {
```

```
    case 1: printf ("gennaio\n"); break;
    case 2: printf ("febbraio\n"); break;
    case 3: printf ("marzo\n"); break;
    case 4: printf ("aprile\n"); break;
    case 5: printf ("maggio\n"); break;
    case 6: printf ("giugno\n"); break;
    case 7: printf ("luglio\n"); break;
    case 8: printf ("agosto\n"); break;
    case 9: printf ("settembre\n"); break;
    case 10: printf ("ottobre\n"); break;
    case 11: printf ("novembre\n"); break;
    case 12: printf ("dicembre\n"); break;
  }
```

Come si vede, dopo l'istruzione con cui si emette il nome del mese attraverso lo standard output, viene richiesta l'interruzione esplicita dell'analisi della struttura, attraverso l'istruzione '**`break`**', perché altrimenti verrebbero eseguite le istruzioni del caso successivo, se presente. Infatti, un gruppo di casi può essere raggruppato assieme, quando si vuole che ognuno di questi esegua lo stesso insieme di istruzioni.

```
int i_anno;
int i_mese;
int i_giorni;
…
switch (i_mese)
  {
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
```

```
    case 12:
        i_giorni = 31;
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        i_giorni = 30;
        break;
    case 2:
        if (((i_anno % 4 == 0) && !(i_anno % 100 == 0)) ||
                (i_anno % 400 == 0))
            i_giorni = 29;
        else
            i_giorni = 28;
        break;
  }
```

È anche possibile dichiarare un caso predefinito che si verifichi quando nessuno degli altri si avvera.

```
int i_mese;
…
switch (i_mese)
  {
    case 1: printf ("gennaio\n"); break;
    case 2: printf ("febbraio\n"); break;
    …
    case 11: printf ("novembre\n"); break;
    case 12: printf ("dicembre\n"); break;
    default: printf ("mese non corretto\n"); break;
  }
```

La struttura di selezione che si ottiene con l'istruzione '**`switch`**' può

apparire disarmonica rispetto all'organizzazione del linguaggio C, per la presenza delle voci '**`case`** ***valore*** **`:`**'. Queste voci sono sostanzialmente delle «etichette» che individuano una posizione nel codice, da raggiungere in base al valore preso in considerazione per la selezione.

66.1.5.3 Iterazione con condizione di uscita iniziale: «while»

«

L'iterazione si ottiene normalmente in C attraverso l'istruzione '**`while`**', la quale esegue un'istruzione, o un gruppo di queste, finché la condizione continua a restituire il valore *Vero*. La condizione viene valutata prima di eseguire il gruppo di istruzioni e poi ogni volta che termina un ciclo, prima dell'esecuzione del successivo.

```
while (condizione) istruzione
```

L'esempio seguente fa apparire per 10 volte la lettera «x».

```
int i = 0;

while (i < 10)
  {
    i++;
    printf ("x");
  }
printf ("\n");
```

Nel blocco di istruzioni di un ciclo '**`while`**', ne possono apparire alcune particolari:

- '**`break`**', che serve a uscire definitivamente dalla struttura del ciclo;

- '**continue**', che serve a interrompere l'esecuzione del gruppo di istruzioni, riprendendo immediatamente con il ciclo successivo (a partire dalla valutazione della condizione).

L'esempio seguente è una variante del calcolo di visualizzazione mostrato sopra, modificato in modo da vedere il funzionamento dell'istruzione '**break**'. All'inizio della struttura, '**while (1)**' equivale a stabilire che il ciclo è senza fine, perché la condizione è sempre vera. In questo modo, solo la richiesta esplicita di interruzione dell'esecuzione della struttura (attraverso l'istruzione '**break**') permette l'uscita da questa.

```
int i = 0;

while (1)
  {
    if (i >= 10)
      {
        break;
      }
    i++;
    printf ("x");
  }
printf ("\n");
```

66.1.5.4 Iterazione con condizione di uscita finale: «do-while»

Una variante del ciclo '**while**', in cui l'analisi della condizione di uscita avviene dopo l'esecuzione del blocco di istruzioni che viene iterato, è definito dall'istruzione '**do**'.

```
do blocco_di_istruzioni while (condizione);
```

In questo caso, si esegue un gruppo di istruzioni una volta, poi se ne ripete l'esecuzione finché la condizione restituisce il valore *Vero*.

```
int i = 0;

do
  {
    i++;
    printf ("x");
  }
while (i < 10);
printf ("\n");
```

L'esempio mostrato è quello già usato nella sezione precedente, con l'adattamento necessario a utilizzare questa struttura di controllo.

La struttura di controllo '**do**...**while**' è in disuso, perché, generalmente, al suo posto si preferisce gestire i cicli di questo tipo attraverso una struttura '**while**', pura e semplice.

66.1.5.5 Ciclo enumerativo: «for»

«

In presenza di iterazioni in cui si deve incrementare o decrementare una variabile a ogni ciclo, si usa preferibilmente la struttura '**for**', che in C permetterebbe un utilizzo più ampio di quello comune:

```
for ([espressione1]; [espressione2]; [espressione3]) istruzione
```

La forma tipica di un'istruzione '**for**' è quella per cui la prima espressione corrisponde all'assegnamento iniziale di una variabile, la seconda a una condizione che deve verificarsi fino a che si vuole che sia eseguita l'istruzione (o il gruppo di istruzioni) e la terza all'incremento o decremento della variabile inizializzata con la prima espressione. In pratica, l'utilizzo normale del ciclo '**for**' potrebbe esprimersi nella sintassi seguente:

```
for (var = n; condizione; var++) istruzione
```

Il ciclo '**for**' potrebbe essere definito anche in maniera differente, più generale: la prima espressione viene eseguita una volta sola all'inizio del ciclo; la seconda viene valutata all'inizio di ogni ciclo e il gruppo di istruzioni viene eseguito solo se il risultato è *Vero*; l'ultima viene eseguita alla fine dell'esecuzione del gruppo di istruzioni, prima che si ricominci con l'analisi della condizione.

L'esempio già visto, in cui viene visualizzata per 10 volte una «x», potrebbe tradursi nel modo seguente, attraverso l'uso di un ciclo '**for**':

```
int i;

for (i = 0; i < 10; i++)
  {
    printf ("x");
  }
printf ("\n");
```

Anche nelle istruzioni controllate da un ciclo '**for**' si possono collocare istruzioni '**break**' e '**continue**', con lo stesso significato visto per il ciclo '**while**' e '**do**...**while**'.

Sfruttando la possibilità di inserire più espressioni in una singola istruzione, si possono realizzare dei cicli '**for**' molto più complessi, anche se questo è sconsigliabile per evitare di scrivere codice troppo difficile da interpretare. In questo modo, l'esempio precedente potrebbe essere ridotto a quello che segue, dove si usa un punto e virgola solitario per rappresentare un'istruzione nulla:

```
int i;

for (i = 0; i < 10; printf ("x"), i++)
  {
    ;
  }
printf ("\n");
```

Se si utilizzano istruzioni multiple, separate con la virgola, occorre tenere presente che **l'espressione che esprime la condizione deve rimanere singola** (se per la condizione si usasse un'espressione multipla, conterebbe solo la valutazione dell'ultima). Naturalmente, nel caso della condizione, si possono costruire condizioni complesse con l'ausilio degli operatori logici, ma rimane il fatto che l'operatore virgola ('`,`') non dovrebbe avere senso lì.

Nel modello sintattico iniziale si vede che le tre espressioni sono opzionali e rimane solo l'obbligo di mettere i punti e virgola relativi. L'esempio seguente mostra un ciclo senza fine che viene interrotto attraverso un'istruzione '**break**':

```
int i = 0;
for (;;)
  {
    if (i >= 10)
      {
        break;
      }
    printf ("x");
    i++;
  }
```

66.1.6 Funzioni

Il linguaggio C offre le funzioni come mezzo per realizzare la scomposizione del codice in subroutine. Prima di poter essere utilizzate attraverso una chiamata, le funzioni devono essere dichiarate, anche se non necessariamente descritte. In pratica, se si vuole indicare nel codice una chiamata a una funzione che viene descritta più avanti, occorre almeno dichiararne il prototipo.

Le funzioni del linguaggio C prevedono il passaggio di parametri solo **per valore**, con tutti i tipi di dati, esclusi gli array (che invece vanno passati per riferimento, attraverso il puntatore alla loro posizione iniziale in memoria).

Il linguaggio C, attraverso la libreria standard, offre un gran numero di funzioni comuni, i cui prototipi vengono incorporati nel codice attraverso l'inclusione di file di intestazione, con l'istruzione '**`#include`**' del precompilatore. Per esempio, come si è già visto, per poter utilizzare la funzione ***printf()*** si deve inserire la riga '**`#include <stdio.h>`**' nella parte iniziale del file sorgente.

66.1.6.1 Dichiarazione di un prototipo

«

Quando la descrizione di una funzione può essere fatta solo dopo l'apparizione di una sua chiamata, occorre dichiararne il prototipo all'inizio, secondo la sintassi seguente:

```
tipo nome ( [tipo [ nome ] [,...] ] );
```

Il tipo, posto all'inizio, rappresenta il tipo di valore che la funzione restituisce. Se la funzione non deve restituire alcunché, si utilizza il tipo '**void**'. Se la funzione utilizza dei parametri, il tipo di questi deve essere elencato tra le parentesi tonde. L'istruzione con cui si dichiara il prototipo termina regolarmente con un punto e virgola.

Lo standard C stabilisce che una funzione che non richiede parametri deve utilizzare l'identificatore '**void**' in modo esplicito, all'interno delle parentesi.

Segue la descrizione di alcuni esempi.

- ```
  int fattoriale (int);
  ```
  In questo caso, viene dichiarato il prototipo della funzione '**fattoriale**', che richiede un parametro di tipo '**int**' e restituisce anche un valore di tipo '**int**'.

- ```
  int fattoriale (int n);
  ```
  Come nell'esempio precedente, dove in più, per comodità si aggiunge il nome del parametro che comunque viene ignorato dal compilatore.

- ```
  void elenca (void);
  ```

  Si tratta della dichiarazione di una funzione che fa qualcosa senza bisogno di ricevere alcun parametro e senza restituire alcun valore (‘**void**’ è il tipo che rappresenta una variabile di rango nullo e, come tale, incapace di accogliere qualunque valore).

66.1.6.2 Descrizione di una funzione

La descrizione della funzione, rispetto alla dichiarazione del prototipo, richiede l’indicazione dei nomi da usare per identificare i parametri (mentre nel prototipo questi sono facoltativi) e naturalmente l’aggiunta delle istruzioni da eseguire. Le parentesi graffe che appaiono nello schema sintattico fanno parte delle istruzioni necessarie.

```
tipo nome ( [tipo parametro [, ...] ] )
{
   istruzione ;
   ...
}
```

Per esempio, la funzione seguente esegue il prodotto tra i due parametri forniti e ne restituisce il risultato:

```
int prodotto (int x, int y)
{
  return (x * y);
}
```

I parametri indicati tra parentesi, rappresentano una dichiarazione di variabili locali[13] che contengono inizialmente i valori usati nella chiamata. Il valore restituito dalla funzione viene definito attraverso
```

l'istruzione '**return**', come si può osservare dall'esempio. Naturalmente, nelle funzioni di tipo '**void**' l'istruzione '**return**' va usata senza specificare il valore da restituire, oppure si può fare a meno del tutto di tale istruzione.

Nei manuali tradizionale del linguaggio C si descrivono le funzioni nel modo visto nell'esempio precedente; al contrario, nella guida *GNU coding standards* si richiede di mettere il nome della funzione in corrispondenza della colonna uno, così:

```
int
prodotto (int x, int y)
{
  return (x * y);
}
```

Le variabili dichiarate all'interno di una funzione, oltre a quelle dichiarate implicitamente come mezzo di trasporto degli argomenti della chiamata, sono visibili solo al suo interno, mentre quelle dichiarate al di fuori di tutte le funzioni, sono variabili globali, accessibili potenzialmente da ogni parte del programma.[14] Se una variabile locale ha un nome coincidente con quello di una variabile globale, allora, all'interno della funzione, quella variabile globale non è accessibile.

Le regole da seguire, almeno in linea di principio, per scrivere programmi chiari e facilmente modificabili, prevedono che si debba fare in modo di rendere le funzioni indipendenti dalle variabili globali, fornendo loro tutte le informazioni necessarie attraverso i parametri. In questo modo diventa del tutto indifferente il fatto che una variabile locale vada a mascherare una variabile globale; inoltre, ciò permette di non dover tenere a mente il ruolo di queste variabili globali e (se

non si usano le variabili «statiche») fa sì che si ottenga una funzione completamente «rientrante».

66.1.6.3 File di intestazione e libreria

Una libreria di funzioni si compone almeno di due parti fondamentali: i prototipi delle funzioni e la descrizione delle funzioni stesse. Secondo la tradizione, l'inclusione di codice attraverso l'istruzione '**`#include`**' del precompilatore, si usa esclusivamente per includere «file di intestazione», contraddistinti convenzionalmente da un nome che termina con il suffisso '**`.h`**'. Questi file di intestazione devono essere costruiti con certi criteri, in modo che la loro inclusione multipla non possa creare problemi. Per quanto riguarda le funzioni, questi file possono contenerne esclusivamente i prototipi.

Per esempio, si potrebbe dire che per poter usare la funzione ***printf()*** si debba includere la «libreria» standard '`stdio.h`'. L'affermazione in sé può essere accettabile, ma non è precisa. Infatti, il file di intestazione '`stdio.h`' contiene prototipi e altre definizioni della porzione della libreria standard che consente di usare la funzione ***printf()***, ma la descrizione effettiva di tale funzione si trova in un altro file.

66.1.7 Vincoli nei nomi

Quando si definiscono variabili e funzioni nel proprio programma, occorre avere la prudenza di non utilizzare nomi che coincidano con quelli delle librerie che si vogliono usare e che non possano andare in conflitto con l'evoluzione del linguaggio. A questo proposito va osservata una regola molto semplice: non si vanno usati nomi «esterni» che inizino con il trattino basso ('_'); in tutti gli altri casi, inve-

ce, non si possono usare i nomi che iniziano con un trattino basso e continuano con una lettera maiuscola o un altro trattino basso.

Il concetto di nome esterno viene descritto a proposito della compilazione di un programma che si sviluppa in più file-oggetto da collegare assieme (sezione 66.3). L'altro vincolo serve a impedire, per esempio, la creazione di nomi come '**_Bool**' o '**__STDC_IEC_559__**'. Rimane quindi la possibilità di usare nomi che inizino con un trattino basso, purché continuino con un carattere minuscolo e siano visibili solo nell'ambito del file sorgente che si compone.

66.1.8 I/O elementare

«

L'input e l'output elementare che si usa nella prima fase di apprendimento del linguaggio C si ottiene attraverso l'uso di due funzioni fondamentali: ***printf()*** e ***scanf()***. La prima si occupa di emettere una stringa dopo averla trasformata in base a dei codici di composizione determinati; la seconda si occupa di ricevere input (generalmente da tastiera) e di trasformarlo secondo codici di conversione simili alla prima. Infatti, il problema che si incontra inizialmente, quando si vogliono emettere informazioni attraverso lo standard output per visualizzarle sullo schermo, sta nella necessità di convertire in qualche modo tutti i dati che non siano già di tipo '**char**'. Dalla parte opposta, quando si inserisce un dato che non sia da intendere come un semplice carattere alfanumerico, serve una conversione adatta nel tipo di dati corretto.

Per utilizzare queste due funzioni, occorre includere il file di intestazione '`stdio.h`', come è già stato visto più volte negli esempi.

Le due funzioni, ***printf()*** e ***scanf()***, hanno in comune il fatto di disporre di una quantità variabile di parametri, dove solo il primo è stato precisato. Per questa ragione, la stringa che costituisce il primo argomento deve contenere tutte le informazioni necessarie a individuare quelli successivi; pertanto, si fa uso di ***specificatori di conversione*** che definiscono il tipo e l'ampiezza dei dati da trattare. A titolo di esempio, lo specificatore '**`%i`**' si riferisce a un valore intero di tipo '**`int`**', mentre '**`%li`**' si riferisce a un intero di tipo '**`long int`**'.

Vengono mostrati solo alcuni esempi, perché una descrizione più approfondita nell'uso delle funzioni ***printf()*** e ***scanf()*** appare in altre sezioni (67.3 e 69.17). Si comincia con l'uso di ***printf()***:

```
…
double capitale = 1000.00;
double tasso    = 0.5;
int    interesse = (capitale * tasso) / 100;
…
printf ("%s: il capitale %f, ", "Ciao", capitale);
printf ("investito al tasso %f%% ", tasso);
printf ("ha prodotto un interesse pari a %i.\n", interesse);
…
```

Gli specificatori di conversione usati in questo esempio si possono considerare quelli più comuni: '**`%s`**' incorpora una stringa; '**`%f`**' traduce in testo un valore che originariamente è di tipo '**`double`**'; '**`%i`**' traduce in testo un valore '**`int`**'; inoltre, '**`%%`**' viene trasformato semplicemente in un carattere percentuale nel testo finale. Alla fine, l'esempio produce l'emissione del testo: «Ciao: il capitale 1000.00, investito al tasso 0.500000% ha prodotto un interesse pari a 5.»

La funzione ***scanf()*** è un po' più difficile da comprendere: la stringa che definisce il procedimento di interpretazione e conversione de-

ve confrontarsi con i dati provenienti dallo standard input. L'uso più semplice di questa funzione prevede l'individuazione di un solo dato:

```
…
int importo;
…
printf ("Inserisci l'importo: ");
scanf ("%i", &importo);
…
```

Il pezzo di codice mostrato emette la frase seguente e resta in attesa dell'inserimento di un valore numerico intero, seguito da [*Invio*]:

```
Inserisci l'importo: _
```

Questo valore viene inserito nella variabile ***importo***. Si deve osservare il fatto che gli argomenti successivi alla stringa di conversione sono dei puntatori, per cui, avendo voluto inserire il dato nella variabile ***importo***, questa è stata indicata preceduta dall'operatore '**&**' in modo da fornire alla funzione l'indirizzo corrispondente (si veda la sezione 66.5 sulla gestione dei puntatori).

Con una stessa funzione ***scanf()*** è possibile inserire dati per diverse variabili, come si può osservare dall'esempio seguente, ma in questo caso, per ogni dato viene richiesta la separazione con spazi orizzontali o anche con la pressione di [*Invio*].

```
printf ("Inserisci il capitale e il tasso:");
scanf ("%i%f", &capitale, &tasso);
```

66.1.9 Restituzione di un valore

In un sistema Unix e in tutti i sistemi che si rifanno a quel modello, i programmi, di qualunque tipo siano, al termine della loro esecuzione, restituiscono un valore che può essere utilizzato da uno script di shell per determinare se il programma ha fatto ciò che si voleva o se è intervenuto qualche tipo di evento che lo ha impedito.

Convenzionalmente si tratta di un valore numerico, con un intervallo di valori abbastanza ristretto, in cui zero rappresenta una conclusione normale, ovvero priva di eventi indesiderati, mentre qualsiasi altro valore rappresenta un'anomalia. A questo proposito si consideri quello «strano» atteggiamento degli script di shell, per cui zero equivale a *Vero*.

Lo standard del linguaggio C prescrive che la funzione ***main()*** debba restituire un tipo intero, contenente un valore compatibile con l'intervallo accettato dal sistema operativo: tale valore intero è ciò che dovrebbe lasciare di sé il programma, al termine del proprio funzionamento.

Se il programma deve terminare, per qualunque ragione, in una funzione diversa da ***main()***, non potendo usare l'istruzione '**`return`**' per questo scopo, si può richiamare la funzione ***exit()***:

```
exit (valore_restituito);
```

La funzione ***exit()*** provoca la conclusione del programma, dopo aver provveduto a scaricare i flussi di dati e a chiudere i file. Per questo motivo, non restituisce un valore all'interno del programma, al contrario, fa in modo che il programma restituisca il valore indicato

come argomento.

Per poterla utilizzare occorre includere il file di intestazione '`stdlib.h`' che tra l'altro dichiara già due macro-variabili adatte a definire la conclusione corretta o errata del programma: ***EXIT_SUCCESS*** e ***EXIT_FAILURE***.[15] L'esempio seguente mostra in che modo queste macro-variabili potrebbero essere usate:

```
#include <stdlib.h>
…
…
if (…)
  {
    exit (EXIT_SUCCESS);
  }
else
  {
    exit (EXIT_FAILURE);
  }
```

Naturalmente, se si può concludere il programma nella funzione ***main()***, si può fare lo stesso con l'istruzione '**`return`**':

```
#include <stdlib.h>
…
…
int main (…)
{
    …
    if (…)
      {
        return (EXIT_SUCCESS);
      }
    else
      {
```

```
        return (EXIT_FAILURE);
      }
    ...
}
```

66.1.10 Attributi per GNU C

Il compilatore GNU C prevede l'uso di «attributi» nel proprio codice, come estensione del linguaggio. Dal momento che il compilatore GNU C è molto importante e diffuso, conviene sapere che forma possono avere tali attributi, almeno per non restare sbalorditi nella lettura del codice di altri autori:

```
__attribute__ ((tipo_di_attributo))
```

Frequentemente, questi attributi vanno collocati alla fine della dichiarazione di ciò a cui si riferiscono, come nell'esempio seguente, dove viene assegnato l'attributo '**deprecated**' al prototipo di una funzione:

```
...
mia_funzione (void) __attribute__ ((deprecated));
...
```

Se può servire, il nome dell'attributo può apparire anche preceduto e terminato da due trattini bassi; pertanto, l'esempio già visto può essere scritto anche così:

```
...
mia_funzione (void) __attribute__ ((__deprecated__));
...
```

Il fatto che siano previsti tali attributi dal compilatore GNU C, rende molto difficile l'individuazione di un errore frequente e banale: la mancanza del punto e virgola alla fine di un prototipo di funzione. Per esempio, si può supporre di avere realizzato un proprio file di intestazione con il contenuto seguente:

```
…
mia_funzione_1 (int a, int b);
mia_funzione_2 (int a, int b)
mia_funzione_3 (int a, int b);
mia_funzione_4 (int a, int b);
…
```

Come si vede, il prototipo di ***mia_funzione_2()*** non è concluso con il punto e virgola. Durante la compilazione di un file che include questa porzione di codice, l'errore che viene evidenziato dal compilatore GNU C è incomprensibile, rispetto alla realtà effettiva:

```
In file included from …/lib/stdio.h:5,
                 from asctime.c:3:
…/lib/stdarg.h: In function 'asctime':
…/lib/stdarg.h:4: error: storage class specified for
  parameter 'va_list'
In file included from …/lib/stdio.h:10,
                 from asctime.c:3:
…/lib/sys/types.h:8: error: storage class specified for
  parameter 'blkcnt_t'
…/lib/sys/types.h:9: error: storage class specified for
  parameter 'blksize_t'
…/lib/sys/types.h:10: error: storage class specified for
  parameter 'dev_t'
…
…
…/lib/stdio.h:94: error: expected declaration specifiers
```

```
  or '...' before 'va_list'
…/lib/stdio.h:96: error: expected declaration specifiers
  or '...' before 'va_list'
asctime.c:7: error: expected '=', ',', ';', 'asm' or
  '__attribute__' before '{' token
asctime.c:99: error: old-style parameter declarations in
  prototyped function definition
asctime.c:99: error: expected '{' at end of input
make: *** [asctime] Error 1
…
```

L'esempio mostrato si riferisce a un errore provocato volutamente nel file di intestazione '`time.h`', a cui mai viene fatto riferimento nell'analisi del compilatore. Pertanto, di fronte a errori così incomprensibili, è determinante il controllo della conclusione corretta dei prototipi delle funzioni, all'interno dei file di intestazione prodotti per proprio conto.

66.2 Istruzioni del precompilatore

Il linguaggio C non può fare a meno del precompilatore e le sue direttive sono regolate dallo standard. Il precompilatore è un programma, o quella parte del compilatore, che si occupa di pre-elaborare un sorgente per generarne uno nuovo, il quale poi viene compilato con tutte le trasformazioni apportate.

Tradizionalmente, in un sistema operativo che si rifà al modello dei sistemi Unix, il precompilatore è costituito dal programma '**`cpp`**' che può essere utilizzato direttamente o in modo trasparente dal compilatore '**`cc`**'. Volendo simulare i passaggi iniziali della compilazione di un programma ipotetico denominato '`prg.c`', evidenziando il ruolo del precompilatore, questi si potrebbero esprimere così:

```
$ cpp -E -o prg.i prg.c[Invio]

$ cc -o prg.o prg.i[Invio]

$ ...
```

In questo caso, il file '`prg.i`' generato dal precompilatore è quello che viene chiamato dalla documentazione standard una ***unità di traduzione***. Una unità di traduzione singola può essere il risultato della fusione di diversi file, incorporati attraverso le direttive '**`#include`**', come viene descritto nel capitolo. Ciò che occorre osservare è che, quando si parla di campo di azione legato al «file», ci si riferisce al file generato dal precompilatore, ovvero all'unità di traduzione.

Va osservato che esistono programmi che utilizzano il precompilatore del linguaggio C per fini estranei al linguaggio stesso. Per esempio i file di configurazione delle risorse di X (il sistema grafico) vengono fatti elaborare da '**`cpp`**' prima di essere interpretati.

66.2.1 Linguaggio a sé stante

«

Le direttive del precompilatore rappresentano un linguaggio a sé stante, con proprie regole. In generale:

- le direttive iniziano con il simbolo '**`#`**', preferibilmente nella prima colonna;
- le direttive non utilizzano alcun simbolo di conclusione (non si usa il punto e virgola);
- ogni direttiva occupa una riga, la quale può essere spezzata e ripresa in righe successive, utilizzando il simbolo '**`\`**' subito prima del codice di interruzione di riga;

- su una riga può essere inserita una sola direttiva, perché non c'è altro modo di distinguere la fine di una dall'inizio della successiva.

Se appare un simbolo '**#**' privo di altre indicazioni, questo viene semplicemente ignorato dal precompilatore. Di solito le direttive del precompilatore si scrivono senza annidamenti, ma questo fatto rischia di rendere particolarmente complicata la lettura del sorgente. A ogni modo, se si usano gli annidamenti, di solito questi riguardano solo le altre direttive e non il codice del linguaggio C puro e semplice.

I commenti del linguaggio C possono apparire solo alla fine delle direttive, ma non in tutte; pertanto vanno usati con prudenza. Vengono usati sicuramente alla fine delle direttive '**#else**' e '**#endif**' per ricordare a quale condizione si riferiscono.

66.2.2 Direttiva «#include»

La direttiva '**#include**' permette di includere un file. Generalmente si tratta di un cosiddetto ***file di intestazione***, contenente una serie di definizioni necessarie al file sorgente in cui vengono incorporate. Il file da incorporare può essere indicato delimitandolo con le parentesi angolari, oppure con gli apici doppi; il modo in cui si delimita il nome del file serve a stabilire come questo deve essere cercato:[16]

```
#include <file>
```

```
#include "file"
```

I due esempi seguenti mostrano la richiesta di includere il file ‘`stdio.h`’ secondo le due forme possibili:

```
#include <stdio.h>
```

```
#include "stdio.h"
```

Delimitando il nome tra parentesi angolari si fa riferimento a un file che dovrebbe trovarsi in una posizione stabilita dalla configurazione del compilatore; per esempio, nel caso di GNU C in un sistema GNU/Linux, dovrebbe trattarsi della directory ‘`/usr/include/`’. Se invece si delimita il nome tra apici doppi, generalmente si fa riferimento a una posizione precisa nel file system, attraverso l’indicazione di un percorso (secondo la modalità prevista dal sistema operativo); pertanto, scrivendo il nome del file come nell’esempio, si dovrebbe intendere che la sua collocazione debba essere la directory corrente.

Di norma, quando si indica un file da includere delimitandolo con gli apici doppi e senza indicare alcun percorso, se questo file non si trova nella directory corrente, allora viene cercato nella directory predefinita, come se fosse stato indicato tra le parentesi angolari.

Un file incorporato attraverso la direttiva ‘**`#include`**’, può a sua volta fare lo stesso con altri; naturalmente, questa possibilità va considerata per evitare di includere più volte lo stesso file e di solito si usa un accorgimento che viene descritto più avanti nel capitolo.

66.2.3 Direttiva «#define»

La direttiva '**#define**' serve a definire quelle che sono note come ***macro***, ovvero delle variabili del precompilatore che, successivamente, il precompilatore stesso espande secondo regole determinate. Lo standard del linguaggio C distingue queste macro in due categorie: *object-like macro* e *function-like macro*. Nel corso di questi capitoli si usa la definizione di ***macro-variabile*** nel primo caso e di ***macroistruzione*** nel secondo.

Come sottoinsieme delle macro-variabili vengono considerate le ***costanti manifeste***, per rappresentare dei valori semplici che si ripetono nel sorgente. Per esempio, ***NULL*** è la costante manifesta standard per rappresentare il puntatore nullo.

```
#define macro [sequenza_di_caratteri]
```

La direttiva '**#define**' usata secondo la sintassi mostrata consente di definire delle macro-variabili, ovvero ciò che lo standard definisce *object-like macro*. Ciò che si ottiene è la sostituzione nel sorgente del nome indicato con la sequenza di caratteri che lo segue. Si osservi l'esempio seguente:

```
#define SALUTO Ciao! Come stai?
```

In questo caso viene dichiarata la macro-variabile ***SALUTO*** in modo tale che tutte le occorrenze di questo nome, successive alla sua dichiarazione, vengano sostituite con '**Ciao! Come stai?**'. È molto importante comprendere questo particolare: tutto ciò che appare dopo il nome della macro, a parte lo spazio che lo separa, viene utilizzato nella sostituzione. L'esempio seguente, invece rappresenta un

programma completo.

Listato 66.68. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/8xkVUB59* , *http://ideone.com/HSVI2* .

```
#include <stdio.h>
#define SALUTO "Ciao! come stai?\n"
int main (void)
{
  printf (SALUTO);
  return 0;
}
```

In questo caso, la macro-variabile ***SALUTO*** può essere utilizzata in un contesto in cui ci si attende una stringa letterale, perché include gli apici doppi che sono necessari per questo scopo. Nell'esempio si vede l'uso della macro-variabile come argomento della funzione ***printf()*** e l'effetto del programma è quello di mostrare il messaggio seguente:

```
Ciao! come stai?
```

È bene precisare che la sostituzione delle macro-variabili non avviene se i loro nomi appaiono tra apici doppi, ovvero all'interno di stringhe letterali. Si osservi l'esempio seguente.

Listato 66.70. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/qfPSZZm0* , *http://ideone.com/CAAk3* .

```
#include <stdio.h>
#define SALUTO Ciao! come stai?
int main (void)
{
  printf ("SALUTO\n");
  return 0;
}
```

In questo caso, la funzione ***printf()*** emette effettivamente la parola '**SALUTO**' e non avviene alcuna espansione di macro:

```
SALUTO
```

Una volta compreso il meccanismo basilare della direttiva '**#define**' si può osservare che questa può essere utilizzata in modo più complesso, facendo anche riferimento ad altre macro già definite:

```
#define UNO 1
#define DUE UNO+UNO
#define TRE DUE+UNO
```

In presenza di una situazione come questa, utilizzando la macro ***TRE***, si ottiene prima la sostituzione con '**DUE+UNO**', quindi con '**UNO+UNO+1**', infine con '**1+1+1**' (dopo, tocca al compilatore).

Tradizionalmente i nomi delle macro-variabili vengono definiti utilizzando solo lettere maiuscole, in modo da poterli distinguere facilmente nel sorgente.

Come è possibile vedere meglio in seguito, è sensato anche dichiarare una macro senza alcuna corrispondenza. Ciò può servire per le direttive '**#ifdef**' e '**#ifndef**'.

Nella definizione di una macro-variabile può apparire l'operatore '**##**', con lo scopo di attaccare ciò che si trova alle sue estremità. Si osservi l'esempio seguente.

Listato 66.73. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/XQ6Ns1AT* , *http://ideone.com/R5G3o* .

```
#include <stdio.h>
#define UNITO 1234 ## 5678
int main (void)
{
  printf ("%i\n", UNITO);
  return 0;
}
```

Eseguendo questo programma si ottiene semplicemente l'emissione del numero 12345678. Questo operatore può servire anche per unire assieme il nome di una macro-variabile, benché questo sia poco consigliabile.

Listato 66.74. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/lZ0QKzln* , *http://ideone.com/qOqC1* .

```
#include <stdio.h>
#define MIAMACRO 12345678
#define UNITO MI ## A ## MA ## CRO
int main (void)
{
  printf ("%i\n", UNITO);
  return 0;
}
```

66.2.4 Direttiva «#define» con parametri

La direttiva '**#define**' può essere usata per creare una macroistruzione, ovvero una cosa che viene usata con l'apparenza di una funzione:

```
#define macro(parametro [, parametro]...) sequenza_di_caratteri
```

Per comprendere il meccanismo è meglio avvalersi di esempi. In quello seguente, l'istruzione '**i = QUADRATO(i)**' si traduce in '**i = (i)*(i)**':

```
#define QUADRATO(A)         (A)*(A)
...
...
i = QUADRATO (i);
...
```

Si osservi il fatto che, nella definizione, la stringa di sostituzione è stata composta utilizzando le parentesi: ciò permette di evitare problemi successivamente, nelle precedenze di valutazione del-

le espressioni, se l'argomento della funzione simulata attraverso la macroistruzione è composto:

```
…
i = QUADRATO (123 * 34 + 3);
…
```

In questo caso, la sostituzione genera '**`i = (123 * 34 + 3)*(123 * 34 + 3)`**' e si può vedere che le parentesi sono appropriate. L'esempio seguente, costituito da un programma completo, mostra l'uso di due parametri.

Listato 66.77. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/MAxBs8MQ* , *http://ideone.com/ML1LU* .

```
#include <stdio.h>
#define MAX(X, Y) ((X) > (Y) ? (X) : (Y))
int main (void)
{
  printf ("valore massimo tra %i e %i: %i\n",
           3, 4, MAX (3, 4));
  return 0;
}
```

La macroistruzione '**`MAX (3, 4)`**' si traduce in '**`((3) > (4) ? (3) : (4))`**'.

È molto importante fare attenzione alla spaziatura nella dichiarazione di una macroistruzione: si può scrivere '**#define MAX(x,y)** ...', '**#define MAX(x,y)** ...', '**#define MAX(x,y)** ...', '**#define MAX(x, y)** ...', ecc. Quello che invece non si può proprio è l'inserimento di uno spazio tra il nome della macroistruzione e la parentesi tonda aperta. Pertanto, se si scrive '**#define MAX (x, y)** ...' si commette un errore!
Al contrario, quando la macroistruzione viene richiamata, questo spazio può essere inserito senza problemi, come apparso già negli esempi.

Nella definizione di una macroistruzione può essere usato l'operatore '**##**' già descritto nella sezione precedente. Nell'esempio seguente si ottiene di visualizzare il numero 12345678.

Listato 66.78. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/GHXfnaHL* , *http://ideone.com/WRD5n* .

```
#include <stdio.h>
#define UNISCI(A, B) A ## B
int main (void)
{
  printf ("%i\n", UNISCI(1234, 5678));
  return 0;
}
```

Inoltre, è disponibile l'operatore '**#**' che ha lo scopo di racchiudere tra apici doppi la metavariabile che lo segue immediatamente. Si osservi l'esempio seguente:

Listato 66.79. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/1nEG1ryz* , *http://ideone.com/czt2V* .

```
#include <stdio.h>
#define STRINGATO(a) # a
#define SALUTO STRINGATO (Ciao! come stai?\n)
int main (void)
{
  printf (SALUTO);
  return 0;
}
```

Prima viene definita la macroistruzione ***STRINGATO***, con la quale si vuole che il suo argomento sia raccolto tra apici doppi. Subito dopo viene definita la macro-variabile ***SALUTO*** che viene rimpiazzata da '**`STRINGATO (Ciao! come stai?\n)`**' e quindi da '**`"Ciao! come stai?\n"`**'. Alla fine, il programma mostra regolarmente il messaggio già visto in un altro esempio precedente:

```
Ciao! come stai?
```

Si osservi cosa accadrebbe modificando l'esempio nel modo seguente, dove si vuole che la macroistruzione ***STRINGATO*** utilizzi due parametri:

```
…
#define STRINGATO(a, b) # a # b
#define SALUTO STRINGATO (Ciao!, come stai?\n)
…
```

Evidentemente si vuole che i due argomenti forniti alla macroistruzione ***STRINGATO*** siano raccolti ognuno tra apici doppi, pertanto la macro-variabile si trova a essere dichiarata, sostanzialmente come '**`"Ciao!" "come stai?\n"`**'. Alla fine il risultato mostrato dal

programma è differente, perché la sequenza delle due stringhe viene intesa come una sequenza sola, ma in tal caso manca lo spazio tra le due parti:

```
Ciao!come stai?
```

Si può complicare ulteriormente l'esempio per dimostrare fino a dove si estende la competenza dell'operatore '**#**', come si vede nel listato successivo.

Listato 66.83. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/idGYtl78* , *http://ideone.com/17IQ17c* .

```
#include <stdio.h>
#define STRINGATO(a, b) # a , b
#define SALUTO STRINGATO (%i un amore\n, 6)
int main (void)
{
  printf (SALUTO);
  return 0;
}
```

Qui gli spazi sono importanti, infatti, la macroistruzione ***STRINGATO*** si traduce in '**"*a*" , *b***' e la virgola non avrebbe potuto essere unita alla lettera «a», altrimenti sarebbe stata inserita dentro la coppia di apici doppi. La macro-variabile ***SALUTO*** si traduce poi in '**"%i un amore\n" , 6**', pertanto, alla fine, il programma mostra il messaggio seguente:

```
6 un amore
```

Per concludere viene mostrato un esempio ulteriore, con il quale si crea una sorta di funzione che il precompilatore deve trasformare in un blocco di istruzioni. Viene simulato il comportamento della

funzione standard ***strncpy()***, senza però restituire un valore.

Listato 66.85. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/eCbQbPAR* , *http://ideone.com/A2Q2L* .

```
#include <stdio.h>

#define STRNCPY(DST, ORG, N) { \
    char *s1 = (DST); \
    const char *s2 = (ORG); \
    size_t n = (N); \
    int i; \
    for (i = 0; i < n && s2[i] != 0; i++) \
      { \
        s1[i] = s2[i]; \
      } \
    s1[i] = 0; }

int main (void)
{
  char stringa[100];
  STRNCPY (stringa, "Buon giorno a tutti!", 50)    // [1]
                              // [1] Si osservi che manca il
                              //     punto e virgola finale!
  printf ("%s\n", stringa);
  return 0;
}
```

Si può vedere che, per richiamare questa macroistruzione, non si richiede che le sia aggiunto il punto e virgola. Infatti, la macro in sé si espande in un raggruppamento tra parentesi graffe, che non ne ha bisogno; d'altra parte, volendoglielo aggiungere, non si può creare alcun problema.

La dichiarazione di una macroistruzione può prevedere una quantità variabile di parametri, come avviene già per le funzioni (sezione 66.6.3). Per ottenere questo si aggiungono dei puntini di sospensione alla fine dell'elenco dei parametri fissi, quindi, si utilizza la parola chiave '**__VA_ARGS__**' per individuare gli argomenti opzionali. L'esempio seguente riproduce il funzionamento di ***printf()***, richiamando la stessa funzione.

Listato 66.86. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/j98SG985Wo* , *http://ideone.com/cOKlA* .

```
#include <stdio.h>

#define PRINTF(A, ...) printf (A, __VA_ARGS__)

int main (void)
{
  PRINTF ("I primi numeri interi: %i, %i, %i\n", 1, 2, 3);
  return 0;
}
```

Questa volta il punto e virgola finale serve, perché non è stato incluso nella definizione della macroistruzione.

A proposito di '**__VA_ARGS__**' va ancora osservato che individua sì gli argomenti opzionali, ma di questi ne deve essere specificato almeno uno. Pertanto, la macroistruzione ***PRINTF()***, per come è stata dichiarata nell'esempio precedente, va usata sempre con almeno due argomenti. In questo caso, per poter usare la macroistruzione con un argomento solo, la sua definizione va modificata nel modo seguente:

```
…
#define PRINTF(...) printf (__VA_ARGS__)
…
```

66.2.5 Direttive «#if», «#else», «#elif» e «#endif»

«

Le direttive '**#if**', '**#else**', '**#elif**' e '**#endif**', permettono di delimitare una porzione di codice che debba essere utilizzato o ignorato in relazione a una certa espressione che può essere calcolata solo attraverso definizioni precedenti.

```
#if espressione
    espressione
[#elif espressione
     espressione]
…
[#else
     espressione]
#endif
```

Le espressioni che rappresentano le condizioni da valutare seguono regole equivalenti a quelle del linguaggio, tenendo conto che se si vogliono usare delle variabili, queste possono solo essere quelle del precompilatore. L'esempio seguente mostra la dichiarazione di una macro-variabile a cui si associa un numero, quindi si vede un confronto basato sul valore in cui si espande la macro-variabile stessa:

```
#define DIM_MAX 1000
…
…
```

```
int main (void)
{
…
#if DIM_MAX>100
  printf ("Dimensione enorme.\n");
  …
#else
  printf ("Dimensione normale.\n");
  …
#endif
…
}
```

L'esempio mostra il confronto tra la macro-variabile ***DIM_MAX*** e il valore 100. Essendo stata dichiarata per tradursi in 1000, il confronto è equivalente a 1000 > 100 che risulta vero, pertanto il compilatore include solo le istruzioni relative.

Gli operatori di confronto che si possono utilizzare per le espressioni logiche sono i soliti, in particolare, è bene ricordare che per valutare l'uguaglianza si usa l'operatore '**==**', come nell'esempio successivo:

```
#define NAZIONE ita
…
…
int main (void)
{
#if NAZIONE==ita
  char valuta[] = "EUR";
  …
#elif NAZIONE==usa
  char valuta[] = "USD";
  …
#endif
```

```
…
}
```

Queste direttive condizionali possono essere annidate; inoltre possono contenere anche altri tipi di direttiva del precompilatore.

66.2.6 Direttive «#if defined», «#if !defined», «#ifdef» e «#ifndef»

«

Nelle espressioni che esprimono una condizione per la direttiva '**`#if`**' è possibile usare l'operatore '**`defined`**', seguito dal nome di una macro-variabile. La condizione '**`defined`** ***macro***' si avvera se la macro indicata risulta definita, anche se dovesse essere priva di valore. Per converso, la condizione '**`!defined`** ***macro***' si avvera quando la macro non risulta definita.

La direttiva '**`#if defined`**' può essere abbreviata come '**`#ifdef`**', mentre '**`#if !defined`**' si può esprimere come '**`#ifndef`**'.

```
#define DEBUG
…
int main (void)
{
…
#if defined DEBUG
  printf ("Punto di controllo n. 1\n");
  …
#endif // DEBUG
…
}
```

```
#define DEBUG
…
int main (void)
{
…
#ifdef DEBUG
  printf ("Punto di controllo n. 1\n");
  …
#endif // DEBUG
…
}
```

I due esempi equivalenti mostrano il caso in cui sia dichiarata una macro ***DEBUG*** (che non si traduce in alcunché) e in base alla sua esistenza viene incluso il codice che mostra un messaggio particolare.

```
#define OK
…
int main (void)
{
#if !defined OK
  printf ("Punto di controllo n. 1\n");
  …
#endif // OK
…
}
```

```
#define OK
…
int main (void)
{
#ifndef OK
  printf ("Punto di controllo n. 1\n");
  …
#endif // OK
…
}
```

Questi due esempi ulteriori sono analoghi a quanto già mostrato, con la differenza che le istruzioni controllate vengono incluse nella compilazione solo se la macro indicata non è stata dichiarata.

Quando si scrivono delle condizioni basate sull'esistenza o meno di una macro, può essere utile aggiungere alla conclusione un commento con cui si ricorda a quale macro si sta facendo riferimento, in modo da districarsi più facilmente in presenza di più livelli di annidamento. Ma occorre fare molta attenzione, perché se si commettono errori con questi commenti il compilatore non può dare alcuna segnalazione in merito e si rende incomprensibile il sorgente alla rilettura successiva.

Esiste una situazione ricorrente in cui viene utilizzata la direttiva '**`#if !defined`**' o '**`#ifndef`**' che è bene conoscere. Spesso i file di intestazione che vengono inclusi con direttive '**`#include`**' includono a loro volta tutto quello che serve loro, ma così facendo c'è la possibilità che lo stesso file venga incluso più volte. Per evitare di prendere in considerazione una seconda volta lo stesso file, si usa

un artificio molto semplice, come si vede nel listato successivo che riproduce il contenuto del file ‘`stdbool.h`’ di una libreria standard ipotetica:

```
#ifndef _STDBOOL_H
#define _STDBOOL_H      1

#define bool    _Bool
#define true    1
#define false   0
#define __bool_true_false_are_defined   1

#endif // _STDBOOL_H
```

Come si vede, se il codice viene eseguito per la prima volta, la condizione ‘**`ifndef _STDBOOL_H`**’ non si avvera e di conseguenza la macro-variabile ***_STDBOOL_H*** viene creata effettivamente e quindi viene considerato tutto il resto del codice fino alla direttiva ‘**`#endif`**’. Ma quando si tenta di eseguire lo stesso codice per la seconda volta, o per altre volte successive, dato che la macro-variabile ***_STDBOOL_H*** risulta già definita, questo codice viene ignorato semplicemente, senza altre conseguenze.

Le direttive che consentono di compilare selettivamente solo una porzione del codice, consentono di realizzare del codice molto sofisticato, ma rischiano di renderlo estremamente complesso da interpretare attraverso la lettura umana. Pertanto, è bene limitarne l’uso alle situazioni che sono utili effettivamente.

66.2.7 Direttiva «#undef»

La direttiva '**#undef**' permette di eliminare una macro a un certo punto del sorgente:

```
#undef macro
```

Si mostra un esempio molto semplice, nel quale prima si dichiara la macro-variabile ***NAZIONE***, poi, quando non serve più, questa viene eliminata.

```
#define NAZIONE ita
…
/* In questa posizione, NAZIONE risulta definita */
…
#undef NAZIONE
…
/* In questa posizione, NAZIONE non è definita */
…
```

66.2.8 Direttiva «#line»

Di norma, il compilatore abbastanza evoluto consente di inserire nel file eseguibile delle informazioni che consentano di abbinare il codice eseguibile alle righe del file sorgente originale. Per esempio, con GNU C si può usare l'opzione '**-gstabs**' e altre simili. Naturalmente, in condizioni normali il compilatore conta da solo le righe e annota il nome del file sorgente originale.

Con la direttiva '**#line**' è possibile istruire il compilatore in modo che tenga in considerazione un numero di riga differente, ma soprattutto consente di specificare a quale file sorgente ci si vuole riferire.

```
#line n_riga ["nome_file_sorgente"]
```

C'è da osservare che, per il programmatore, è poco probabile che sia necessario indicare una riga diversa nello stesso sorgente. In effetti, diventa più utile se si abbina il nome di un altro file. Per comprendere come possa essere utilizzata questa possibilità, occorre ipotizzare la costruzione di un altro compilatore per un linguaggio nuovo, con il quale si genera codice in linguaggio C. A titolo di esempio si suppone di volere tradurre il file '`hanoi.pseudo`' che si vede nel listato 66.96 in un sorgente C, denominato '`hanoi.c`', mantenendo il riferimento alle righe originali.

Listato 66.96. Il file '`hanoi.pseudo`'.

```
1   HANOI (N, P1, P2)
2       IF N > 0
3           THEN
4               HANOI (N-1, P1, 6-P1-P2)
5               scrivi: "Muovi l'anello" N "dal piolo" P1 "al piolo" P2
6               HANOI (N-1, 6-P1-P2, P2)
7       END IF
8   END HANOI
9
10  MAIN ()
11      HANOI (3, 1, 2)
12  END MAIN
```

Per ottenere il risultato atteso, il file '`hanoi.c`' deve contenere diverse direttive '**`#line`**', come si vede nel listato 66.97, anche se alcune di quelle potrebbero essere omesse, contando sull'incremento automatico da parte del compilatore.

Listato 66.97. Il file ‘`hanoi.c`’.

```
#include <stdio.h>

#line 1 "hanoi.pseudo"
void hanoi (int N, int P1, int P2)
{
  #line 2 "hanoi.pseudo"
  if (N > 0)
    {
      #line 4 "hanoi.pseudo"
      hanoi (N-1, P1, 6-P1-P2);
      #line 5 "hanoi.pseudo"
      printf ("Muovi l'anello %i dal piolo %i al piolo %i\n", N, P1, P2);
      #line 6 "hanoi.pseudo"
      hanoi (N-1, 6-P1-P2, P2);
      #line 7 "hanoi.pseudo"
    }
  #line 8 "hanoi.pseudo"
}

#line 10 "hanoi.pseudo"
int main (void)
{
  #line 11 "hanoi.pseudo"
  hanoi (3, 1, 2);
  #line 12 "hanoi.pseudo"
  return 0;
  #line 12 "hanoi.pseudo"
}
```

La compilazione del file ‘`hanoi.c`’ potrebbe avvenire nel modo seguente:

```
$ cc -Wall -gstabs hanoi.c
```

Si dovrebbe ottenere il file eseguibile ‘**`a.out`**’ e si verifica sommariamente se funziona:

```
$ ./a.out
```

```
Muovi l'anello 1 dal piolo 1 al piolo 2
Muovi l'anello 2 dal piolo 1 al piolo 3
Muovi l'anello 1 dal piolo 2 al piolo 3
Muovi l'anello 3 dal piolo 1 al piolo 2
Muovi l'anello 1 dal piolo 3 al piolo 1
Muovi l'anello 2 dal piolo 3 al piolo 2
Muovi l'anello 1 dal piolo 1 al piolo 2
```

Il risultato è quello previsto. Se lo si esegue con l'ausilio di programmi come GDB, si può osservare che il riferimento al sorgente originale è quello del file '`hanoi.pseudo`':

```
$ gdb a.out

(gdb) break main[Invio]

 Breakpoint 1 at 0x80483d8: file hanoi.pseudo, line 11.

(gdb) run[Invio]

 Starting program: /home/tizio/a.out
 …
 Breakpoint 1, main () at hanoi.pseudo:11
 11          HANOI (3, 1, 2)

(gdb) stepi[Invio]

 0x080483e0      11          HANOI (3, 1, 2)

(gdb) stepi[Invio]

 0x080483e8      11          HANOI (3, 1, 2)

(gdb) stepi[Invio]

 0x080483ef      11          HANOI (3, 1, 2)
```

```
(gdb) stepi[Invio]

 hanoi (n=3, p1=1, p2=2) at hanoi.pseudo:2
 2           IF N > 0

(gdb) stepi[Invio]

 0x08048355      2           IF N > 0

(gdb) stepi[Invio]

 0x08048357      2           IF N > 0

(gdb) stepi[Invio]

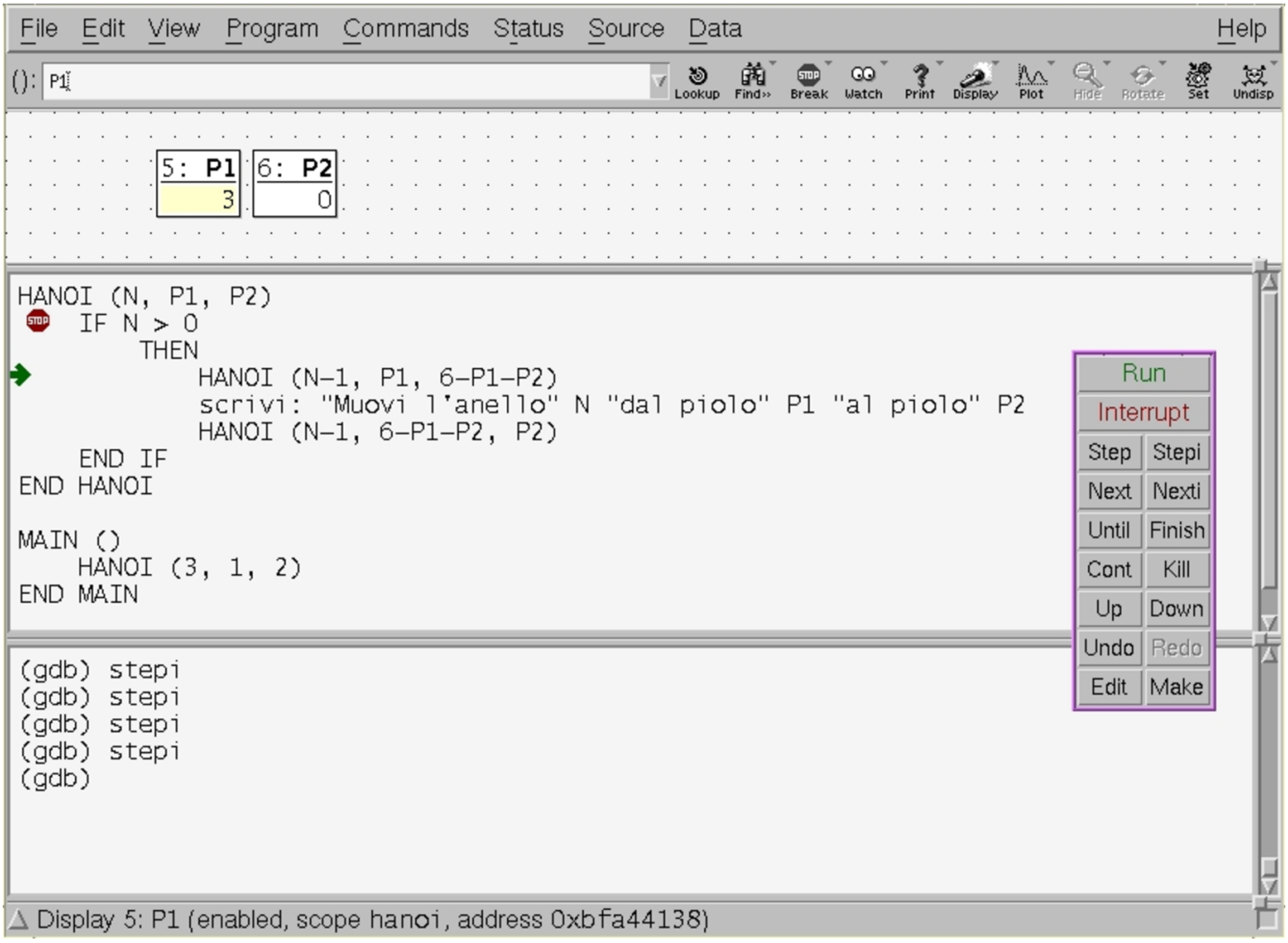
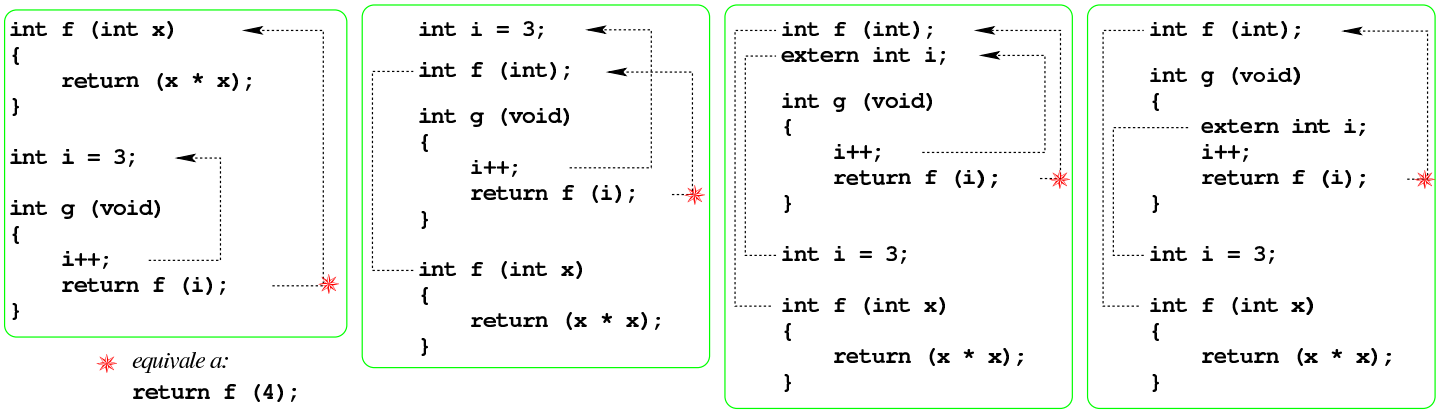
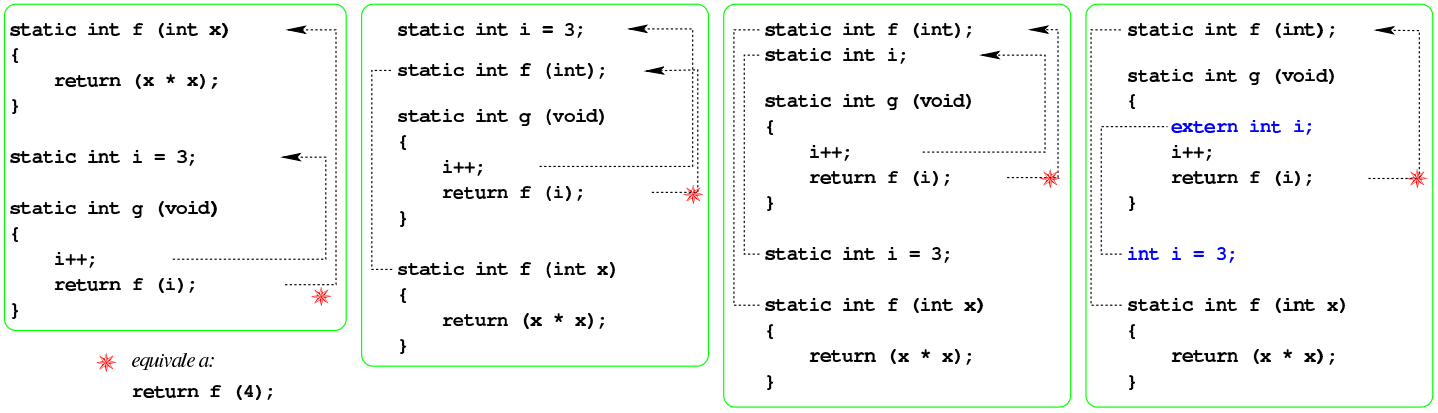
 2           IF N > 0
 (gdb) stepi
 0x0804835e      2           IF N > 0

(gdb) stepi[Invio]

 4                   HANOI (N-1, P1, 6-P1-P2)

(gdb) stepi[Invio]

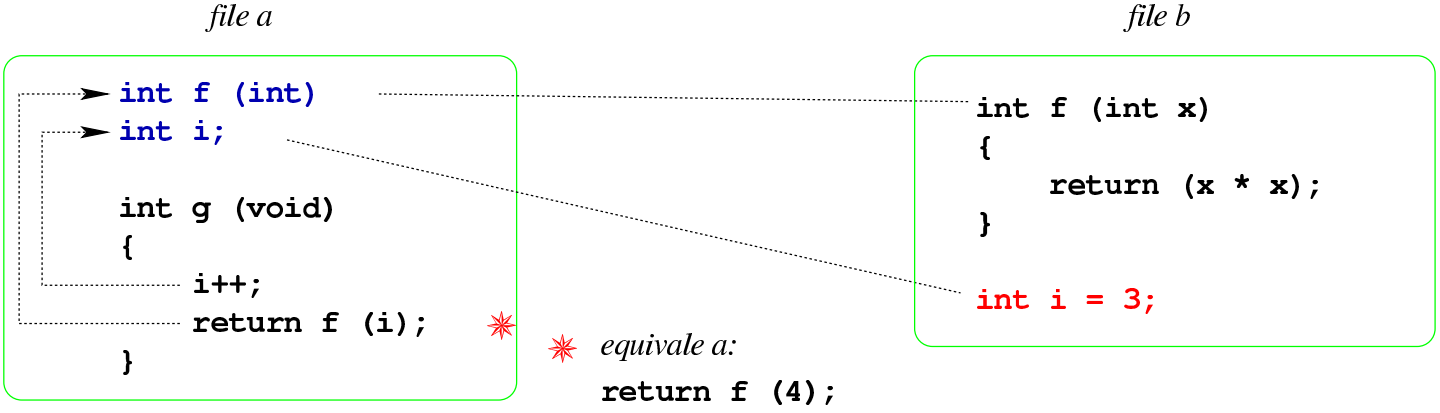
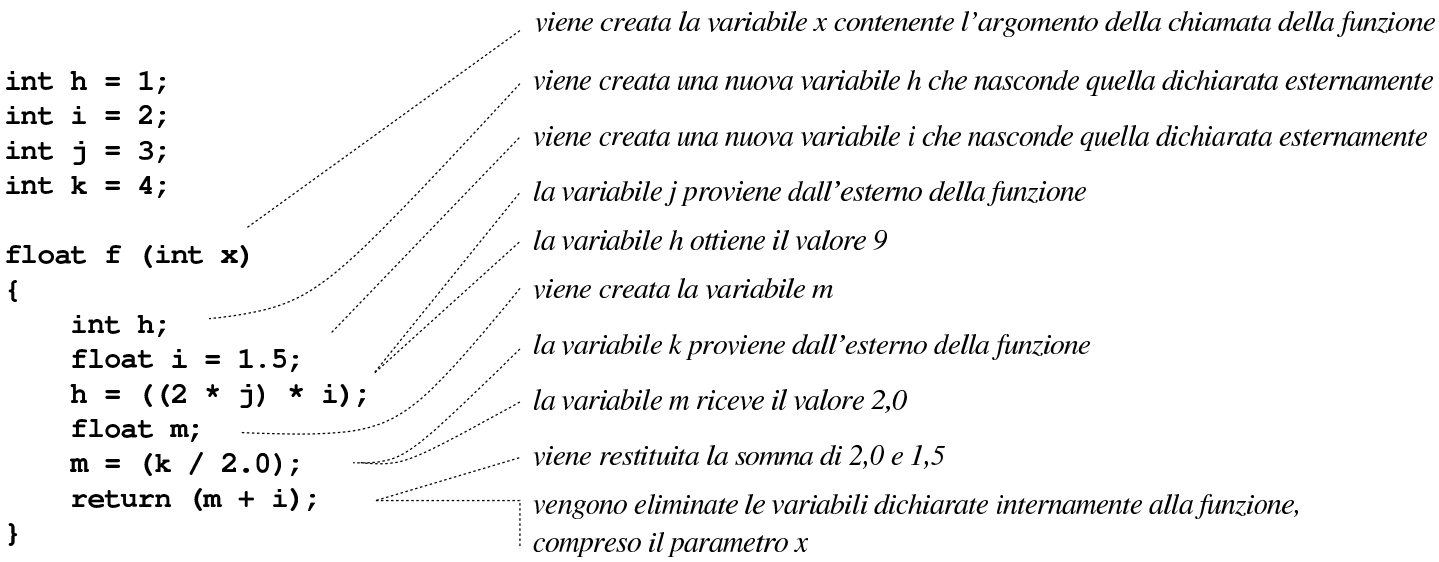
 0x08048363      4                   HANOI (N-1, P1, 6-P1-P2)

(gdb) quit[Invio]
```

Figura 66.110. Esecuzione controllata del programma attraverso DDD.



66.2.9 Direttiva «#error»

La direttiva '**`#error`**' serve a generare un messaggio diagnostico in fase di compilazione, normalmente con lo scopo di interrompere lì il procedimento. In pratica è un modo per interrompere la compilazione già in fase di elaborazione da parte del precompilatore, al verificarsi di certe condizioni.

```
#error messaggio
```

Il messaggio viene trattato in modo letterale, senza l'espansione delle macro.

```
#if ! __STDC_IEC_559__
#error compilatore non conforme alle specifiche IEC 60599!
#endif
```

L'esempio mostra una situazione verosimile per l'utilizzo della direttiva '**`#error`**', dove si controlla che il valore in cui si espande la macro-variabile ***__STDC_IEC_559__*** sia diverso da zero, ma se non è così viene visualizzato il messaggio di errore e la compilazione dovrebbe venire interrotta.

66.2.10 Macro predefinite

«

Lo standard del C prevede che il compilatore disponga di alcune macro-variabili predefinite, elencate sinteticamente nella tabella successiva.

Tabella 66.112. Macro-variabili predefinite secondo lo standard.

Macro-variabile	Descrizione
`__DATE__` `__TIME__`	La data e l'ora della compilazione sono accessibili attraverso le macro-variabili ***__DATE__*** e ***__TIME__***. Il formato della prima macro-variabile è '***"Mmm gg aaaa"***' e quello della seconda è '***"hh:mm:ss"***'. Come si vede, le due macro-variabili si espandono in una stringa delimitata correttamente da apici doppi.

Macro-variabile	Descrizione
`__FILE__` `__LINE__`	Attraverso le macro-variabili ***__FILE__*** e ***__LINE__*** il programma può accedere all'informazione sul nome del file sorgente e della riga originale. Il nome del file e il numero della riga possono essere alterati attraverso la direttiva '**`#line`**'.
`__STDC__` `__STDC_HOSTED__` `__STDC_VERSION__`	La macro-variabile ***__STDC__*** che si espande nel valore '**`1`**' sta a indicare che si tratta di un compilatore conforme allo standard; la macro ***__STDC_HOSTED__***, se si espande nel valore '**`1`**', indica una conformità stretta, definita come *hosted implementation*; la macro ***__STDC_VERSION__*** si espande nella versione dello standard. Il valore in cui si espande la terza macro-variabile contiene l'anno e il mese, come per esempio '**`199901L`**', con la specificazione che si tratta di una costante numerica di tipo '**`long int`**'.

Macro-variabile	Descrizione
`__STDC_IEC_559__`	Se esiste la macro-variabile ***__STDC_IEC_599__*** che si espande nel valore '**`1`**', si intende indicare la conformità alle specifiche dello standard IEC 60559, inerenti l'aritmetica a virgola mobile.
`__STDC_IEC_559_COMPLEX__`	Se esiste la macro-variabile ***__STDC_IEC_599_COMPLEX__*** che si espande nel valore '**`1`**', si intende indicare la conformità alle specifiche dello standard IEC 60559, inerenti l'aritmetica «complessa».
`__STDC_ISO_10646__`	Se esiste la macro-variabile ***__STDC_ISO_10646__***, questa dovrebbe espandersi nella versione dello standard ISO/IEC 10646 che riguarda la codifica universale dei caratteri. La versione che si ottiene è un numero contenente l'anno e il mese, seguito dalla lettera «L», a indicare che si tratta di una costante numerica di tipo '**`long int`**'.

A parte il caso di ***__FILE__*** e ***__LINE__***, le macro-variabili si espandono in un valore fisso.

66.2.11 Pragma

Attraverso i «pragma» è possibile dare al compilatore delle istruzioni che sono al di fuori dello standard. Il pragma, in sé, è un messaggio testuale che viene passato al compilatore, il quale può interpretarlo in fase di precompilazione o in quella successiva. Lo standard prevede due forme per esprimere un pragma al compilatore:

```
#pragma messaggio
```

```
_Pragma ("messaggio")
```

Il testo che compone il pragma nella sua prima forma viene trattato letteralmente, mentre quello del secondo modello richiede la protezione di alcuni caratteri: '`\"`' e '`\\`' corrispondono rispettivamente a '`"`' e '`\`'. I due esempi seguenti sono equivalenti:

```
#pragma GCC dependency "parse.y"
```

```
_Pragma ("GCC dependency \"parse.y\"")
```

Lo standard prevede anche che sia possibile creare delle macroistruzioni che incorporino un pragma, come nell'esempio seguente:

```
#define DO_PRAGMA(x) _Pragma (#x)
DO_PRAGMA (GCC dependency "parse.y")
```

66.3 Dal campo di azione alla compilazione

«

Il problema del campo di azione di variabili e funzioni va visto assieme a quello della compilazione di un programma composto da più file sorgenti, attraverso la produzione di file-oggetto distinti. Leggendo questo capitolo occorre tenere presente che la descrizione della questione è semplificata, omettendo alcuni dettagli. D'altra parte, per poter comprendere il problema la semplificazione è necessaria, tenendo conto che nel linguaggio C, per controllare il campo di azione delle variabili e delle funzioni, si utilizzano parole chiave non proprio «azzeccate» e in certi casi con significati diversi in base al contesto.

Per una descrizione più precisa e dettagliata, dopo la lettura di questo capitolo è necessario rivolgersi ai documenti che definiscono lo standard del linguaggio.

66.3.1 Il punto di vista del «collegatore»

«

Il programma che raccoglie assieme diversi file-oggetto per creare un file eseguibile (ovvero il *linker*), deve «collegare» i riferimenti incrociati a simboli di variabili e funzioni. In pratica, se nel file '`uno.o`' si fa riferimento alla funzione ***f()*** dichiarata nel file '`due.o`', nel programma risultante tale riferimento deve essere risolto con degli indirizzi appropriati. Naturalmente, lo stesso vale per le variabili globali, dichiarate da una parte e utilizzate anche dall'altra.

Per realizzare questi riferimenti incrociati, occorre che le variabili e le funzioni utilizzate al di fuori del file-oggetto in cui sono dichiarate, siano pubblicizzate in modo da consentire il richiamo da altri file-oggetto. Per quanto riguarda invece le variabili e le funzioni

dichiarate e utilizzate esclusivamente nello stesso file-oggetto, non serve questa forma di pubblicità.

Nei documenti che descrivono il linguaggio C standard si usa una terminologia specifica per distinguere le due situazioni: quando una variabile o una funzione viene dichiarata e usata solo internamente al file-oggetto rilocabile che si ottiene, è sufficiente che abbia una «collegabilità interna», ovvero un ***linkage interno***; quando invece la si usa anche al di fuori del file-oggetto in cui viene dichiarata, richiede una «collegabilità esterna», ovvero un ***linkage esterno***.

Nel linguaggio C, il fatto che una variabile o una funzione sia accessibile al di fuori del file-oggetto rilocabile che si ottiene, viene determinato in modo implicito, in base al contesto, nel senso che non esiste una classe di memorizzazione esplicita per definire questa cosa.

66.3.2 Campo di azione legato al file sorgente

Il file sorgente che si ottiene dopo l'elaborazione da parte del precompilatore, è suddiviso in componenti costituite essenzialmente dalla dichiarazione di variabili e di funzioni (prototipi inclusi). L'ordine in cui appaiono queste componenti determina la ***visibilità*** reciproca: in linea di massima si può accedere solo a quello che è già stato dichiarato. Inoltre, in modo predefinito, dopo la trasformazione in file-oggetto, queste componenti sono accessibili anche da altri file, per i quali, l'ordine di dichiarazione nel file originale non è più importante.[17]

Figura 66.116. Quattro file sorgenti equivalenti, a confronto. La variabile ***i***, la funzione ***f()*** e la funzione ***g()*** sarebbero accessibili anche da altri file. La funzione ***g()*** utilizza la variabile ***i***, dichiarata esternamente a lei.



Nell'esempio della figura precedente, la funzione ***g()*** accede direttamente alla variabile ***i*** che risulta dichiarata al di fuori della funzione stessa. Il campo di azione di questa variabile inizia dalla sua dichiarazione e termina alla fine del file; quando la variabile viene definita in una posizione successiva al suo utilizzo, questa deve essere dichiarata preventivamente come «esterna», attraverso lo specificatore di classe di memorizzazione '**`extern`**'.

Per isolare le funzioni e la variabile degli esempi mostrati, in modo che non siano disponibili per il collegamento con altri file, si dichiarano per il solo uso locale attraverso lo specificatore di classe di memorizzazione '**`static`**', come si vede nella figura successiva. Va osservato che, nell'ultimo caso, la variabile ***i*** non può essere isolata dall'esterno, perché si trova in una posizione successiva al suo utilizzo, pertanto vi si accede come se fosse dichiarata in un altro file.

Figura 66.117. Quattro file sorgenti equivalenti a confronto, in cui, dove è stato possibile, le variabili e le funzioni sono state isolate dal collegamento con l'esterno.



Per accedere a una funzione o a una variabile definita in un altro file[18] si deve dichiarare localmente la funzione o la variabile con lo specificatore di classe di memorizzazione '**`extern`**'. La figura successiva mostra l'esempio già apparso, ma diviso in due file.

Figura 66.118. Due file collegati tra di loro: il primo file («a») viene proposto in due versioni equivalenti.

file a

```
extern int f (int)
extern int i;

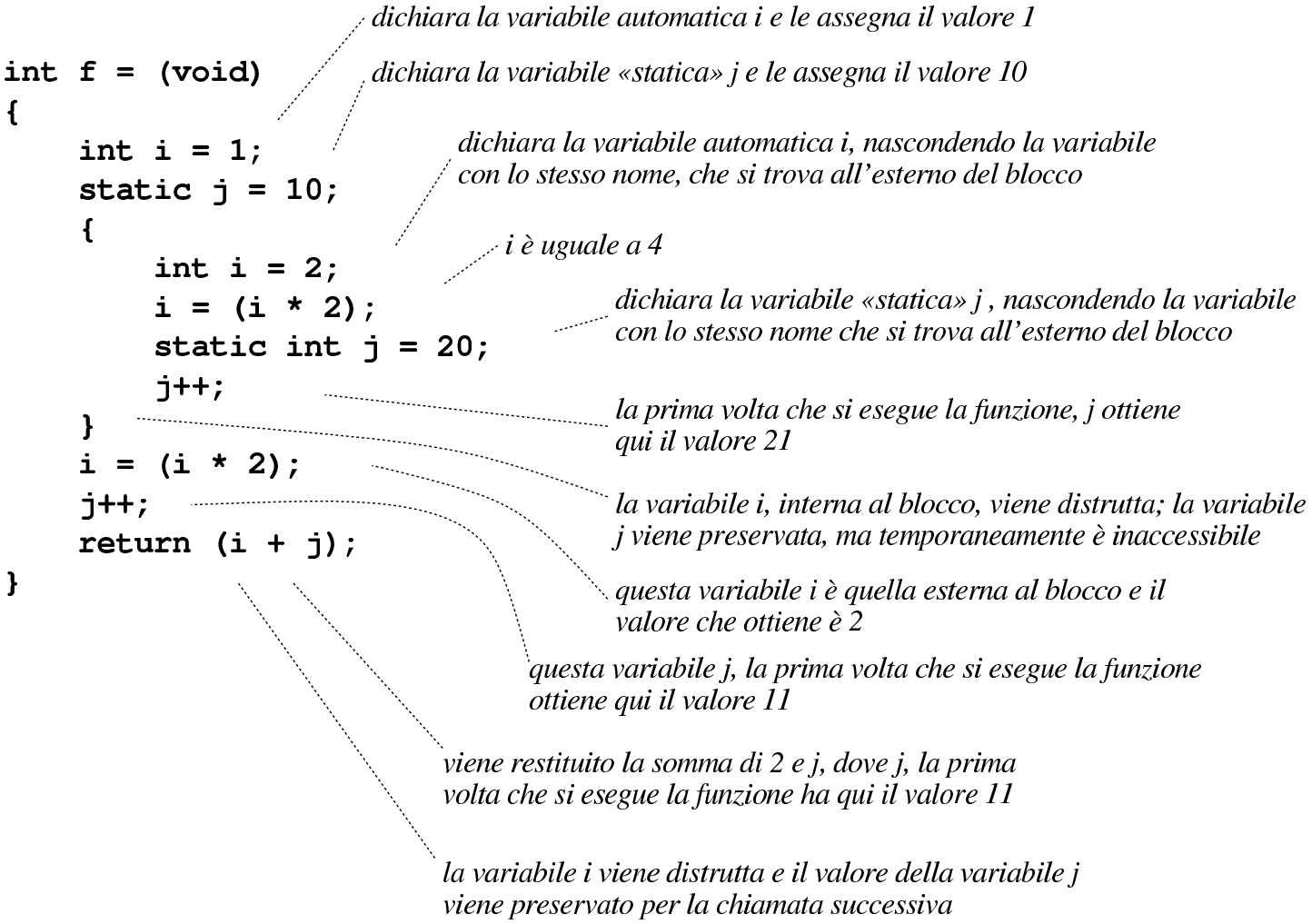
int g (void)
{
    i++;
    return f (i);
}
```

file b

```
int f (int x)
{
    return (x * x);
}

int i = 3;
```

✳ *equivale a:*

```
return f (4);
```

file a

```
extern int f (int)

int g (void)
{
    extern int i;
    i++;
    return f (i);
}
```

file b

```
int f (int x)
{
    return (x * x);
}

int i = 3;
```

Questi esempi mostrano che è possibile dichiarare la variabile «esterna» direttamente all'interno della funzione che ne fa uso; tuttavia, per la scrittura di un programma ordinato, è più grazioso se questa dichiarazione appare al di fuori delle funzioni.

Negli esempi mostrati non appare la funzione ***main()*** che, invece, in un programma comune deve esistere. È da osservare che la funzione ***main()*** non può essere dichiarata con lo specificatore di classe di memorizzazione '**`static`**', anche se tutto è incluso in un file unico, perché dopo la produzione del file-oggetto rilocabile, per produrre un file eseguibile si associano normalmente delle librerie che contengono il codice iniziale del programma, il quale va a chiamare poi

la funzione ***main()***. In altre parole, la compilazione prevede quasi sempre l'associazione con un file-oggetto fantasma contenente il codice responsabile della chiamata della funzione ***main()***, la quale, così, deve essere accessibile all'esterno del proprio file.

Tabella 66.119. Specificatori di classe di memorizzazione utilizzabili nella dichiarazione delle funzioni e delle variabili al di fuori delle funzioni.

Parola chiave	Descrizione
	L'assenza dello specificatore di classe implica la dichiarazione di una variabile o di una funzione accessibile anche da altri file.
`static`	Lo specificatore di classe '**`static`**' definisce una variabile o una funzione che può essere utilizzata solo all'interno del file in cui appare.
`extern`	Indica il riferimento a una variabile o a una funzione dichiarata in un altro file, oppure, nel caso delle variabili, anche nel file stesso ma in una posizione successiva.

66.3.3 Semplificazione dovuta all'uso comune dei file di intestazione

Nella tradizione del linguaggio C si fa uso di file di intestazione, ovvero porzioni di codice, in cui, tra le altre cose, si vanno a mettere i prototipi delle funzioni e le dichiarazioni delle variabili globali, a cui tutto il programma deve poter accedere.

Per semplificare questo lavoro di fusione, spesso un file incluso ne include automaticamente altri, da cui il proprio codice può dipendere. Così facendo, può anche succedere che lo stesso prototipo o la

stessa variabile appaiano dichiarati più volte nello stesso file finale (quello generato dal precompilatore).

Oltre a questo fatto, se il proprio programma è suddiviso in più file, i quali devono includere questo o quel file di intestazione, diventa impossibile precisare da quale parte i prototipi e le variabili vengono dichiarate e da quale altra parte vengono richiamate. Pertanto, di norma si lascia fare al compilatore. L'esempio di compilazione di due file, presentato alla fine della sezione precedente, va rivisto secondo quanto si vede nella figura successiva.

Figura 66.120. Due file collegati tra di loro senza dichiarare espressamente la classe di memorizzazione '**extern**'.



Naturalmente, è bene che le funzioni e le variabili pubbliche siano dichiarate sempre nello stesso modo; inoltre, se le variabili pubbliche devono essere inizializzate, ciò può avvenire una volta sola, in un solo file.

La classe di memorizzazione '**extern**' è predefinita per i prototipi di funzione (purché non siano incorporati all'interno di altre funzioni) e per la dichiarazione delle variabili, purché assieme alla dichiarazione non ci sia anche un'inizializzazione. In pratica, nell'esempio non si può dichiarare espressamente con la parola chiave '**extern**' la variabile ***i*** nel file '`b`', dove viene anche inizializzata. Se si tenta di farlo, il compilatore dovrebbe segnalare un errore.

66.3.4 Campo di azione interno alle funzioni

All'interno delle funzioni sono accessibili le variabili globali dichiarate esternamente a loro (come descritto nella sezione precedente), inoltre sono dichiarate implicitamente le variabili che costituiscono i parametri, dai quali si ricevono gli argomenti della chiamata, e si possono aggiungere altre variabili «locali». I parametri e le altre variabili che si dichiarano nella funzione sono visibili solo nell'ambito della funzione stessa; inoltre, se i nomi delle variabili e dei parametri sono gli stessi di variabili dichiarate esternamente, ciò rende temporaneamente inaccessibili quelle variabili esterne.

In condizioni normali, sia le variabili che costituiscono i parametri, sia le altre variabili dichiarate localmente all'interno di una funzione, vengono eliminate all'uscita dalla funzione stessa. Di norma ciò avviene utilizzando la pila dei dati che di solito ogni processo elaborativo dispone (si veda eventualmente la sezione 64.10).

Figura 66.121. Variabili «automatiche» dichiarate implicitamente come tali.



Le variabili create all'interno di una funzione, nel modo descritto dalla figura precedente, sono ***variabili automatiche*** ed è possibile esplicitare questa loro caratteristica con lo specificatore di classe di memorizzazione '**`auto`**'. Pertanto, la stessa cosa sarebbe stata ottenuta scrivendo l'esempio come nella figura successiva.

Figura 66.122. Variabili «automatiche» dichiarate espressamente attraverso lo specificatore di classe di memorizzazione '**auto**'.

```
int h = 1;
int i = 2;
int j = 3;
int k = 4;

float f (int x)
{
    auto int h;
    auto float i = 1.5;
    h = ((2 * j) * i);
    auto float m;
    m = (k / 2.0);
    return (m + i);
}
```

All'interno di una funzione è possibile utilizzare variabili che facciano riferimento a porzioni di memoria che non vengono rilasciate all'uscita della funzione stessa, pur isolandole rispetto alle variabili dichiarate esternamente. Si ottiene questo con lo specificatore di classe di memorizzazione '**static**' che non va confuso con lo stesso specificatore usato per le variabili dichiarate esternamente alle funzioni. In altre parole, quando in una funzione si dichiara una variabile con lo specificatore di classe di memorizzazione '**static**', si ottiene di conservare il contenuto di quella variabile che torna a essere accessibile nelle chiamate successive della funzione.

Di norma, la dichiarazione di una variabile di questo tipo coinci-

de con la sua inizializzazione; in tal caso, l'inizializzazione avviene solo quando si chiama la funzione la prima volta.

Figura 66.123. Variabili «statiche» (da intendersi come variabili private) dichiarate all'interno delle funzioni.

```
int i = 2;

int f (void)
{
    static int i = 0;
    i++;
    return i;
}
```

viene creata la variabile i che nasconde quella dichiarata esternamente
l'inizializzazione della variabile avviene solo la prima volta
viene incrementata di una unità la variabile i
viene restituito il valore della variabile i
il contenuto della variabile i viene preservato per la prossima chiamata

All'interno delle funzioni possono essere usati anche gli specificatori di classe di memorizzazione '**`register`**' e '**`extern`**', come descritto nella tabella successiva.

Tabella 66.124. Specificatori di classe di memorizzazione utilizzabili nella dichiarazione delle variabili all'interno delle funzioni.

Parola chiave	Descrizione
`auto`	È lo specificatore di classe di memorizzazione predefinito e indica che la variabile viene creata in corrispondenza della dichiarazione e viene eliminata all'uscita della funzione.
`register`	Con lo specificatore di classe di memorizzazione '**`register`**' si chiede di creare una variabile automatica che, se possibile, utilizzi un registro del microprocessore o qualunque altra risorsa limitata che possa ridurne i tempi di accesso.

Parola chiave	Descrizione
`static`	Definisce una variabile «privata» allocando della memoria che non viene rilasciata alla conclusione dell'attività della funzione, conservando il valore memorizzato per la chiamata successiva della stessa funzione. Si tratta comunque di una variabile a cui può accedere solo la funzione in cui è dichiarata.
`extern`	Indica il riferimento a una variabile dichiarata «esternamente» (come già mostrato nella sezione precedente). In generale, sarebbe meglio dichiarare in questo modo solo le variabili che sono definite al di fuori delle funzioni, lasciando che le funzioni vi accedano semplicemente in qualità di variabili globali.

66.3.5 Campo di azione interno ai raggruppamenti di istruzioni

Le variabili dichiarate all'interno di raggruppamenti di istruzioni, ovvero all'interno di parentesi graffe, si comportano esattamente come quelle dichiarate all'interno delle funzioni: il loro campo di azione termina all'uscita dal blocco. L'esempio della figura successiva mostra un raggruppamento di istruzioni contenente la dichiarazione di una variabile automatica e di una «statica», con la descrizione dettagliata di ciò che accade, dentro e fuori dal raggruppamento.

Figura 66.125. Vita delle variabili all'interno dei raggruppamenti di istruzioni.



La dimostrazione serve a comprendere che, all'interno di una funzione, la posizione in cui si dichiara una variabile non è indifferente: in generale, per migliorare la leggibilità del codice, sarebbe bene dichiarare le variabili all'inizio delle funzioni, evitando accuratamente di farlo all'interno di raggruppamenti annidati.

66.3.6 Funzioni annidate

«

Così come esistono i raggruppamenti di istruzioni, all'interno dei quali la dichiarazione delle variabili ha un proprio campo di azione limitato, è possibile anche dichiarare delle sottofunzioni, accessibili

solo all'interno delle funzioni stesse, dopo che sono state dichiarate. Queste sottofunzioni non possono avere uno specificatore di classe di memorizzazione e appartengono esclusivamente alla funzione che le contiene.

In generale, l'uso di sottofunzioni è sconsigliabile e, d'altra parte, originariamente non era permesso.

66.3.7 Visibilità, accessibilità, staticità

Va chiarita la distinzione che c'è tra la visibilità di una variabile e l'accessibilità al suo contenuto. Quando una funzione dichiara delle variabili automatiche o statiche con un certo nome, se questa funzione chiama a sua volta un'altra funzione che al suo interno fa uso di variabili con lo stesso nome, queste ultime non si riferiscono alla prima funzione. Si osservi l'esempio del listato seguente.

Listato 66.126. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/bZoOp7vp* , *http://ideone.com/ZpYU4* .

```
#include <stdio.h>

int x = 100;

int f (void)
{
    return x;
}

int main (int argc, char *argv[])
{
    int x = 7;
    printf ("x == %i\n", x);
```

```
    printf ("f() == %i\n", f());
    return 0;
}
```

Avviando questo programma si ottiene il testo seguente:

```
x == 7
f() == 100
```

In pratica, la funzione ***f()*** che utilizza la variabile ***x***, si riferisce alla variabile con quel nome, dichiarata esternamente alle funzioni, che risulta inizializzata con il valore 100, ignorando perfettamente che la funzione ***main()*** la sta chiamando mentre gestisce una propria variabile automatica con lo stesso nome. Pertanto, la variabile automatica ***x*** della funzione ***main()*** non è visibile alle funzioni che questa chiama a sua volta.

D'altra parte, anche se la variabile automatica ***x*** non risulta visibile, il suo contenuto può essere accessibile, dal momento della sua dichiarazione fino alla fine della funzione (ma questo richiede l'uso di puntatori, come descritto nella sezione 66.5). Alla fine dell'esecuzione della funzione, tutte le sue variabili automatiche perdono la propria identità, in quanto scaricate dalla pila dei dati, e il loro spazio di memoria può essere utilizzato per altri dati (per altre variabili automatiche di altre funzioni).

Si osservi che lo stesso risultato si otterrebbe anche se la variabile ***x*** della funzione ***main()*** fosse dichiarata come statica:

```
...
int main (int argc, char *argv[])
{
    static int x = 7;
    printf ("x == %i\n", x);

    printf ("f() == %i\n", f());
    return 0;
}
```

Le variabili statiche, siano esse dichiarate al di fuori o all'interno delle funzioni, hanno in comune il fatto che utilizzano la memoria dal principio alla fine del funzionamento del programma, anche se dal punto di vista del programma stesso non sono sempre visibili. Pertanto, il loro spazio di memoria sarebbe sempre accessibile, anche se sono oscurate temporaneamente o se ci si trova fuori dal loro campo di azione, attraverso l'uso di puntatori. Naturalmente, il buon senso richiede di mettere la dichiarazione di variabili statiche al di fuori delle funzioni, se queste devono essere manipolate da più di una di queste.

Le variabili che utilizzano memoria dal principio alla fine dell'esecuzione del programma, ma non sono statiche, sono quelle variabili dichiarate all'esterno delle funzioni, per le quali il compilatore predispone un simbolo che consenta la loro identificazione nel file-oggetto. Il fatto di non essere statiche (ovvero il fatto di guadagnare un simbolo di riconoscimento nel file-oggetto) consente loro di essere condivise tra più file (intesi come unità di traduzione), ma per il resto valgono sostanzialmente le stesse regole di visibilità. Il buon senso stesso fa capire che tali variabili possano essere dichiarate solo esternamente alle funzioni, perché dentro le funzioni si usa preva-

lentemente la pila dei dati e perché comunque, ciò che è dichiarato dentro la funzione deve avere una visibilità limitata.

66.3.8 Compilazione di un progetto composto da più file

«

Viene riproposto l'esempio utilizzato più volte in questo capitolo, nella sua versione per due file, completandolo con una funzione ***main()***, in modo da poterlo compilare e dimostrare i passaggi necessari in situazioni del genere.

Listato 66.129. File '`a.c`'.

```
#include <stdio.h>

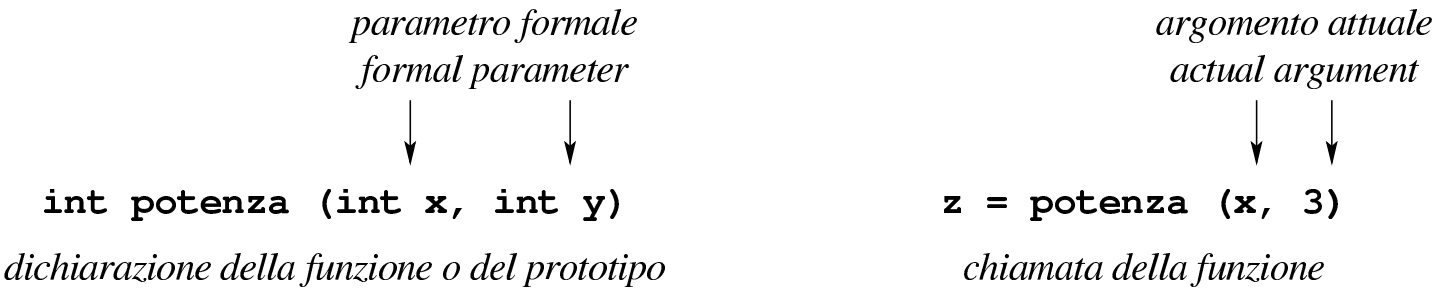
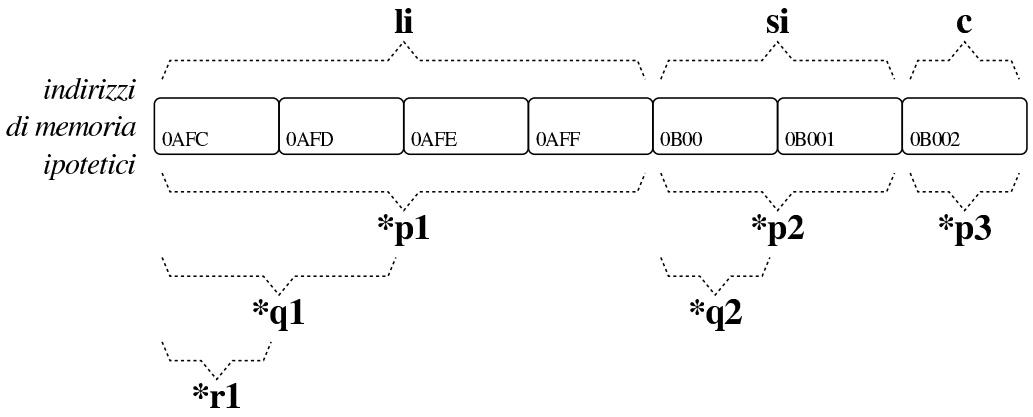
int f (int);
int i;

int g (void)
{
    i++;
    return f (i);
}

int main (void)
{
    printf ("valore originale di i = %i, ", i);
    printf ("valore restituito da g() = %i\n", g());
    printf ("valore originale di i = %i, ", i);
    printf ("valore restituito da g() = %i\n", g());
    printf ("valore originale di i = %i, ", i);
    printf ("valore restituito da g() = %i\n", g());
    printf ("valore originale di i = %i, ", i);
    printf ("valore restituito da g() = %i\n", g());
    return 0;
}
```

Listato 66.130. File ‘`b.c`’.

```
int f (int x)
{
    return (x * x);
}

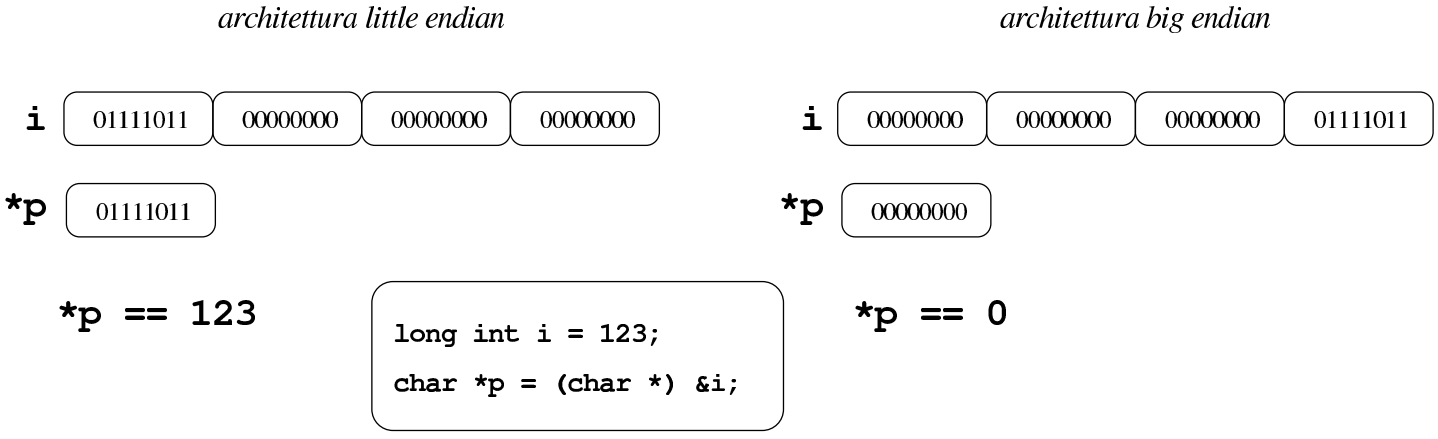
int i = 1;
```

Disponendo di più file sorgenti separati, la compilazione avviene in due fasi: la generazione dei file oggetto e il «collegamento» (*link*) di questi in modo da ottenere un file eseguibile. Fortunatamente, tutto questo può essere gestito tramite lo stesso compilatore ‘**`cc`**’.

Per generare i file oggetto si utilizza ‘**`cc`**’ con l’opzione ‘**`-c`**’; se si può disporre del compilatore GNU C, è meglio aggiungere anche l’opzione ‘**`-Wall`**’. Si suppone che il primo file sia stato nominato ‘`a.c`’ e il secondo ‘`b.c`’. Si inizia dalla compilazione dei singoli file in modo da generare i file oggetto ‘`a.o`’ e ‘`b.o`’.

```
$ cc -Wall -c a.c[Invio]

$ cc -Wall -c b.c[Invio]
```

Quindi si passa all’unione dei due risolvendo i riferimenti incrociati, generando il file eseguibile ‘**`prova`**’.

```
$ cc -o prova a.o b.o[Invio]
```

Ecco cosa si dovrebbe vedere eseguendo il file che si ottiene dalla compilazione:

```
$ ./prova[Invio]
```

```
valore originale di i = 1, valore restituito da g() = 4
valore originale di i = 2, valore restituito da g() = 9
valore originale di i = 3, valore restituito da g() = 16
valore originale di i = 4, valore restituito da g() = 25
```

Per un uso migliore del compilatore si veda la parte 65.

66.3.9 Osservazioni sulla vita delle costanti letterali

«

Una costante letterale può essere gestita dal compilatore come meglio crede, ma quando si tratta di un'informazione che non può risiedere completamente in una parola del microprocessore e non si può collocare in un'istruzione del linguaggio macchina, è evidente che debba essere conservata nella memoria usata dal programma. Si osservi l'esempio seguente:

```
void f (void)
{
    char x[] = "ciao amore";
    printf ("%s\n", x);
}
```

L'array ***x[]***, o meglio, il puntatore che lo rappresenta, viene creato ogni volta alla chiamata della funzione ***f()*** e anche distrutto alla sua conclusione. Ma questo array viene inizializzato ogni volta con una stringa prestabilita, la quale deve essere disponibile per tutto il tempo di funzionamento del programma. In altri termini, quella stringa è un array senza nome allocato in memoria dal principio dell'esecuzione del programma, pertanto al di fuori della pila dei dati.

66.3.10 Libreria standard e file di intestazione

La libreria standard del linguaggio C prevede la disponibilità di una serie di funzioni, macro del precompilatore e tipi di dati per usi specifici.

Dal punto di vista del programmatore, si ha la percezione della presenza di questa libreria attraverso l'inclusione dei «file di intestazione», ovvero di quei file che per tradizione hanno un nome che finisce per '`.h`' e si incorporano attraverso le direttive '**`#include`**' del precompilatore. Tuttavia, di norma le funzioni della libreria standard sono contenute in un file-oggetto già compilato (che può essere realizzato in forma differente, a seconda che serva per l'accesso dinamico alle funzioni, oppure che debba essere incorporato nel file eseguibile finale, come spiegato nella sezione 65.7), noto come libreria C, o solo Libc, che viene incluso automaticamente nella compilazione di un progetto, a meno di escluderlo espressamente.

Con il compilatore GNU C, per escludere l'utilizzo di qualunque libreria predefinita vanno usate le opzioni '**`-nostartfiles`**' e '**`-nodefaultlibs`**'; eventualmente l'opzione '**`-nostdlibs`**' dovrebbe valere per entrambe queste opzioni e può essere usata assieme a loro, benché sia ridondante.

Anche se la libreria C viene realizzata nel modo descritto, il concetto di libreria standard non si esaurisce nei file-oggetto che contengono le sue funzioni, perché rimane la necessità di dichiarare le macro del precompilatore, i tipi di dati che fanno parte dello standard complessivo, ma soprattutto i prototipi delle funzioni che compongono

la libreria. Pertanto, i file di intestazione rimangono indispensabili e fanno parte integrante della libreria.

A titolo dimostrativo, si può osservare il programma seguente che, pur facendo uso della libreria standard, in quanto si sfrutta la funzione ***printf()***, non incorpora alcun file di intestazione. In tal caso, però, è indispensabile dichiarare il prototipo della funzione utilizzata:

```
extern int printf (const char *format,...);

int main (void)
{
    printf ("Ciao a tutti!\n");
    return 0;
}
```

66.4 Annotazioni sulla terminologia

«

I documenti che descrivono lo standard del linguaggio C utilizzano una terminologia specifica. Qui si descrivono alcuni di quei termini con delle annotazioni riguardo al contesto a cui si riferiscono.

66.4.1 Parametri e argomenti

«

Generalmente, i termini «argomento» e «parametro», riferiti alle funzioni o alle procedure dei linguaggi di programmazione, vengono usati in modo intercambiabile, benché si intuisca una differenza tra i due. Lo standard C chiarisce l'ambito corretto di utilizzo per entrambi: i valori annotati in una chiamata di funzione sono gli argomenti attuali; le variabili che descrivono formalmente ciò che una funzione deve ricevere dall'esterno sono i parametri formali.

Figura 66.134. Distinzione tra parametri e argomenti.



66.4.2 Byte e caratteri

Secondo il linguaggio C, il byte è l'unità di memorizzazione più piccola che possa essere utilizzata per contenere un carattere, tra quelli dell'insieme minimo. Pertanto, per definizione, il tipo '**char**' (indifferentemente se con o senza segno) occupa esattamente un byte.

In pratica, per il linguaggio C il byte non è necessariamente un insieme di otto bit, anche se di norma questa corrispondenza è valida.

Va considerato anche che il tipo '**char**', senza altre indicazioni, può essere inteso come valore con segno o senza segno, a seconda della piattaforma. Tuttavia, come punto fermo, l'insieme di caratteri minimo deve essere rappresentabile con valori positivi. In pratica, di norma questo insieme minimo di caratteri corrisponde alla codifica ASCII, la quale si rappresenta completamente con 7 bit, pertanto l'ottavo bit di un byte standard potrebbe essere usato come segno, senza interferire con l'interpretazione corretta dei caratteri. In altri termini, per utilizzare il tipo '**char**' in modo compatibile da una piattaforma all'altra, questo va considerato solo per i valori utili alla rappresentazione dell'insieme di caratteri minimo, con i quali si ha la certezza di avere a che fare sempre solo con valori positivi.

66.4.3 Unità di traduzione

«

Il file generato dal precompilatore, formato normalmente dall'incorporazione di diversi file, viene definito una ***unità di traduzione***. Il concetto di «traduzione» deriva dal fatto che il precompilatore, oltre a incorporare altri file, traduce le macro-variabili e le macroistruzioni espandendole secondo la loro dichiarazione; pertanto, i file sorgenti originali subiscono una prima trasformazione che produce il codice C vero e proprio.

Quando si fa riferimento al campo di azione delle variabili definite al di fuori delle funzioni, si afferma che questo riguarda l'ambito del file. In tal caso, per file si intende l'unità di traduzione.

66.4.4 «Linkage»

«

Quando si fa riferimento a variabili o funzioni che sono dichiarate esternamente a tutte le funzioni, il campo di azione è legato al file (nel senso di unità di traduzione), essendo accessibili solo a partire dalla dichiarazione stessa. Quando si combinano assieme più file attraverso il meccanismo del «collegamento» (*link*), il programma che esegue questo compito tratta i nomi uguali di variabili e di funzioni nel senso di un riferimento alla stessa cosa (la stessa variabile o la stessa funzione). Quando una variabile o una funzione è dichiarata in modo tale da consentire questo collegamento, si ha un ***linkage esterno***; quando la dichiarazione è tale da impedirlo (con lo specificatore di classe di memorizzazione '`static`'), si ha un ***linkage interno***.

Si può rendere esplicito che una variabile o una funzione sono da cercarsi al di fuori del proprio file, oppure in una posizione più avanzata dello stesso file, richiedendo un *linkage esterno* con lo specificatore

di classe di memorizzazione '**`extern`**'. In tal caso, si può collegare esternamente anche una variabile indicata all'interno di una funzione o di un altro tipo di blocco, sempre con lo specificatore '**`extern`**'.

Le variabili che, diversamente, sono dichiarate all'interno di un blocco di qualunque genere, non sono collegabili, soprattutto nel caso delle variabili automatiche, la cui vita dipende dal blocco in cui sono contenute.

66.4.5 Durata di memorizzazione

Nella documentazione standard si usa spesso il termine *storage duration*, ovvero ***durata di memorizzazione***, per fare riferimento al tempo di vita di una certa informazione contenuta in memoria.

Di norma si possono distinguere due casi fondamentali: ciò che viene memorizzato in un'area di memoria sempre disponibile (anche se non è detto che a ogni parte del programma sia consentito di accedervi) e ciò che si mette nella pila dei dati. Nel primo caso di parla di *static storage duration*, in quanto i dati stanno lì e non si muovono; nel secondo si parla di *automatic storage duration*, in quanto la memoria della pila viene liberata e riutilizzata in modo dinamico.

È per questa ragione che, nella dichiarazione delle variabili all'interno delle funzioni, esiste lo specificatore di classe '**`static`**', a indicare una variabile che, pur essendo accessibile solo all'interno della funzione, va collocata al di fuori della pila dei dati, in modo da conservare il proprio contenuto durante le chiamate successive della stessa funzione.

66.4.6 «Lvalue» e «rvalue»

«

Nello standard del linguaggio C, il termine *lvalue* indica, approssimativamente, ciò che appare a sinistra di un operatore di assegnamento, nelle condizioni per cui ciò è ammissibile. Per esempio, nell'espressione seguente, la variabile ***x*** rappresenta un *lvalue*:

```
x = 3;
```

L'espressione seguente, invece, **non è valida**, perché la costante '**3**' non può essere un *lvalue*:

```
3 = x;  // Non è valida, perché «3» non è un «lvalue».
```

Il termine poteva significare, originariamente, *left-value*, da contrapporsi a un possibile *right-value*, costituito da ciò che in un'espressione si trova alla destra dell'operatore di assegnamento. Tuttavia, lo standard attuale definisce la sigla in questione un *location value*, ovvero un'espressione che si riferisce a un'area di memorizzazione.

Un'espressione che sia un *lvalue* deve anche consentire la lettura dell'area di memorizzazione a cui si riferisce; pertanto, ciò che è un *lvalue* deve poter essere usato alla destra di un operatore di assegnamento (in qualità di *rvalue*). D'altra parte, non è garantito che un *lvalue* individui sempre un'area di memorizzazione modificabile, dal momento che esistono variabili qualificate come costanti, alle quali si assegna un valore in fase di dichiarazione, ma successivamente non è più consentita la modifica. Per distinguere anche questa situazione, volendo escludere il caso delle costanti, si specifica che l'espressione *lvalue* deve anche essere modificabile.

Tabella 66.137. Operatori che richiedono un operando di tipo *lvalue*. In tutti i casi, escluso '**&*lvalue***', deve trattarsi di un ***lvalue*** modificabile in quel contesto.

Parola chiave	Descrizione
&***lvalue***	Indirizzo di ***lvalue***.
++***lvalue*** ***lvalue***++ −−***lvalue*** ***lvalue***−−	Incremento e decremento.

Parola chiave	Descrizione
lvalue*=*rvalue ***lvalue*+=*rvalue*** ***lvalue*−=*rvalue*** ***lvalue**=*rvalue*** ***lvalue*%=*rvalue*** ***lvalue*<<=*rvalue*** ***lvalue*>>=*rvalue*** ***lvalue*&=*rvalue*** ***lvalue*^=*rvalue*** ***lvalue*|=*rvalue*** ***lvalue*~=*rvalue***	Assegnamenti.

Attualmente, lo standard C, al posto di *rvalue*, preferisce esprimere il concetto come «valore di un'espressione».

66.4.7 «Digraph» e «Trigraph»

«

In varie situazioni lo standard C consente l'utilizzo di sequenze speciali di caratteri, in sostituzione di simboli che in certi contesti potrebbero mancare, essendo invece indispensabili. In generale, quando per la scrittura dei file sorgenti si può contare su un insieme di

caratteri pari a quello della codifica ASCII, queste sequenze speciali non vanno usate assolutamente, perché complicano terribilmente la lettura dei file. A ogni modo, conviene essere a conoscenza della loro esistenza e del significato che assumono.

Digraph	*Trigraph*	Carattere corrispondente
`<:`	`??(`	[
`:>`	`??)`	]
`<%`	`??<`	{
`%>`	`??>`	}
`%:`	`??=`	#
`%:%:`	`??=??=`	##
	`??!`	|
	`??'`	^
	`??/`	\
	`??-`	~

66.5 Puntatori, array, stringhe e allocazione dinamica della memoria

All'inizio del capitolo sono stati mostrati solo i tipi di dati più semplici. Per poter utilizzare gli array si gestiscono dei puntatori alle zone di memoria contenenti tali strutture.

Quando si ha a che fare con i puntatori è importante considerare che il modello di memoria che si ha di fronte è un'astrazione, nel senso che una struttura di dati appare idealmente continua, mentre nella realtà il compilatore potrebbe anche provvedere a scomporla in blocchi separati.

Nella spiegazione che si fa qui, come nelle altre sezioni del capitolo, l'esposizione è semplificata rispetto alle definizioni dello standard; pertanto, per un approccio più preciso ci si deve rivolgere ai documenti ufficiali sul linguaggio C.

66.5.1 Espressioni a cui si assegnano dei valori

«

Quando si utilizza un operatore di assegnamento, come '**=**' o altri operatori composti, ciò che si mette alla sinistra rappresenta la «variabile ricevente» del risultato dell'espressione che si trova alla destra dell'operatore (nel caso di operatori di assegnamento composti, l'espressione alla destra va considerata come quella che si ottiene scomponendo l'operatore). Ma il linguaggio C consente di rappresentare quella «variabile ricevente» attraverso un'espressione, come nel caso dei puntatori che vengono descritti in questo capitolo. Pertanto, per evitare confusione, la documentazione dello standard chiama l'espressione a sinistra dell'operatore di assegnamento un *lvalue* (*Left value* o *Location value*).

Nel capitolo si evita questa terminologia, tuttavia è importante comprendere che un'espressione può rappresentare una «variabile», pur senza averle dato un nome (nella sezione 66.4.6 il concetto di *lvalue* e di *rvalue* viene descritto con migliore dettaglio).

66.5.2 Puntatori

Una variabile, di qualunque tipo sia, rappresenta normalmente un valore posto da qualche parte nella memoria del sistema.[19] Quando si usano i tipi di dati normali, è il compilatore a prendersi cura di tradurre i riferimenti agli spazi di memoria rappresentati simbolicamente attraverso dei nomi.

Attraverso l'operatore di indirizzamento e-commerciale ('**&**'), è possibile ottenere il puntatore (riferito alla rappresentazione ideale di memoria del linguaggio C) a una variabile «normale». Tale valore può essere inserito in una variabile particolare, adatta a contenerlo: una ***variabile puntatore***.

Per esempio, se ***p*** è una variabile puntatore adatta a contenere l'indirizzo di un intero, l'esempio mostra in che modo assegnare a tale variabile il puntatore alla variabile ***i***:

```
int i = 10;
…
p = &i;   // L'indirizzo di «i» viene assegnato al
          // puntatore «p».
```

La dichiarazione di una variabile puntatore avviene in modo simile a quello delle variabili normali, con l'aggiunta di un asterisco prima del nome. L'esempio seguente dichiara la variabile ***p*** come puntatore a un tipo '**int**'. Si osservi che va indicato il tipo di dati a cui si punta, perché questa informazione è parte integrante del puntatore.

```
int *p;
```

Non deve essere interesse del programmatore il modo esatto in cui si rappresentano i puntatori dei vari tipi di dati, diversamente non

ci sarebbe l'utilità di usare un linguaggio come il C invece di un semplice assemblatore di linguaggio macchina.

Una volta dichiarata la variabile puntatore, questa viene utilizzata normalmente, senza asterisco, finché si intende fare riferimento al puntatore stesso.

L'asterisco usato nella dichiarazione serve a definire il tipo di dati, quindi, '**`int *`*p**' rappresenta la dichiarazione della variabile ***p*** di tipo '**`int *`**'. Tuttavia si può fare un ragionamento leggermente differente, con l'aiuto delle parentesi: '**`int (*`*p**`)`**' è la dichiarazione di una zona di memoria senza nome, di tipo '**`int`**', a cui punta la variabile ***p*** attraverso la dereferenziazione ***p**. Le due cose sono equivalenti, in quanto portano comunque alla creazione della variabile ***p*** di tipo puntatore a intero, ma la seconda forma consente di comprendere, successivamente, la sintassi per la creazione di un puntatore a funzione.

È importante chiarire subito in che modo si dichiarano più variabili puntatore con una sola istruzione; si osservi l'esempio seguente in cui si creano le variabili ***p*** e ***p2***, in particolare per il fatto che l'asterisco va ripetuto:

```
int *p, *p2;
```

Attraverso l'operatore di «dereferenziazione», l'asterisco ('**`*`**'), è possibile accedere alla zona di memoria a cui la variabile punta. Per «dereferenziare» si intende quindi l'azione con cui si toglie il riferimento e si raggiungono i dati a cui un puntatore si riferisce.[20]

Attenzione a non fare confusione con gli asterischi: una cosa è quello usato per dichiarare o per dereferenziare un puntatore e un'altra è l'operatore con cui invece si ottiene la moltiplicazione.

L'esempio già accennato potrebbe essere chiarito nel modo seguente, dove si mostra anche la dichiarazione della variabile puntatore:

```
int i = 10;
int *p;
…
p = &i;
```

A questo punto, dopo aver assegnato a ***p*** il puntatore alla variabile ***i***, è possibile accedere alla stessa area di memoria in due modi diversi: attraverso la variabile ***i***, oppure attraverso la dereferenziazione di ***p***, ovvero la traduzione ****p***.

```
int i = 10;
int *p;
…
p = &i;
…
*p = 20;
```

Nell'esempio, l'istruzione '****p*=20**' è tecnicamente equivalente a '***i*=20**'. Per chiarire un po' meglio il ruolo delle variabili puntatore, si può complicare l'esempio nel modo seguente:

```
int i = 10;
int *p;
int *p2;
…
p = &i;
…
p2 = p;
…
*p2 = 20;
```

In particolare è stata aggiunta una seconda variabile puntatore, ***p2***, solo per fare vedere che è possibile passare un puntatore anche ad altre variabili senza dover usare l'asterisco. Comunque, in questo caso, '***p2*=20**' è tecnicamente equivalente sia a '***p*=20**', sia a '***i*=20**'.

Si osservi che l'asterisco è un operatore che, evidentemente, ha la precedenza rispetto a quelli di assegnamento. Eventualmente, anche in questo caso si possono usare le parentesi per togliere ambiguità al codice:

```
int i = 10;
int *p;
…
p = &i;
…
(*p2) = 20;
```

Come accennato inizialmente, il tipo di dati a cui un puntatore si rivolge, fa parte integrante del puntatore stesso. Ciò è importante perché quando si dereferenzia un puntatore occorre sapere quanto è grande l'area di memoria a cui si deve accedere a partire dal puntatore. Per questa ragione, quando si assegna a una variabile puntatore un altro puntatore, questo deve essere compatibile, nel senso che de-

ve riferirsi allo stesso tipo di dati, altrimenti si rischia di ottenere un risultato inatteso. A questo proposito, l'esempio seguente contiene probabilmente un errore:

```
char *pc;
int  *pi;
…
pi = pc;   // I due puntatori si riferiscono a dati di tipo
           // differente!
…
```

Quando invece si vuole trasformare realmente un puntatore in modo che si riferisca a un tipo di dati differente, si può usare un cast, come si farebbe per convertire i valori numerici:

```
char *pc;
int  *pi;
…
pi = (int *) pc;   // Il programmatore dimostra di essere
                   // consapevole di ciò che sta facendo
                   // attraverso un cast!
…
…
```

Nello schema seguente appare un esempio che dovrebbe consentire di comprendere la differenza che c'è tra i puntatori, in base al tipo di dati a cui fanno riferimento. In particolare, ***p1***, ***q1*** e ***r1*** fanno tutti riferimento all'indirizzo ipotetico $0AFC_{16}$, ma l'area di memoria che considerano è diversa, pertanto ****p1***, ****q1*** e ****r1*** sono tra loro «variabili» differenti, anche se si sovrappongono parzialmente.




```
long int li;
short int si;
char c;

long int *p1 = &li;
short int *p2 = &si;
char *p3 = &c;

short int *q1 = (short int *) &li;
char *q2 = (char *) &si;

char * r1 = (char *) &li;
```

L'esempio seguente rappresenta un programma completo che ha lo scopo di determinare se l'architettura dell'elaboratore è di tipo *big endian* o di tipo *little endian*. Per capirlo si dichiara una variabile di tipo '**`long int`**' che si intende debba essere di rango superiore rispetto al tipo '**`char`**', assegnandole un valore abbastanza basso da poter essere rappresentato anche in un tipo '**`char`**' senza segno. Con un puntatore di tipo '**`char *`**' si vuole accedere all'inizio della variabile contenente il numero intero '**`long int`**': se già nella porzione letta attraverso il puntatore al primo «carattere» si trova il valore assegnato alla variabile di tipo intero, vuol dire che i byte sono invertiti e si ha un'architettura *little endian*, mentre diversamente si presume che sia un'architettura *big endian*.

Listato 66.149. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/Abe4VgIo* , *http://ideone.com/WVpmK* .

```
#include <stdio.h>
int main (void)
{
    long int i = 123;
    char *p = (char *) &i;
    if (*p == 123)
      {
        printf ("little endian\n");
      }
    else
      {
        printf ("big endian\n");
      }
    return 0;
}
```

Figura 66.150. Schematizzazione dell'operato del programma di esempio, per determinare l'ordine dei byte usato nella propria architettura.



Il linguaggio C utilizza il passaggio degli argomenti alle funzioni per valore; per ottenere il passaggio per riferimento occorre utilizzare

dei puntatori. Si immagini di volere realizzare una funzione banale che modifica la variabile utilizzata nella chiamata, sommandovi una quantità fissa. Invece di passare il valore della variabile da modificare, si può passare il suo puntatore; in questo modo la funzione (che comunque deve essere stata realizzata appositamente per questo scopo) agisce nell'area di memoria a cui punta questo puntatore.

```
…
void funzione_stupida (int *x)
{
     (*x)++;
}
…
int main (void)
{
     int y = 10;
     …
     funzione_stupida (&y);
     …
     return 0;
}
```

L'esempio mostra la dichiarazione e descrizione di una funzione che non restituisce alcun valore e ha un parametro costituito da un puntatore a un intero. Il lavoro della funzione è solo quello di incrementare il valore contenuto nell'area di memoria a cui si riferisce tale puntatore.

Poco dopo, nella funzione ***main()*** inizia il programma vero e proprio; viene dichiarata la variabile ***y*** corrispondente a un intero normale inizializzato a 10, poi, a un certo punto viene chiamata la funzione vista prima, passando il puntatore a ***y***.

Il risultato è che dopo la chiamata, la variabile ***y*** contiene il valore

precedente incrementato di un'unità.

Quando si usano i puntatori, invece delle variabili comuni, occorre considerare che se la vita della variabile a cui un puntatore fa riferimento si è esaurita, il puntatore relativo diventa privo di valore. Questo significa che il fatto di avere conservato il puntatore a una certa area di memoria **non** implica automaticamente la garanzia che tale zona contenga dati validi o che sia ancora raggiungibile.

66.5.3 Array

Nel linguaggio C, l'array è una sequenza ordinata di elementi dello stesso tipo nella rappresentazione ideale di memoria di cui si dispone. In questo senso, quando si dichiara un array, quello che il programmatore ottiene in pratica è il riferimento alla posizione iniziale di questo, mentre gli elementi successivi si raggiungono tenendo conto della lunghezza di ogni elemento.

Questo ragionamento vale in senso generale ed è un po' approssimativo. In contesti particolari, il riferimento a un array restituisce qualcosa di diverso dal puntatore al primo elemento.

Visto in questi termini, si può intendere che l'array in C è sempre a una sola dimensione, tutti gli elementi devono essere dello stesso tipo in modo da avere la stessa lunghezza e la quantità degli elementi, una volta definita, è fissa.

È compito del programmatore ricordare la quantità di elementi che compone l'array, perché determinarlo diversamente è complicato e

a volte non è possibile. Inoltre, quando un programma tenta di accedere a una posizione oltre il limite degli elementi esistenti, c'è il rischio che non si verifichi alcun errore, arrivando però a dei risultati imprevedibili.

Lo standard prescrive che sia consentito raggiungere l'indirizzo successivo all'ultimo elemento, anche se tale contenuto diventa privo di significato. Ciò serve a garantire che non si provochino errori nell'accesso alla memoria, se l'indice va oltre il limite di un array, ma per una sola posizione, per leggere un contenuto privo di utilità. In pratica, ciò significa che dopo un array ci deve essere qualunque altra variabile, o al limite uno spazio inutilizzato. Ma questo è compito del compilatore.

La dichiarazione di un array avviene in modo intuitivo, definendo il tipo degli elementi e la loro quantità. L'esempio seguente mostra la dichiarazione dell'array ***a*** di sette elementi di tipo '**`int`**':

```
int a[7];
```

Per accedere agli elementi dell'array si utilizza un indice, il cui valore iniziale è sempre zero e, di conseguenza, quello con cui si raggiunge l'elemento ***n***-esimo deve avere il valore ***n***-1. L'esempio seguente mostra l'assegnamento del valore 123 al **secondo** elemento:

```
a[1] = 123;
```

In presenza di array monodimensionali che hanno una quantità ridotta di elementi, può essere sensato attribuire un insieme di valori iniziale all'atto della dichiarazione.

Alcuni compilatori consentono l'inizializzazione degli array solo quando questi sono dichiarati all'esterno delle funzioni, con un campo di azione globale, oppure all'interno delle funzioni, ma dichiarati come «statici», nel senso che continuano a esistere all'uscita della funzione.

```
int a[] = {123, 453, 2, 67};
```

L'esempio mostrato dovrebbe chiarire in che modo si possono dichiarare gli elementi dell'array, tra parentesi graffe, togliendo così la necessità di specificare la quantità di elementi. Tuttavia, le due cose possono coesistere:

```
int a[10] = {123, 453, 2, 67};
```

In tal caso, l'array si compone di 10 elementi, di cui i primi quattro con valori prestabiliti, mentre gli altri ottengono il valore zero. Si osservi però che il contrario non può essere fatto:

```
int a[5] = {123, 453, 2, 67, 32, 56, 78};      // Non si può!
```

Gli standard recenti del linguaggio C consentono anche la dichiarazione di array per i quali il compilatore non può sapere subito la quantità di elementi da predisporre, **purché ciò avvenga nel campo di azione delle funzioni** (o di blocchi inferiori). In pratica, in questi casi è possibile indicare la quantità di elementi attraverso un'espressione che si traduca in un numero intero, come nell'esempio seguente, dove la quantità di elementi è data dal prodotto tra la variabile ***s*** e la costante 3:

```
int s = 33;
…
int a[s * 3];
```

Gli array dichiarati al di fuori delle funzioni (quelli il cui campo di azione è legato al file) e quelli che, pur essendo dichiarati nelle funzioni, continuano a esistere per tutto il tempo di esecuzione del programma (in quanto «statici»), possono avere soltanto una quantità di elementi già stabilita in fase di compilazione. Per fare riferimento a array definiti in altri file, oppure in posizioni più avanzate dello stesso file, è possibile usare una dichiarazione «esterna», nella quale è bene specificare la quantità di elementi, ma questa deve essere coerente con quella della dichiarazione a cui si fa riferimento:

```
extern int i[3];
…
int i[3];
```

In alternativa si può fare una dichiarazione esterna di un array senza specificarne la quantità di elementi, ma questo implica che, fino a quando non appare la dichiarazione completa, l'array sia di tipo incompleto e non si possa determinare la sua dimensione con l'aiuto dell'operatore '**sizeof**':

```
extern int i[];          // Tipo incompleto.
…
int i[3];
```

La scansione di un array avviene generalmente attraverso un'iterazione enumerativa, in pratica con un ciclo '**for**' che si presta particolarmente per questo scopo. Si osservi l'esempio seguente:

```
int a[7];
int i;
…
for (i = 0; i < 7; i++)
  {
    …
    a[i] = …;
    …
  }
```

L'indice ***i*** viene inizializzato a zero, in modo da cominciare dal primo elemento dell'array; il ciclo può continuare fino a che ***i*** continua a essere inferiore a sette, infatti l'ultimo elemento dell'array ha indice sei; alla fine di ogni ciclo, prima che riprenda il successivo, viene incrementato l'indice di un'unità.

Per scandire un array in senso opposto, si può agire in modo analogo, come nell'esempio seguente:

```
int a[7];
int i;
…
for (i = 6; i >= 0; i--)
  {
    …
    a[i] = …;
    …
  }
```

Questa volta l'indice viene inizializzato in modo da puntare alla posizione finale; il ciclo viene ripetuto fino a che l'indice è maggiore o uguale a zero; alla fine di ogni ciclo, l'indice viene decrementato di un'unità.

Se non si può conoscere la dimensione dell'array, questa deve essere calcolata con l'ausilio dell'operatore '**sizeof**', come nell'esempio seguente, ammesso che il contesto sia tale da consentire all'operatore di restituire un valore valido:

```
// Da qualche parte si dichiara il valore di «x» come numero
// intero.
…
int a[7 * x];
int i;
…
int s = (sizeof a) / (sizeof (a[0]));
for (i = 0; i < s; i++)
  {
    …
    a[i] = …;
    …
  }
```

Il calcolo della quantità di elementi è ottenuto determinando la dimensione dell'array in byte e dividendo tale valore per la dimensione in byte di un intero, ovvero per la dimensione di ogni elemento dell'array stesso.

Quando un array è argomento dell'operatore '**sizeof**', si ottiene la dimensione complessiva dell'array stesso (nell'unità gestita da '**sizeof**'. Tuttavia occorre considerare che, se l'array non è ancora stato definito nella sua dimensione, non si può avere il risultato atteso.

66.5.4 Array multidimensionali

Gli array in C sono monodimensionali, però nulla vieta di creare un array i cui elementi siano array tutti uguali. Per esempio, nel modo seguente, si dichiara un array di cinque elementi che a loro volta sono insiemi di sette elementi di tipo '**int**'. Nello stesso modo si possono definire array con più di due dimensioni.

```
int a[5][7];
```

L'esempio seguente mostra il modo normale di scandire un array a due dimensioni:

```
int a[5][7];
int i;
int j;
…
for (i = 0; i < 5; i++)
  {
    …
    for  (j = 0; j < 7; j++)
      {
        …
        a[i][j] = …;
        …
      }
    …
  }
```

Anche se in pratica un array a più dimensioni è solo un array «normale» in cui si individuano dei sottogruppi di elementi, la scansione deve avvenire sempre indicando formalmente lo stesso numero di elementi prestabiliti per le dimensioni rispettive, anche se dovrebbe essere possibile attuare qualche trucco. Per esempio, tornando al

listato mostrato, se si vuole scandire in modo continuo l'array, ma usando un solo indice, bisogna farlo gestendo l'ultimo:

```
int a[5][7][9];
int j;
…
for (j = 0; j < (5 * 7 * 9); j++)
  {
    …
    a[0][0][j] = …;
    …
  }
```

Rimane comunque da osservare il fatto che questo non sia un bel modo di programmare.

Anche gli array a più dimensioni possono essere inizializzati, secondo una modalità analoga a quella usata per una sola dimensione, con la differenza che l'informazione sulla quantità di elementi per dimensione non può essere omessa. L'esempio seguente è un programma completo, in cui si dichiara e inizializza un array a due dimensioni, per poi mostrarne il contenuto:

Listato 66.166. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/Ht9r6QwN* , *http://ideone.com/xzD7f* .

```
#include <stdio.h>

int main (int argc, char *argv[])
{
  int a[3][4] = {{1,  2,  3,  4},
                 {5,  6,  7,  8},
                 {9, 10, 11, 12}};
  int i, j;

  for (i = 0; i < 3; i++)
    {
      for (j = 0; j < 4; j++)
        {
          printf ("a[%i][%i]=%i\t", i, j, a[i][j]);
        }
      printf ("\n");
    }

  return 0;
}
```

Il programma dovrebbe mostrare il testo seguente:

```
a[0][0]=1       a[0][1]=2       a[0][2]=3       a[0][3]=4
a[1][0]=5       a[1][1]=6       a[1][2]=7       a[1][3]=8
a[2][0]=9       a[2][1]=10      a[2][2]=11      a[2][3]=12
```

Anche nell'inizializzazione di un array a più dimensioni si possono omettere degli elementi, come nell'estratto seguente:

```
…
  int a[3][4] = {{1, 2},
                 {5, 6, 7, 8}};
…
```

In tal caso, il programma si mostrerebbe così:

```
a[0][0]=1       a[0][1]=2       a[0][2]=0       a[0][3]=0
a[1][0]=5       a[1][1]=6       a[1][2]=7       a[1][3]=8
a[2][0]=0       a[2][1]=0       a[2][2]=0       a[2][3]=0
```

Di certo, pur sapendo di voler utilizzare un array a più dimensioni, si potrebbe pretendere di inizializzarlo come se fosse a una sola, come nell'esempio seguente, ma il compilatore dovrebbe avvisare del fatto:

```
…
  int a[3][4] = {1, 2, 3, 4, 5, 6,            // Così non è
                 7, 8, 9, 10, 11, 12};        // grazioso.
…
```

66.5.5 Natura dell'array

Inizialmente si è accennato al fatto che quando si crea un array, quello che viene restituito in pratica è un puntatore alla sua posizione iniziale, ovvero all'indirizzo del primo elemento di questo. Si può intuire che non sia possibile assegnare a un array un altro array, anche se ciò potrebbe avere significato. Al massimo si può assegnare elemento per elemento.

Per evitare errori del programmatore, la variabile che contiene l'indirizzo iniziale dell'array, quella che in pratica rappresenta l'array stesso, è in **sola lettura**. Quindi, nel caso dell'array già visto, la va-

riabile ***a*** non può essere modificata, mentre i singoli elementi ***a[i]*** sì:

```
int a[7];
```

Data la filosofia del linguaggio C, se fosse possibile assegnare un valore alla variabile ***a***, si modificherebbe il puntatore, facendo in modo che questo punti a un array differente. Ma per raggiungere questo risultato vanno usati i puntatori in modo esplicito. Si osservi l'esempio seguente.

Listato 66.172. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/SLI8GS82* , *http://ideone.com/RImyk* .

```
#include <stdio.h>

int main (void)
{
  int a[3];
  int *p;

  p = a;          // «p» diventa un alias dell'array «a».

  p[0] = 10;      // Si può fare solo con gli array
  p[1] = 100;     // a una sola dimensione.
  p[2] = 1000;    //

  printf ("%i %i %i \n",  a[0], a[1], a[2]);

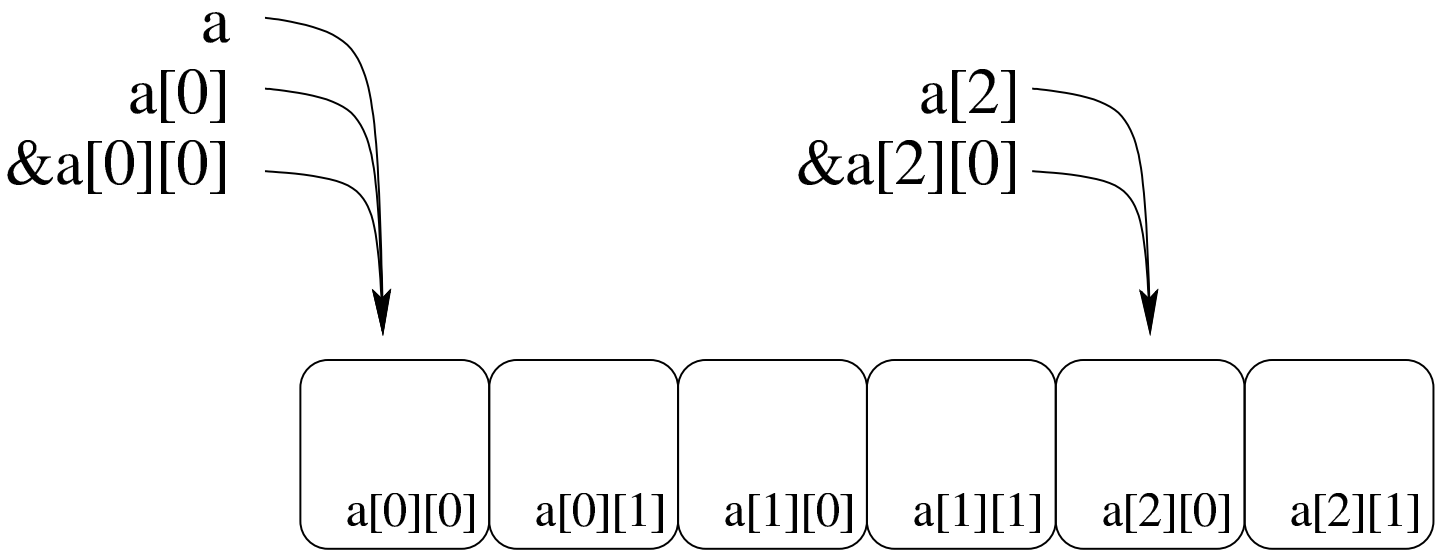
  return 0;
}
```

Viene creato un array, ***a***, di tre elementi di tipo '**int**', e subito dopo una variabile puntatore, ***p***, al tipo '**int**'. Si assegna quindi alla

variabile ***p*** il puntatore rappresentato da ***a***; da quel momento si può fare riferimento all'array indifferentemente con il nome ***a*** o ***p***.

Si può osservare anche che l'operatore '**&**', seguito dal nome di un array, produce ugualmente l'indirizzo dell'array che è equivalente a quello fornito senza l'operatore stesso, con la differenza che riguarda l'array nel suo complesso:

```
…
p = &a;     // I due puntatori non sono dello stesso tipo!
…
```

Pertanto, in questo caso si pone il problema di compatibilità del tipo di puntatore che si può risolvere con un cast esplicito:

```
…
p = (int *) &a;    // «p» diventa un alias dell'array «a».
…
```

In modo analogo, si può estrapolare l'indice che rappresenta l'array dal primo elemento, cosa che si ottiene senza incorrere in problemi di compatibilità tra i puntatori. Si veda la trasformazione dell'esempio nel modo seguente.

Listato 66.175. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/91Df91s1dq* , *http://ideone.com/6rPpE* .

```
#include <stdio.h>

int main (void)
{
  int a[3];
  int *p;

  p = &a[0];     // «p» diventa un alias dell'array «a».

  p[0] = 10;     // Si può fare solo con gli array
  p[1] = 100;    // a una sola dimensione.
  p[2] = 1000;   //

  printf ("%i %i %i \n",  a[0], a[1], a[2]);

  return 0;
}
```

Anche se si può usare un puntatore come se fosse un array, va osservato che la variabile ***p***, in quanto dichiarata come puntatore, viene considerata in modo differente dal compilatore; per esempio non è possibile determinare la dimensione dell'array a cui punta attraverso l'operatore '**sizeof**', perché si otterrebbe semplicemente la quantità di byte che costituisce la variabile puntatore.

Quando si opera con array a più dimensioni, il riferimento a una porzione di array restituisce l'indirizzo della porzione considerata. Per esempio, si supponga di avere dichiarato un array a due dimensioni, nel modo seguente:

```
int a[3][2];
```

Se a un certo punto, in riferimento allo stesso array, si scrivesse '**a[2]**', si otterrebbe l'indirizzo del terzo gruppo di due interi:



Tenendo d'occhio lo schema appena mostrato, considerato che si sta facendo riferimento all'array ***a*** di 3×2 elementi di tipo '**int**', va osservato che:

- in condizioni normali '**a**' si traduce nel puntatore a un array di due elementi di tipo '**int**';
- '**a[0]**' e '**&a[0][0]**' si traducono nel puntatore a un elemento di tipo '**int**' (precisamente il primo);
- '**&a**' si traduce nel puntatore a un array composto da 3×2 elementi di tipo '**int**'.

Pertanto, se questa volta si volesse assegnare a una variabile puntatore di tipo '**int ***' l'indirizzo iniziale dell'array, nell'esempio seguente si creerebbe un problema di compatibilità:

```
    …
    int a[3][2];
    int *p;
    p = a;     // I due puntatori non sono dello stesso tipo!
    …
```

Pertanto, occorrerebbe riferirsi all'inizio dell'array in modo differente oppure attraverso un cast.

66.5.6 Puntatori costanti

Si può far sì che un puntatore funzioni in modo più simile a quello di un array a una sola dimensione, dichiarando il puntatore come costante, nel senso che il puntatore in sé non può essere cambiato:

```
…
    int a[3];
    int *const p = a;  // Puntatore in sola lettura.
    p[1] = 9;
    p = a;                // Questo non si può!
…
```

L'esempio seguente, invece, fa sì che la memoria a cui si vuole accedere tramite il puntatore sia protetta in sola lettura:

```
…
    int a[3];
    const int *p = a;    // Qui è la memoria a essere
                         // in sola lettura.
    p[1] = 9;            // Questo non si può!
    p = a;
…
```

Anche se si può bloccare il puntatore, così da farlo funzionare in modo equivalente a un array vero e proprio, rimane però il fatto che

‘**sizeof**’, usato per «misurare» un puntatore, restituisce comunque la grandezza della variabile che costituisce il puntatore stesso. Inoltre ci sono altre questioni che riguardano i puntatori, affrontate in una sezione separata, a proposito dell’aritmetica dei puntatori.

66.5.7 Array e funzioni

«

Si è visto che le funzioni possono accettare solo parametri composti da tipi di dati elementari, compresi i puntatori. In questa situazione, l’unico modo per trasmettere a una funzione un array attraverso i parametri, è quello di inviarne il puntatore iniziale. Di conseguenza, le modifiche che vengono poi apportate da parte della funzione si riflettono nell’array di origine. Si osservi l’esempio seguente.

Listato 66.181. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/2Zibjb5j* , *http://ideone.com/geaQV* .

```
#include <stdio.h>

void elabora (int *p)
{
    p[0] = 10;
    p[1] = 100;
    p[2] = 1000;
}

int main (void)
{
    int a[3];

    elabora (a);
    printf ("%i %i %i \n",  a[0], a[1], a[2]);
```

```
    return 0;
}
```

La funzione ***elabora()*** utilizza un solo parametro, rappresentato da un puntatore a un tipo '**int**'. La funzione **presume** che il puntatore si riferisca all'inizio di un array di interi e così assegna alcuni valori ai primi tre elementi (anche il numero degli elementi non può essere determinato dalla funzione).

All'interno della funzione ***main()*** viene dichiarato l'array ***a*** di tre elementi interi e subito dopo viene passato come argomento alla funzione ***elabora()***. Così facendo, in realtà si passa il puntatore al primo elemento dell'array.

Infine, la funzione altera gli elementi come è già stato descritto e gli effetti si possono osservare così:

```
10 100 1000
```

L'esempio potrebbe essere modificato per presentare la gestione dell'array in modo più elegante. Per la precisione si tratta di ritoccare la funzione '**elabora**':

```
void elabora (int a[])
{
    a[0] = 10;
    a[1] = 100;
    a[2] = 1000;
}
```

Si tratta sostanzialmente della stessa cosa, solo che si pone l'accento sul fatto che l'argomento è un array di interi, benché di tipo incompleto.

In entrambi i casi, se all'interno della funzione si tenta di misurare la dimensione dell'array con l'operatore '**sizeof**', si ottiene solo la grandezza della variabile usata per contenere il puntatore relativo. Sarebbe anche possibile specificare la dimensione dell'array, senza però che questo fatto abbia delle conseguenze significative e senza che '**sizeof**' la consideri:

```
void elabora (int a[3])  // Anche così sizeof restituisce
{                        // solo la grandezza del puntatore.
    a[0] = 10;
    a[1] = 100;
    a[2] = 1000;
}
```

66.5.8 Aritmetica dei puntatori

Con le variabili puntatore è possibile eseguire delle operazioni elementari: possono essere incrementate e decrementate. Il risultato che si ottiene è il riferimento a una zona di memoria adiacente, in funzione della dimensione del tipo di dati per il quale è stato creato il puntatore. Si osservi l'esempio seguente:

```
int i = 10;
int j;
int *p = &i;
p++;
j = *p;          // Attenzione!
```

In questo caso viene creato un puntatore al tipo '**int**' che inizialmente contiene l'indirizzo della variabile ***i***. Subito dopo questo puntatore viene incrementato di una unità e ciò comporta che si riferisca a un'area di memoria adiacente, immediatamente successiva a quella occupata dalla variabile ***i*** (molto probabilmente si tratta dell'area

occupata dalla variabile ***j***). Quindi si tenta di copiare il valore di tale area di memoria, interpretato come '**int**', all'interno della variabile ***j***.

Se un programma del genere funziona nell'ambito di un sistema operativo che controlla l'utilizzo della memoria, se l'area che si tenta di raggiungere incrementando il puntatore non è stata allocata, si ottiene un «errore di segmentazione» e l'arresto del programma stesso. L'errore si verifica quando si tenta l'accesso, mentre la modifica del puntatore è sempre lecita.

Lo stesso meccanismo riguarda tutti i tipi di dati che non sono array, perché per gli array, l'incremento o il decremento di un puntatore riguarda i componenti dell'array stesso. In pratica, quando si gestiscono tramite puntatori, gli array sono da intendere come una serie di elementi dello stesso tipo e dimensione, dove, nella maggior parte dei casi, il nome dell'array si traduce nell'indirizzo del primo elemento:

```
int i[3] = { 1, 3, 5 };
int *p;
…
p = i;
```

Nell'esempio si vede che il puntatore ***p*** punta all'inizio dell'array di interi ***i[]***.

```
*p = 10; // Equivale a:  i[0] = 10.
p++;
*p = 30; // Equivale a:  i[1] = 30.
p++;
*p = 50; // Equivale a:  i[2] = 50.
```

Ecco che, incrementando il puntatore, si accede all'elemento adia-

cente successivo, in funzione della dimensione del tipo di dati. Decrementando il puntatore si ottiene l'effetto opposto, di accedere all'elemento precedente. La stessa cosa avrebbe potuto essere ottenuta così, senza alterare il valore contenuto nella variabile ***p***:

```
*(p + 0) = 10; // Equivale a:  i[0] = 10.
*(p + 1) = 30; // Equivale a:  i[1] = 30.
*(p + 2) = 50; // Equivale a:  i[2] = 50.
```

Inoltre, come già visto in altre sezioni, si potrebbe usare il puntatore con la stessa notazione propria dell'array, ma ciò solo perché si opera a una sola dimensione:

```
p[0] = 10; // Equivale a:  i[0] = 10.
p[1] = 30; // Equivale a:  i[1] = 30.
p[2] = 50; // Equivale a:  i[2] = 50.
```

Questo lascia intuire che '**i[*n*]**' corrisponda in pratica a '***(i + n)***', cosa che è vera per lo standard del linguaggio, ma potrebbe non essere accettabile dal compilatore che si usa effettivamente:

```
*(i + 0) = 10; // Equivale a:  i[0] = 10.
*(i + 1) = 30; // Equivale a:  i[1] = 30.
*(i + 2) = 50; // Equivale a:  i[2] = 50.
```

In presenza di più dimensioni, il ragionamento è analogo. Nel modello seguente, le lettere ***i*** e ***j*** rappresentano gli indici usati per la scansione, mentre le lettere ***I*** e ***J*** sono la quantità di elementi della dimensione corrispondente. Per esempio, secondo il modello seguente, in un array ***x[10][30]***, la lettera ***J*** corrisponde a 30.

```
x[i][j] == *(x + (i * J) + j)
```

In modo analogo si dovrebbe procedere per dimensioni maggiori:

```
x[i][j][k] == *(x + (i*J*K) + (j*K) + k)
```

Se il compilatore non accetta questo modo di gestire un array, il meccanismo vale per un puntatore dello stesso tipo degli elementi dell'array (che punti all'inizio dell'array stesso). L'esempio seguente mette in evidenza l'uso di un puntatore per scandire un array a due dimensioni.

Listato 66.191. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/tnGTykMU* , *http://ideone.com/ogdeH* .

```
#include <stdio.h>

int main (int argc, char *argv[])
{

  int a[3][4] = {{1,  2,  3,  4},
                 {5,  6,  7,  8},
                 {9, 10, 11, 12}};
  int i, j;
  const int *p = (int *) a;
  int x;

  for (i = 0; i < 3; i++)
    {
      for (j = 0; j < 4; j++)
        {
          x = *(p + i * 4 + j);
          //
          printf ("a[%i][%i]=%i\t", i, j, x);
          //
```

```
        }
      printf ("\n");
    }

  return 0;
}
```

I punti più importanti dell'esempio appaiono evidenziati: trattandosi di un array a più di una dimensione, la copia del puntatore avviene con l'ausilio di un cast; la scansione degli indirizzi, a partire dal puntatore ***p*** avviene attraverso una formula, mentre la forma seguente ha un significato diverso, descritto in un'altra sezione, a proposito dei puntatori a puntatori:

```
...
            x = p[i][j];          // Non è la stessa cosa!
...
```

La versione funzionante dell'esempio mostrato deve fare apparire il testo seguente:

```
a[0][0]=1       a[0][1]=2       a[0][2]=3       a[0][3]=4
a[1][0]=5       a[1][1]=6       a[1][2]=7       a[1][3]=8
a[2][0]=9       a[2][1]=10      a[2][2]=11      a[2][3]=12
```

Naturalmente, quando si usano direttamente i puntatori, è compito esclusivo del programmatore sapere quando l'incremento o il decremento di un puntatore ha significato. Diversamente si rischia di accedere a zone di memoria estranee al contesto di proprio interesse, con risultati imprevedibili.

Prima di concludere l'argomento, vale la pena di tradurre il problema dell'aritmetica dei puntatori in modo opposto, ovvero co-

me indirizzi. Per esempio, dato l'array ***a[]***, a una sola dimensione, si può considerare equivalente la notazione '`&(a[i])`' rispetto a '`(a + i)`'.

66.5.9 Osservazioni sui puntatori

Ammesso che la variabile ***p*** sia un puntatore a qualcosa, la notazione ****p*** equivale a '`p[0]`', così come '`*(p+n)`' corrisponde '`p[n]`'. Pertanto, l'uso delle parentesi quadre contenenti un indice, poste dopo il nome di una variabile puntatore, corrisponde alla dereferenziazione che si fa con l'asterisco.

Ammesso che la variabile ***p*** sia un puntatore a qualcosa, la notazione '`&*p`' corrisponde sempre a '***p***', anche se si tratta di un puntatore nullo.

Ammesso che la variabile ***x*** sia tale da potervi assegnare un valore e che possa essere operando di '`&`', la notazione '`*&x`' corrisponde sempre a '***x***'.

Ammesso che la variabile ***p*** sia un puntatore a qualcosa, la notazione '`*(tipo)p`' individua un'area di memoria che parte dalla posizione indicata dal puntatore e si estende per la dimensione del tipo indicato. In altre parole, si tratta di un cast con il quale si trasforma il tipo di puntatore al volo, ma per questo occorre mostrare un esempio.

Listato 66.194. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/89Ev895Myz* , *http://ideone.com/2A0nj* .

```
#include <stdio.h>
int main (int argc, char *argv[])
{
  int x = 10;
```

```
  void *p = &x;
  printf ("%i\n", *(int *) p);
  return 0;
}
```

In questo caso, il puntatore ***p*** è di tipo indefinito (‘**void**’) e riceve l’indirizzo della variabile ***x***. Successivamente, il valore a cui punta ***p*** viene usato all’interno della funzione ***printf()***, ma prima di essere dereferenziato, viene convertito in un puntatore di tipo ‘**int ***’.

66.5.10 Stringhe

Le stringhe, nel linguaggio C, non sono un tipo di dati a sé stante; si tratta solo di array di caratteri con una particolarità: l’ultimo carattere è sempre zero, ovvero una sequenza di bit a zero, che si rappresenta simbolicamente come carattere con ‘**\0**’. In questo modo, si evita di dover accompagnare le stringhe con l’informazione della loro lunghezza.

Pertanto, va osservato che una stringa è sempre un array di caratteri, ma un array di caratteri non è necessariamente una stringa, in quanto per esserlo occorre che l’ultimo elemento sia il carattere ‘**\0**’. Seguono alcuni esempi che servono a comprendere questa distinzione.

```
char c[20];
```

L’esempio mostra la dichiarazione di un array di caratteri, senza specificare il suo contenuto. Per il momento non si può parlare di stringa, soprattutto perché per essere tale, la stringa deve contenere dei caratteri.

```
char c[] = {'c', 'i', 'a', 'o'};
```

Questo esempio mostra la dichiarazione di un array di quattro caratteri. All'interno delle parentesi quadre non è stata specificata la dimensione perché questa si determina dall'inizializzazione. Anche in questo caso non si può ancora parlare di stringa, perché manca la terminazione.

```
char z[] = {'c', 'i', 'a', 'o', '\0'};
```

Questo esempio mostra la dichiarazione di un array di cinque caratteri corrispondente a una stringa vera e propria. L'esempio seguente è tecnicamente equivalente, solo che utilizza una rappresentazione più semplice:

```
char z[] = "ciao";
```

Pertanto, la stringa rappresentata dalla costante '**"ciao"**' è un array di cinque caratteri, perché, pur senza mostrarlo, include implicitamente anche la terminazione.

L'indicazione letterale di una stringa può avvenire attraverso sequenze separate, senza l'indicazione di alcun operatore di concatenamento. Per esempio, '**"ciao amore\n"**' è perfettamente uguale a '**"ciao " "amore" "\n"**' che viene inteso come una costante unica.

In un sorgente C ci sono varie occasioni di utilizzare delle stringhe letterali (delimitate attraverso gli apici doppi), senza la necessità di dichiarare l'array corrispondente. Però è importante tenere presente la natura delle stringhe per sapere come comportarsi con loro. Per prima cosa, bisogna rammentare che la stringa, anche se espressa in forma letterale, è un array di caratteri; come tale restituisce sem-

plicemente il puntatore del primo di questi caratteri (salvo le stesse eccezioni che riguardano tutti i tipi di array).

```
char *p;
…
p = "ciao";
…
```

L'esempio mostra il senso di quanto affermato: non esistendo un tipo di dati «stringa», si può assegnare una stringa solo a un puntatore al tipo '**char**' (ovvero a una variabile di tipo '**char ***'). L'esempio seguente non è valido, perché non si può assegnare un valore alla variabile che rappresenta un array, dal momento che il puntatore relativo è un valore costante:

```
char z[];
…
z = "ciao";      // Non si può.
…
```

Quando si utilizza una stringa tra gli argomenti della chiamata di una funzione, questa riceve il puntatore all'inizio della stringa. In pratica, si ripete la stessa situazione già vista per gli array in generale.

Listato 66.201. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/qCgdnWnE* , *http://ideone.com/kkzaT* .

```
#include <stdio.h>

void elabora (char *z)
{
  printf (z);
}

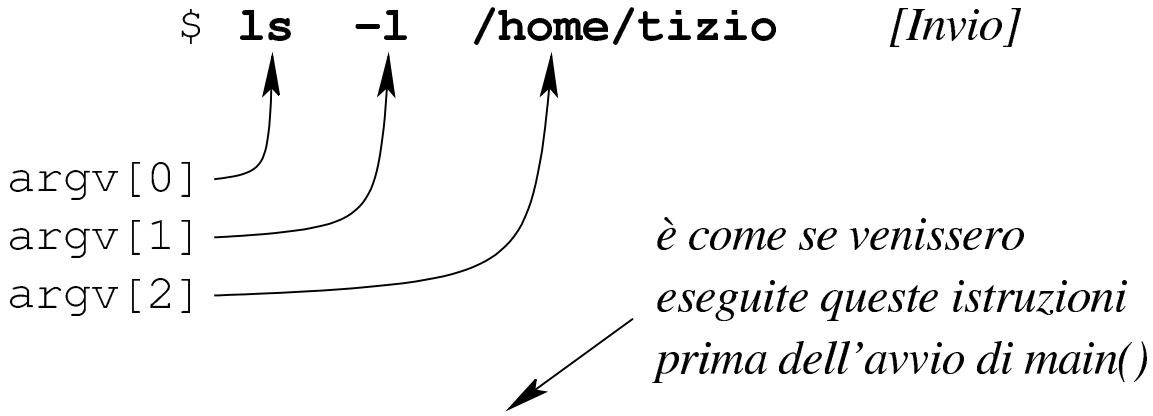
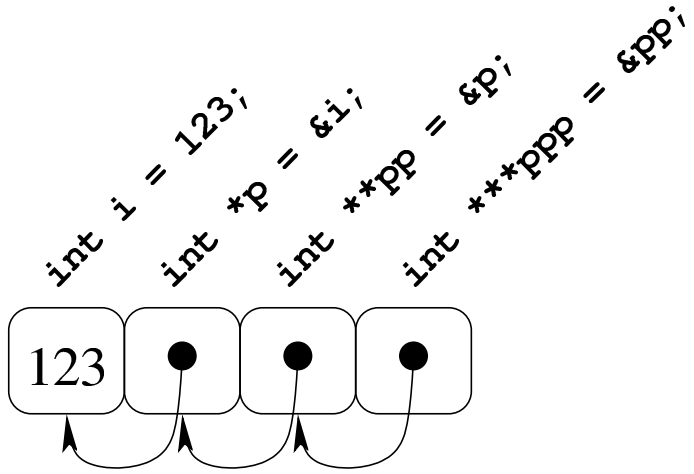
int main (void)
```

```
{
  elabora ("ciao\n");
  return 0;
}
```

L'esempio mostra una funzione banale che si occupa semplicemente di emettere la stringa ricevuta come parametro, utilizzando ***printf()***. La variabile utilizzata per ricevere la stringa è stata dichiarata come puntatore al tipo '**char**' (ovvero come puntatore di tipo '**char ***'), poi tale puntatore è stato utilizzato come argomento per la chiamata della funzione ***printf()***. Volendo scrivere il codice in modo più elegante si potrebbe dichiarare apertamente la variabile ricevente come array di caratteri di dimensione indefinita. Il risultato è lo stesso.

Listato 66.202. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/83oK83pP4x* , *http://ideone.com/uwq6D* .

```
#include <stdio.h>

void elabora (char z[])
{
  printf (z);
}

int main (void)
{
  elabora ("ciao\n");
  return 0;
}
```

Tabella 66.203. Funzioni comuni per la gestione delle stringhe, definite nel file '`string.h`' (il modificatore '**`restrict`**' viene descritto in una sezione apposita).

Funzione	Descrizione
`char *strcpy (char *restrict dst, const char *restrict org);` `char *strncpy (char *restrict dst, const char *restrictorg, size_t n);`	La funzione ***strcpy()*** copia il contenuto della stringa ***org*** nella stringa ***dst***, compreso il carattere di terminazione *<NUL>*. Perché l'operazione possa avvenire è necessario che le due stringhe non si sovrappongano e che per la stringa di destinazione ci sia abbastanza spazio per i caratteri da copiare. La funzione restituisce il puntatore all'inizio della stringa di destinazione. La funzione ***strncpy()*** si comporta sostanzialmente come ***strcpy()***, con la differenza che copia al massimo ***n*** caratteri, aggiungendo comunque il carattere di terminazione *<NUL>*.

Funzione	Descrizione
`char *strcat (char *restrict dst,` `const char *restrict org);` `char *strncat (char *restrict dst,` `const char *restrict org,` `size_t n);`	La funzione ***strcat()*** accoda alla stringa ***dst*** il contenuto della stringa ***org***, sovrascrivendo il carattere *<NUL>* che concludeva la prima stringa e aggiungendolo comunque alla fine della copia. Perché l'operazione possa avvenire è necessario che le due stringhe non si sovrappongano e, soprattutto, che ci sia abbastanza spazio disponibile dopo la prima stringa da estendere. La funzione restituisce il puntatore alla prima stringa. La funzione ***strncat()*** si comporta sostanzialmente come ***strcat()***, con la differenza che copia al massimo ***n*** caratteri dalla seconda stringa, aggiungendo comunque il carattere di terminazione *<NUL>*.

Funzione	Descrizione
`int strcmp (const char *str_1, const char *str_2);` `int strcoll (const char *str_1, const char *str_2);` `int strncmp (const char *str_1, const char *str_2, size_t n);`	La funzione ***strcmp()*** confronta due stringhe e restituisce zero nel caso siano uguali, oppure un valore minore di zero se la prima stringa è minore della seconda, oppure un valore maggiore di zero se la prima stringa è maggiore della seconda. La funzione ***strcoll()*** funziona sostanzialmente come ***strcmp()***, con la differenza che il confronto ha luogo tenendo conto della configurazione locale (precisamente la categoria '**`LC_COLLATE`**'). La funzione ***strncmp()*** si comporta sostanzialmente come ***strcmp()***, con la differenza che confronta al massimo ***n*** caratteri.

Funzione	Descrizione
`char *strchr (const char *str, int c);` `char *strrchr (const char *str, int c);`	La funzione ***strchr()*** cerca nella stringa ***str*** il carattere ***c*** (il carattere che si ottiene riducendo il valore di ***c*** a quello di un tipo '**`char`**'), includendo nella ricerca anche il carattere di terminazione *<NUL>*. La funzione restituisce un puntatore al carattere trovato, oppure restituisce il puntatore nullo se questo non c'è. La funzione ***strrchr()*** si comporta sostanzialmente come ***strchr()***, con la differenza che cerca l'ultima corrispondenza disponibile nella stringa.

Funzione	Descrizione
`char *strpbrk (const char *str_1, const char *str_2);`	La funzione ***strpbrk()*** cerca nella stringa ***str_1*** la prima corrispondenza con uno qualsiasi dei caratteri contenuti nella stringa ***str_2***. Restituisce il puntatore al carattere trovato nella stringa ***str_1*** che soddisfi la condizione; se non trova alcuna corrispondenza restituisce il puntatore nullo.
`size_t strspn (const char *str_1, const char *str_2);` `size_t strcspn (const char *str_1, const char *str_2);`	La funzione ***strspn()*** conta la lunghezza massima della sottostringa iniziale di ***str_1*** che contiene soltanto caratteri dell'insieme contenuto nella stringa ***str_2***. La funzione ***strcspn()*** svolge il compito opposto, di contare la lunghezza massima della sottostringa iniziale di ***str_2***, contenente solo caratteri che **non** fanno parte dell'insieme contenuto in ***str_2***.

Funzione	Descrizione
`size_t strlen (const char *str);`	La funzione ***strlen()*** restituisce la quantità di caratteri contenuta nella stringa, escluso il carattere di terminazione *<NUL>*.

All'inizio del capitolo, in occasione della descrizione delle costanti letterali per i tipi di dati primitivi, è già descritto il modo con cui si possono rappresentare alcuni caratteri speciali attraverso delle sequenze di escape che vengono annotate qui, nuovamente, per maggiore comodità del lettore, in quanto quelle sequenze sono valide anche nelle stringhe letterali.

Tabella 66.204. Elenco dei modi di rappresentazione delle costanti carattere attraverso codici di escape.

Codice di escape	Descrizione
`\`***ooo***	Notazione ottale.
`\x`***hh***	Notazione esadecimale.
`\\`	Una singola barra obliqua inversa ('****').
`\'`	Un apice singolo destro.
`\"`	Un apice doppio.
`\?`	Un punto interrogativo. Si usa in quanto le sequenze *trigraph* sono formate da un prefisso di due punti interrogativi.

Codice di escape	Descrizione
`\0`	Il codice <*NUL*>.
`\a`	Il codice <*BEL*> (*bell*).
`\b`	Il codice <*BS*> (*backspace*).
`\f`	Il codice <*FF*> (*formfeed*).
`\n`	Il codice <*LF*> (*linefeed*).
`\r`	Il codice <*CR*> (*carriage return*).
`\t`	Una tabulazione orizzontale (<*HT*>).
`\v`	Una tabulazione verticale (<*VT*>).

66.5.11 Parametri della funzione main()

«

La funzione ***main()***, se viene dichiarata con i suoi parametri tradizionali, permette di acquisire la riga di comando utilizzata per avviare il programma. La dichiarazione completa è la seguente:

```
int main (int argc, char *argv[])
{
  ...
}
```

Gli argomenti della riga di comando vengono convertiti in un array di stringhe (cioè di puntatori a '**`char`**'), in cui il primo elemento è il nome utilizzato per avviare il programma e gli elementi successivi

sono gli altri argomenti. Il primo parametro, ***argc***, serve a contenere la quantità di elementi del secondo, ***argv[]***, il quale è l'array di stringhe da scandire. È il caso di annotare che questo array dovrebbe avere sempre almeno un elemento: il nome utilizzato per avviare il programma e, di conseguenza, ***argc*** è sempre maggiore o uguale a uno.[21]

L'esempio seguente mostra in che modo gestire tale array, con la semplice riemissione degli argomenti attraverso lo standard output.

```
#include <stdio.h>

int main (int argc, char *argv[])
{
  int i;

  printf ("Il programma si chiama %s\n", argv[0]);

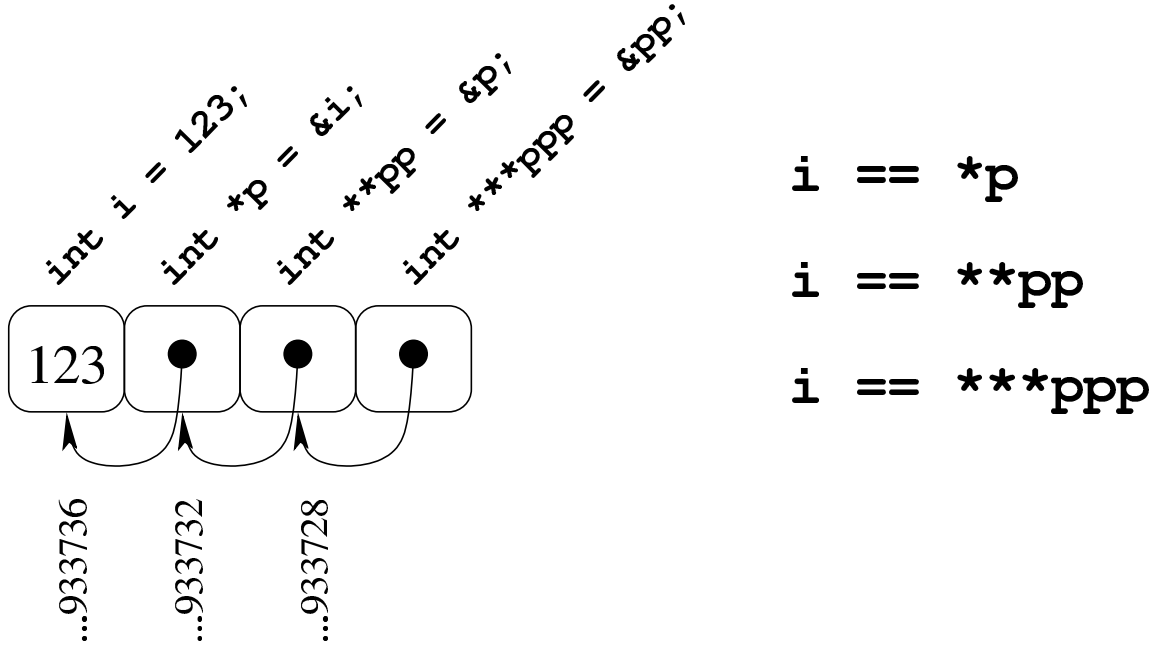
  for (i = 1; i < argc; i++)
    {
      printf ("argomento n. %i: %s\n", i, argv[i]);
    }
}
```

In alternativa, ma con lo stesso effetto, l'array di puntatori a stringhe può essere definito nel modo seguente, come puntatore di puntatori a caratteri:

```
int main (int argc, char **argv)
{
  ...
}
```

Figura 66.208. Schematizzazione di ciò che accade alla chiamata della funzione ***main()***, con un esempio.




```
int argc = 3;
char *argv[argc];
argv[0] = "/bin/ls";
argv[1] = "-l";
argv[2] = "/home/tizio";
main (argc, argv);
```

Chi è abituato a utilizzare linguaggi di programmazione più evoluti del C, può trovare strano che non si possa scrivere '**`main (int argc, char argv[][])`**' e usare di conseguenza l'array. Il motivo per cui ciò non è possibile dipende dal fatto che gli array a più dimensioni sono ottenuti attraverso sottoinsiemi uniformi del tipo dichiarato, così, in questo caso le stringhe dovrebbero essere della stessa dimensione, ma evidentemente ciò non corrisponde alla realtà. Inoltre, la dichiarazione della funzione dovrebbe contenere le dimensioni dell'array che non possono essere note. Pertanto, un array formato da stringhe diseguali, può essere ottenuto solo come array di puntatori al tipo '**`char`**'.

66.5.12 Puntatori a puntatori

Una variabile puntatore potrebbe fare riferimento a un'area di memoria contenente a sua volta un puntatore per un'altra area. Per dichiarare una cosa del genere, si possono usare più asterischi, come nell'esempio seguente:

```
int i = 123;
int *p = &i;          // Puntatore al tipo "int".
int **pp = &p;        // Puntatore di puntatore al tipo
                      // "int".
int ***ppp = &pp;     // Puntatore di puntatore di
                      // puntatore al tipo "int".
```

Il risultato si potrebbe rappresentare graficamente come nello schema seguente:




```
i == *p
i == **pp
i == ***ppp
```

Per dimostrare in pratica il funzionamento di questo meccanismo di riferimenti successivi, si può provare con il programma seguente.

Listato 66.211. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/cettwOsS* , *http://ideone.com/pzBoW* .

```
#include <stdio.h>
int
```

```
main (void)
{
  int i = 123;
  int *p = &i;         // Puntatore al tipo "int".
  int **pp = &p;       // Puntatore di puntatore al tipo
                       // "int".
  int ***ppp = &pp;    // Puntatore di puntatore di puntatore
                       // al tipo "int".

  printf ("i, p, pp, ppp: %i, %u, %u, %u\n",
          i, (unsigned int) p, (unsigned int) pp,
          (unsigned int) ppp);

  printf ("i, p, pp, *ppp: %i, %u, %u, %u\n",
          i, (unsigned int) p, (unsigned int) pp,
          (unsigned int) *ppp);

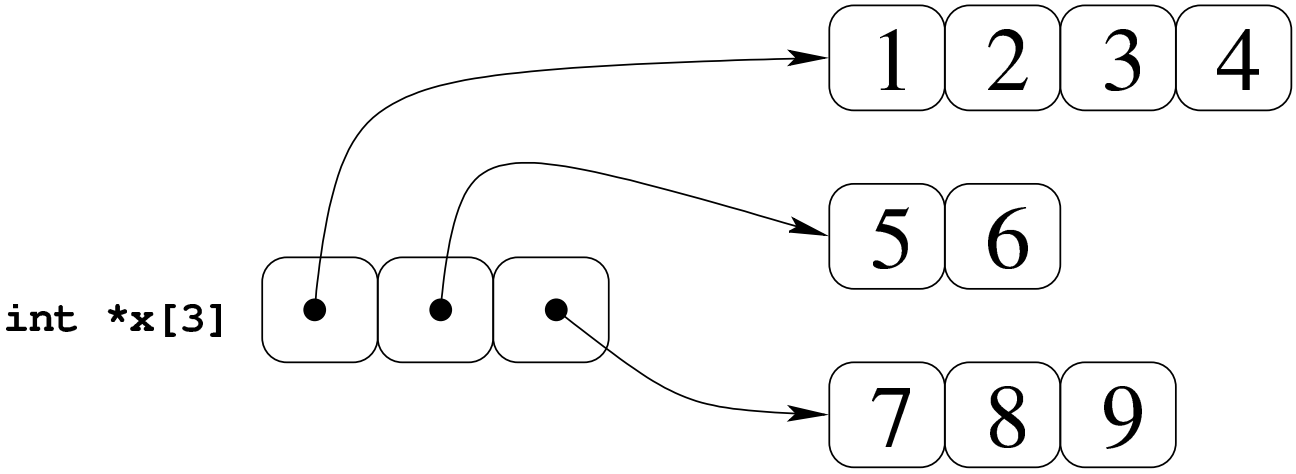
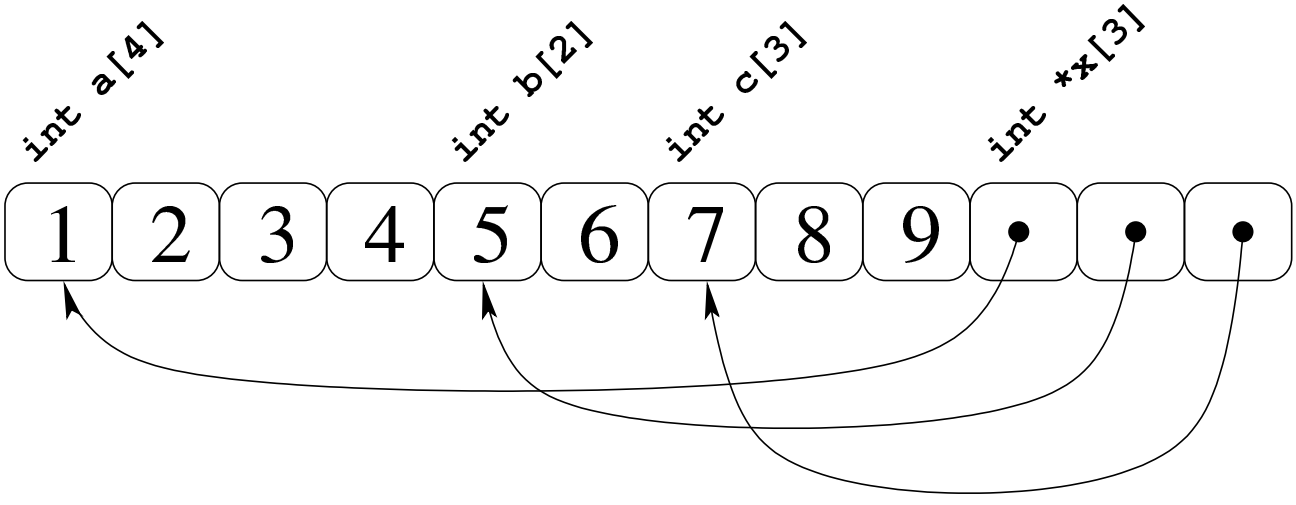
  printf ("i, p, *pp, **ppp: %i, %u, %u, %u\n",
          i, (unsigned int) p, (unsigned int) *pp,
          (unsigned int) **ppp);

  printf ("i, *p, **pp, ***ppp: %i, %i, %i, %i\n",
           i, *p, **pp, ***ppp);

  return 0;
}
```

Eseguendo il programma si dovrebbe ottenere un risultato simile a quello seguente, dove si può verificare l'effetto delle dereferenziazioni applicate alle variabili puntatore:

```
i, p, pp, ppp: 123, 3217933736, 3217933732, 3217933728
i, p, pp, *ppp: 123, 3217933736, 3217933732, 3217933732
i, p, *pp, **ppp: 123, 3217933736, 3217933736, 3217933736
```

```
i, *p, **pp, ***ppp: 123, 123, 123, 123
```

Pertanto si può ricostruire la disposizione in memoria delle variabili:



Come si può comprendere facilmente, la gestione di puntatori a puntatore è difficile e va usata con prudenza e solo quando ne esiste effettivamente l'utilità. Va notato anche che si ottiene la dereferenziazione (la traduzione di un puntatore nel contenuto di ciò a cui punta) usando la notazione tipica degli array, ma questo fatto viene descritto nella sezione successiva.

66.5.13 Puntatori a più dimensioni

Un array di puntatori consente di realizzare delle strutture di dati ad albero, non più uniformi come invece devono essere gli array a più dimensioni consueti. L'esempio seguente mostra la dichiarazione di tre array di interi, con una quantità di elementi disomogenea, e la successiva dichiarazione di un array di puntatori di tipo '**`int *`**', a cui si assegnano i riferimenti ai tre array precedenti. Nell'esempio

appare poi un tipo di notazione per accedere ai dati terminali che dovrebbe risultare intuitiva, ma se ne possono usare delle altre.

Listato 66.214. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/MPES5c6X* , *http://ideone.com/5bP39* .

```
#include <stdio.h>

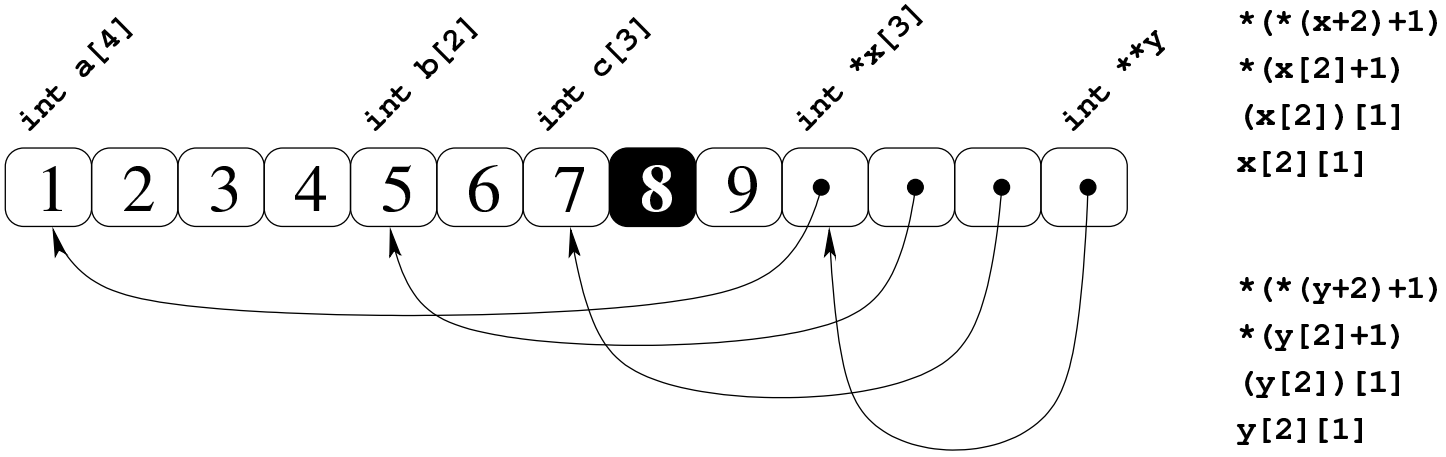
int main (void)
{
  int a[] = {1, 2, 3, 4};
  int b[] = {5, 6,};
  int c[] = {7, 8, 9};
  int *x[] = {a, b, c};

  printf ("*x[0] = {%i, %i, %i, %i}\n",
          *x[0], *(x[0]+1), *(x[0]+2), *(x[0]+3));
  printf ("*x[1] = {%i, %i}\n", *x[1], *(x[1]+1));
  printf ("*x[2] = {%i, %i, %i}\n",
          *x[2], *(x[2]+1), *(x[2]+2));

  return 0;
}
```

La figura successiva dovrebbe facilitare la comprensione del senso dell'array di puntatori. Come si può osservare, per accedere agli elementi degli array a cui puntano quelli di ***x*** è necessario dereferenziare gli elementi. Pertanto, '`*x[0]`' corrisponde al contenuto del primo elemento del primo sotto-array, '`*(x[0]+1)`' corrisponde al contenuto del secondo elemento del primo sotto-array e così di seguito. Dal momento che i sotto-array non hanno una quantità uniforme di elementi, non è semplice la loro scansione.

Figura 66.215. Schematizzazione semplificata del significato dell'array di puntatori definito nell'esempio.



Si potrebbe obbiettare che la scansione di questo array di puntatori a array può avvenire ugualmente in modo sequenziale, come se fosse un array «normale» a una sola dimensione. Molto probabilmente ciò è possibile effettivamente, dal momento che è probabile che il compilatore disponga le variabili in memoria in sequenza, come si vede nella figura successiva, ma ciò non può essere garantito.

Figura 66.216. La disposizione più probabile delle variabili dell'esempio.



Se invece di un array di puntatori si ha un puntatore di puntatori, il meccanismo per l'accesso agli elementi terminali è lo stesso. L'e-

sempio seguente contiene la dichiarazione di un puntatore a puntatori di tipo intero, a cui viene assegnato l'indirizzo dell'array già descritto. La scansione può avvenire nello stesso modo, ma ne viene proposto uno alternativo e più chiaro, con il quale si comprende cosa si intende per puntatore a più dimensioni.

Listato 66.217. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/kDRp85cQ85* , *http://ideone.com/JaqDH* .

```
#include <stdio.h>

int main (void)
{
  int a[] = {1, 2, 3, 4};
  int b[] = {5, 6,};
  int c[] = {7, 8, 9};
  int *x[] = {a, b, c};
  int **y = x;

  printf ("*x[0] = {%i, %i, %i, %i}\n", y[0][0], y[0][1],
                                        y[0][2], y[0][3]);
  printf ("*x[1] = {%i, %i}\n", y[1][0], y[1][1]);
  printf ("*x[2] = {%i, %i, %i}\n", y[2][0], y[2][1],
                                    y[2][2]);

  return 0;
}
```

Come si vede, la variabile ***y*** viene usata come se fosse un array a due dimensioni, ma lo stesso sarebbe valso per la variabile ***x***, in qualità di array di puntatori.

Per capire cosa succede, occorre fare mente locale al fatto che il

nome di una variabile puntatore seguito da un numero tra parentesi quadre corrisponde alla dereferenziazione dell'***n***-esimo elemento successivo alla posizione a cui punta tale variabile, mentre il valore puntato in sé corrisponde all'elemento zero (ciò è come dire che ***p*** equivale a '***p*`[0]`'). Quindi, scrivere '`*(`***p***`+`***n***`)`' è esattamente uguale a scrivere '***p*`[`***n***`]`'. Se il valore a cui punta una variabile puntatore è a sua volta un puntatore, per dereferenziarlo occorrono due fasi: per esempio ****p*** è il valore che si ottiene dereferenziando il primo puntatore e quello che si trova nella prima destinazione (quindi ****p*** equivale a '*****p*`[0]`' e a '***p*`[0][0]`'). Volendo gestire gli indici si possono considerare equivalenti i puntatori: '`*(*(p+`***m***`)+`***n***`)`', '`*(p[`***m***`]+`***n***`)`', '`(p[`***m***`])[`***n***`]`' e '`p[`***m***`][`***n***`]`'.

Figura 66.218. Tanti modi alternativi per raggiungere lo stesso elemento.



Seguendo lo stesso ragionamento si possono gestire strutture ad albero più complesse, con più livelli di puntatori, ma qui non vengono proposti esempi di questo tipo.

Sia l'array di puntatori, sia il puntatore a puntatori, possono essere gestiti con gli indici come se si trattasse di un array a più dimensioni. Pertanto, la notazione '***a*`[`***m***`][`***n***`]`' può rappresentare l'elemento ***m***,***n*** di un array ***a*** ottenuto secondo la rappresentazione «norma-

le» a matrice, oppure secondo uno schema ad albero attraverso dei puntatori: la differenza sta solo nella presenza o meno di elementi costituiti da puntatori.

66.5.14 Puntatori e funzioni

«

Nello standard del linguaggio C, la dichiarazione di una funzione è in pratica la definizione di un puntatore al codice della stessa, un po' come accade con gli array.[22] In generale, è possibile dichiarare dei puntatori a un tipo di funzione definito in base al valore restituito e ai tipi di parametri richiesti, attraverso una forma che richiama quella del prototipo di funzione. Il modello seguente è quello della dichiarazione del prototipo:

tipo ***nome_funzione*** **(** ***tipo_parametro*** **[** ***nome_parametro*** **]** **[,…]** **);**

Questo è invece il modello della dichiarazione del puntatore:

tipo **(*** ***nome_puntatore*** **)** **(** ***tipo_parametro*** **[** ***nome_parametro*** **]** **[,…]** **);**

L'esempio seguente mostra la dichiarazione di un puntatore a una funzione che restituisce un valore di tipo '**int**' e utilizza due parametri di tipo '**int**':

```
int (*f) (int, int);
```

L'esempio seguente è equivalente, con la differenza che si nominano i parametri, anche se ciò è perfettamente inutile, esattamente come nei prototipi delle funzioni:

```
int (*f) (int i, int j);
```

L'assegnamento del puntatore avviene nel modo più semplice possibile, trattando il nome della funzione nello stesso modo in cui si fa con gli array: come un puntatore.

```
int (*f) (int, int);      // Puntatore a funzione.
int prodotto (int, int); // Prototipo di funzione descritta
                          // più avanti.
…
f = prodotto; // Il puntatore «f» contiene il riferimento
              // alla funzione.
```

Una volta assegnato il puntatore, si può eseguire una chiamata di funzione semplicemente utilizzando il puntatore, per cui, i due esempi seguenti sono equivalenti:

```
i = f (2, 3);
```

```
i = prodotto (2, 3);
```

Nel linguaggio C precedente allo standard ANSI, perché il puntatore potesse essere utilizzato in una chiamata di funzione, occorreva indicare l'asterisco, in modo da dereferenziarlo:

```
i = (*f) (2, 3);          // Non serve più.
```

Per concludere viene mostrato un esempio completo, anche se banalizzato: la funzione *f()* restituisce un numero intero ottenuto incrementando di una unità l'argomento ricevuto. Questa funzione viene chiamata attraverso un puntatore denominato ***pf*** .

Listato 66.225. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/rbXNWbOh* , *http://ideone.com/L6ooG* .

```
#include <stdio.h>

int f (int i)
{
    return (i + 1);
}

int main (void)
{
  int x = 4;
  int y;
  int (*pf) (int i);
  pf = f;
  y = pf (x);
  printf ("%i + 1 = %i\n", x, y);
  return 0;
}
```

Riquadro 66.226. Confusione tra le dichiarazioni.

L'interpretazione umana del linguaggio, a proposito dei puntatori, può essere complicata, pertanto l'uso dei puntatori deve essere fatto con criterio, senza abusarne. Gli esempi seguenti sono solo i più semplici:
`int f (...);` /* dichiarazione della funzione ***f()*** che restituisce un valore intero; */
`int *f (...);` /* dichiarazione della funzione ***f()*** che restituisce un puntatore a un intero; */
`int (*f) (...);` /* dichiarazione del puntatore ***f*** a una funzione che restituisce un intero; */
`int *(*f) (...);` /* dichiarazione del puntatore ***f*** a una funzione che restituisce un puntatore a un intero. */
Ancora più difficile sarebbe dichiarare una funzione che restituisce un array, o peggio, un puntatore a un array.

66.5.14.1 Puntatori a funzione, membri di una struttura

«

Le strutture sono descritte in un'altra sezione (66.7), tuttavia è opportuno annotare qui in che modo possa essere utilizzato un puntatore a una funzione, quando è un membro di una struttura:

```
struttura . membro (argomenti);
```

```
(*struttura . membro) (argomenti);
```

I due modelli sono equivalenti e si riferiscono alla chiamata di una funzione, il cui puntatore è costituito dalla variabile ***struttura.membro***. È evidente che risulta più comprensibile la prima delle due modalità. A titolo di esempio, ipotizzando la struttura

totale e il membro ***sottrai***, per una funzione che riceve un argomento di tipo intero (precisamente il numero 7), la chiamata potrebbe essere scritta indifferentemente nei due modi successivi:

```
...
totale.sottrai (7);
...
```

```
...
(*totale.sottrai) (7);
...
```

66.5.15 Puntatori a variabili distrutte

«

L'esempio seguente potrebbe funzionare, ma contiene un errore di principio.

Listato 66.229. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/DRtSxmiS* , *http://ideone.com/egses* .

```
#include <stdio.h>

double *f (void)
{
  double x = 1234.5678;
  return &x;                    // Orrore!
}

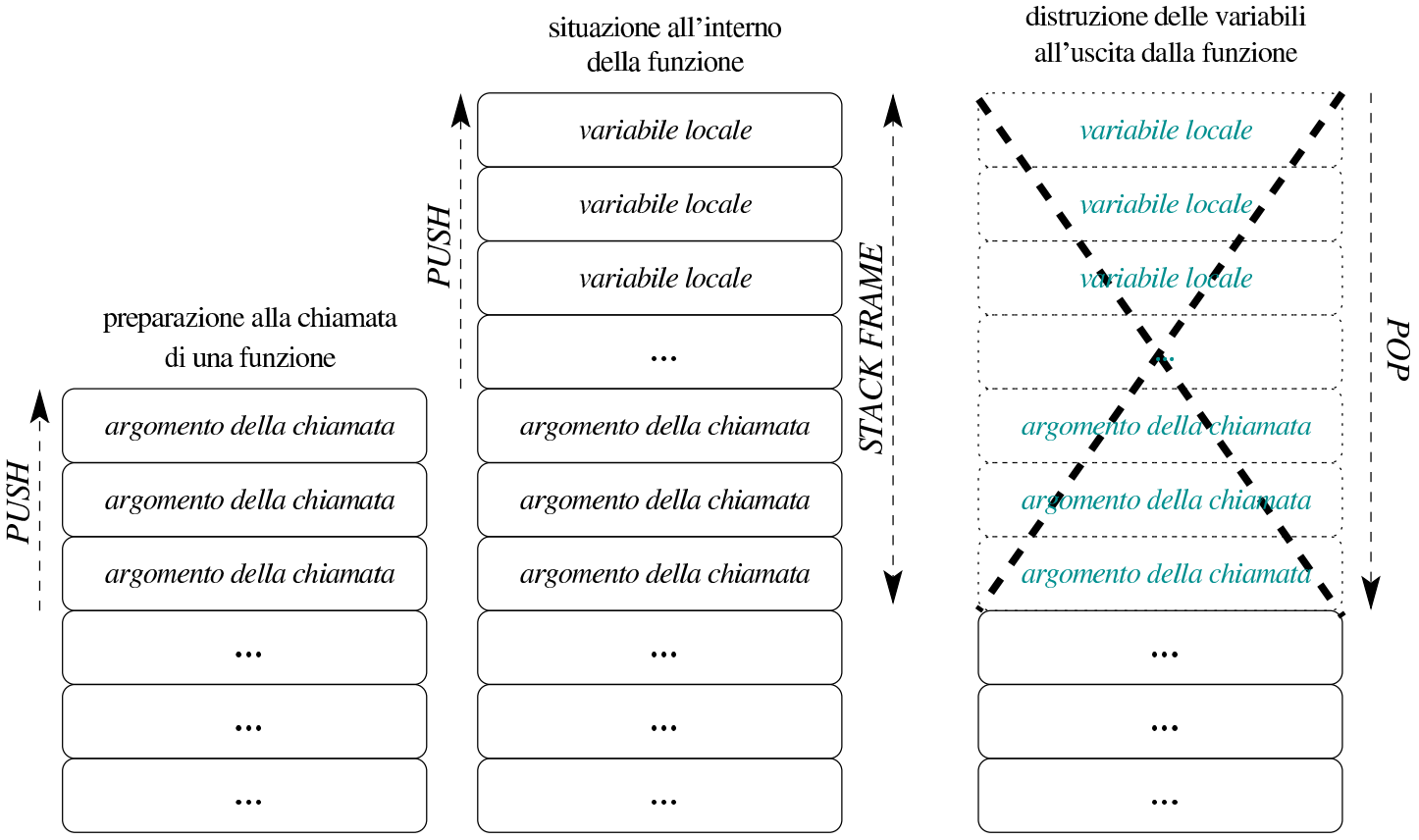
int main (int argc, char *argv[])
{
  double *p;
  p = f ();
  printf ("x = %f\n", *p);
  return 0;
```

```
}
```

La funzione ***f()*** dichiara localmente una variabile che inizializza al valore 1234,5678, quindi restituisce il puntatore a questa variabile. A parte il fatto che il compilatore possa segnalare o meno la cosa, non si può utilizzare un puntatore rivolto a un'area di memoria che, almeno teoricamente, non è più allocata. In altri termini, se si costruisce un puntatore a qualcosa, occorre tenere sempre presente il ciclo di vita della sua destinazione e non solo della variabile che contiene tale riferimento.

Purtroppo questa attenzione non viene imposta e, generalmente, il compilatore consente di usare un puntatore a variabili che, formalmente, sono già state distrutte.

66.5.16 Puntatore nullo

Il linguaggio C prescrive che si possa assegnare a una variabile puntatore il valore zero, in qualità di numero intero:

```
…
double *p = 0;
…
```

Il puntatore che contiene il valore zero è indefinito, nel senso che punta a un'area di memoria irraggiungibile. Un puntatore di questo tipo è noto come ***puntatore nullo*** o *null pointer*; inoltre, due puntatori nulli, qualunque sia il tipo di dati a cui si riferiscono, sono uguali in una comparazione. Pertanto si potrebbe verificare la validità di un puntatore nel modo seguente:

```
…
char *p = 0;
…
if (p == 0)
  {
    // Null pointer.
    …
  }
…
```

A ogni modo, lo standard prescrive che nel file ‘`stddef.h`’ sia definita la macro-variabile ***NULL***, a rappresentare formalmente un puntatore nullo:

```
#include <stddef.h>
…
char *p = NULL;
…
if (p == NULL)
  {
    // Null pointer.
    …
  }
…
```

Va osservato che la variabili puntatore, quando acquisiscono un indirizzo in base al verificarsi di certe condizioni, vanno inizializzate opportunamente al valore nullo (come già apparso negli esempi), in modo da poter poi verificare se hanno ottenuto o meno un tale indirizzo.

66.5.17 Utilizzo della memoria in modo dinamico

L'allocazione dinamica della memoria avviene generalmente attraverso la funzione ***malloc()***, oppure ***calloc()***, definite nella libreria standard, secondo i prototipi contenuti nel file '`stdlib.h`'. Se queste riescono a eseguire l'operazione, restituiscono il puntatore alla memoria allocata, altrimenti restituiscono il valore '**`NULL`**'.

```
void *malloc (size_t dimensione);
```

```
void *calloc (size_t quantità, size_t dimensione);
```

La differenza tra le due funzioni sta nel fatto che la prima, ***malloc()***, viene utilizzata per allocare un'area di una certa dimensione, espressa generalmente in byte, mentre la seconda, ***calloc()***, permette di indicare una quantità di elementi e si presta per l'allocazione di array.

Dovendo utilizzare queste funzioni per allocare della memoria, è necessario conoscere la dimensione dei tipi primitivi di dati, ma per evitare incompatibilità conviene farsi aiutare dall'operatore '**`sizeof`**'.

Il valore restituito da queste funzioni è di tipo '**`void *`**' cioè una specie di puntatore neutro, indipendente dal tipo di dati da utilizzare (in quanto il tipo '**`void`**', in sé, rappresenta una variabile di rango nullo, la quale non può contenere alcun dato). Per questo, in linea di principio, prima di assegnare a un puntatore il risultato dell'esecuzione di queste funzioni di allocazione, è opportuno eseguire un cast.

```
int *pi = NULL;
…
pi = (int *) malloc (sizeof (int));

if (pi != NULL)
  {
    // Il puntatore è valido e allora procede.
    …
  }
else
  {
    // La memoria non è stata allocata e si fa qualcosa
    // di alternativo.
    …
  }
```

Come si può osservare dall'esempio, il cast viene eseguito con la notazione '**`(int *)`**' che richiede la conversione esplicita in un puntatore a '**`int`**'. Lo standard C non richiede l'utilizzo di questo cast, quindi l'esempio si può ridurre al modo seguente:

```
…
pi = malloc (sizeof (int));
…
```

La memoria allocata dinamicamente deve essere liberata in modo esplicito quando non serve più. Infatti, il linguaggio C non offre alcun meccanismo di ***raccolta della spazzatura*** o *garbage collector*. Per questo si utilizza la funzione ***free()*** che richiede semplicemente il puntatore e non restituisce alcunché.

```
void free (void *puntatore);
```

È necessario evitare di deallocare più di una volta la stessa area di memoria, perché ciò potrebbe provocare effetti imprevedibili.

```
int *pi = NULL;
…
pi = (int *) malloc (sizeof (int));

if (pi != NULL)
  {
    // Il puntatore è valido e allora procede.
    …
    free (pi); // Libera la memoria
    pi = NULL; // e per sicurezza azzera il puntatore.
    …
  }
else
  {
    // La memoria non è stata allocata e si fa qualcosa
    // di alternativo.
    …
  }
```

Lo standard prevede una funzione ulteriore, per la riallocazione di memoria: ***realloc()***. Questa funzione si usa per ridefinire l'area di memoria con una dimensione differente:

```
void *realloc (void *puntatore, size_t dimensione);
```

In pratica, la riallocazione deve rendere disponibili gli stessi contenuti già utilizzati, salvo la possibilità che questi siano stati ridotti nella parte terminale. Se invece la dimensione richiesta nella rial-

locazione è maggiore di quella precedente, lo spazio aggiunto può contenere dati casuali. Va osservato che la collocazione in memoria, successiva alla riallocazione, può essere differente da quella precedente. Il funzionamento di ***realloc()*** non è garantito, pertanto occorre verificare nuovamente, dopo il suo utilizzo, che il puntatore ottenuto sia ancora valido.

66.5.18 Puntatori «ristretti»

Lo standard del linguaggio C prevede il modificatore '**`restrict`**' per le variabili puntatore, da usare come nell'esempio seguente:

```
…
int *restrict p;
…
```

L'utilizzo di tale modificatore equivale a una dichiarazione di intenti (ovvero una promessa) che il programmatore fa al compilatore, nei riguardi del puntatore. Precisamente si dichiara che il puntatore viene usato per accedere ad aree di memoria in modo esclusivo, nel senso che nell'ambito del contesto a cui si fa riferimento, non esistono altri accessi alle stesse aree per mezzo di altri puntatori o di altre variabili. Partendo da questo presupposto, il compilatore può ottimizzare il risultato della compilazione semplificando il codice finale.

La definizione formale del significato di questo modificatore è molto complessa e il compilatore non è in grado di segnalarne un uso improprio. Ciò significa che va usata questa possibilità con prudenza, solo quando si ritiene di averne capito il senso e l'utilità.

Come esempio iniziale si può osservare il prototipo della funzione standard ***strcpy()***:

```
char *strcpy (char *restrict dst, const char *restrict org);
```

Ci sono due parametri costituiti da stringhe che non devono risultare sovrapposte e in questo caso, il vincolo '**restrict**' è appropriato per esprimere il concetto: se entrambi i puntatori delle stringhe sono dichiarati con il modificatore '**restrict**', è evidente che le stringhe rispettive non devono sovrapporsi.

L'impegno che il programmatore prende utilizzando il modificatore '**restrict**' è finalizzato solo al favorire l'ottimizzazione della compilazione.

La promessa che un programmatore fa dichiarando un puntatore '**restrict**' è limitata al campo di azione del puntatore stesso. Per esempio, tornando all'esempio del prototipo della funzione ***strcpy()***, lì si intende che i parametri vengono usati nella funzione senza sovrapposizioni, ma, dato il contesto, rimane il fatto che le stringhe fornite come argomento della chiamata debbano già rispettare il vincolo di non essere sovrapposte.

Esempio 66.237. Viene allocata un'area di memoria composta da 100 elementi della grandezza di un intero normale. I primi 50 elementi vengono scanditi con il puntatore ***r1*** mentre quelli restanti con il puntatore ***r2***. Nell'esempio, agli elementi '***r1*[*i*]**' viene assegnato il valore di '***r2*[*i*]+1**', anche se il fatto in sé non ha una grande importanza.

```
int *restrict r1, *restrict r2;
int *m = malloc (100 * sizeof (int));
int i;
```

```
r1 = m;       // r1 viene usato per i primi 50 elementi.
r2 = m + 50; // r2 viene usato per i 50 elementi successivi.

for (i = 0; i < 50; i++)
  {
    r1[i] = r2[i] + 1;
  }
```

Esempio 66.238. Viene allocata un'area di memoria composta da 100 elementi della grandezza di un intero normale. Gli elementi pari vengono scanditi con il puntatore ***r1*** mentre quelli dispari con il puntatore ***r2***. Nell'esempio, agli elementi '***r1*`[j]`' viene assegnato il valore di '***r2*`[j]+1`', anche se il fatto in sé non ha una grande importanza.

```
int *restrict r1, *restrict r2;
int *m = malloc (100 * sizeof (int));
int i;
int j;

r1 = m;          // r1 viene usato per gli elementi con
                 // indice pari.
r2 = m + 1;      // r2 viene usato per gli elementi con
                 // indice dispari.

for (i = 0; i < 50; i++)
  {
    j = i * 2;
    r1[j] = r2[j] + 1;
  }
```

Se il compilatore non riconosce il modificatore '**`restrict`**' significa solo che non è in grado di ottimizzare il codice in un certo modo,

ma non è necessario modificare il proprio programma per togliere la parola chiave relativa, perché è sufficiente sfruttare una macro-variabile del precompilatore, a cui non si assegna alcun valore:

```
…
#define restrict
…
```

Oppure, se ciò non è possibile, la si dichiara come un commento privo di contenuto:

```
…
#define restrict /**/
…
```

66.6 Le funzioni

Per comprendere come «funzionano» le funzioni nel linguaggio C, occorre fare mente locale all'uso della pila dei dati con il linguaggio macchina. Qui si chiariscono alcuni concetti, partendo dal ripasso della pila dei dati.

66.6.1 Pila dei dati

Dal punto di vista del linguaggio macchina, generalmente si dispone di una pila di dati che si sviluppa a partire da un certo indirizzo di memoria, utilizzando di volta in volta indirizzi inferiori della stessa. Attraverso la pila dei dati, prima della chiamata di una funzione, gli argomenti vengono passati alla stessa aggiungendoli alla pila; successivamente, all'interno della funzione, tutte le variabili locali vengono ottenute facendo crescere ulteriormente la pila. Al termine dell'esecuzione della funzione, la pila viene ridotta allo stato prece-

dente alla chiamata, espellendo le variabili locali e i parametri della chiamata.

Figura 66.241. Semplificazione del meccanismo attraverso cui si passano gli argomenti a una funzione e si gestiscono le variabili locali.



Naturalmente, dal momento che la pila di dati viene gestita attraverso la memoria centrale, la quale consente un accesso diretto ai dati, tramite un indirizzo, nella pila si possono gestire dati di tutti i tipi, volendo anche degli array. A proposito degli array, quando questi sono creati all'interno delle funzioni, pertanto attraverso l'uso della pila dei dati, al compilatore non è necessario sapere preventivamente le dimensioni di questi, perché lo spazio che usano nella memoria è allocato dinamicamente, tramite la pila.

66.6.2 Dichiarazione e chiamata di una funzione

La dichiarazione di una funzione prevede l'indicazione del tipo di variabili che compongono i parametri, allo scopo di far sapere al compilatore in che modo inserire gli argomenti nella pila, al momento della chiamata. Si osservi l'esempio seguente in cui si dichiara una funzione con due parametri molto semplici: un intero normale e un intero di dimensione «doppia».

```
void f (int x, long long int y)
{
     …
     …
}
```

Partendo dal presupposto che la pila dei dati sia gestita a blocchi di «parole» del microprocessore, si può ipotizzare ragionevolmente in che modo siano impilati gli argomenti della chiamata. Si suppone di chiamare la funzione nel modo seguente e che la parola sia da 32 bit:

```
…
f (0x13579BDF, 0x123456789ABCDEF);
…
```

Alla chiamata della funzione, i parametri dovrebbero apparire nella pila come nella figura successiva, trascurando il problema dell'inversione eventuale dei byte:

Come si vede, gli argomenti vengono impilati in ordine inverso, in modo tale che il primo argomento appaia all'inizio della pila.

Ci sono molti dettagli da definire sul come vadano impilati gli argomenti di una chiamata; in particolare è da chiarire in che modo vadano trattati i dati la cui dimensione è inferiore alla parola del microprocessore, così come per quelli che si articolano in strutture. Questi dettagli vanno chiariti quando si vogliono scrivere funzioni da usare assieme a codice scritto in linguaggio assemblatore, oppure anche per altri linguaggi, se per quelli si utilizzano compilatori non conformi a quello usato per il C.

66.6.3 Elenco indefinito di parametri

«

Il linguaggio C ammette che le funzioni siano dichiarate con almeno un parametro esplicito e un elenco indefinito di parametri successivi. In altre parole, si ammette che ci sia un parametro certo e un elenco, eventuale, di altri parametri sconosciuti. Questo avviene, per esempio, con funzioni standard quali ***printf()***:

```
int printf (const char *formato, ...);
```

Quando si chiama una funzione del genere, gli argomenti successivi al primo, se riguardano valori numerici, vengono «promossi» in modo tale da avere una dimensione minima di riferimento. Per la precisione, i valori interi di rango inferiore a quello di un intero comune, sono convertiti al livello di intero '**int**' (con segno o senza, in base alle caratteristiche di partenza); i valori in virgola mobile, se sono espressi secondo un formato di rango inferiore a '**double**', vengono trasformati semplicemente in '**double**'. Gli interi e i valori in virgola mobile di rango superiore, rimangono invariati.

È da osservare che, se si tenta di passare come argomento un valore che occupa uno spazio inferiore alla dimensione della parola del microprocessore, pur dichiarando tutti i parametri è molto probabile che il compilatore debba utilizzare ugualmente una parola intera, riempiendo in qualche modo lo spazio restante con dati nulli; pertanto, in presenza di parametri di dimensione non stabilita, è più che appropriata la promozione predefinita degli argomenti a valori multipli della parola.

Viene mostrato un esempio di programma contenente una funzione con un numero indefinito di parametri, nella quale, gli argomenti della chiamata vengono comunque estratti dalla pila dei dati, conoscendo le dimensioni usate nella chiamata. L'esempio funziona con un compilatore GNU C e serve solo per comprendere il meccanismo, ma per il momento non rappresenta il modo corretto di agire a

questo proposito.

Listato 66.245. Per provare il codice attraverso un servizio *pastebin*: http://ideone.com/JzZOP .

```
#include <stdio.h>

void f (int w,...)
{
  //
  // Traduce l'indirizzo di «w» nel puntatore «p».
  //
  char *p = (char *) &w;
  //
  // Sposta il puntatore all'inizio del secondo parametro.
  //
  p = p + sizeof w;
  //
  // Mostra il valore del primo e del secondo parametro.
  //
  printf ("w = %i; ", w);
  printf ("x = %Lf; ", *((long double *)p));
  //
  // Sposta il puntatore all'inizio del terzo parametro.
  //
  p = p + sizeof (long double);
  //
  // Mostra il terzo parametro.
  //
  printf ("y = %lli; ", *((long long int *)p));
  //
  // Sposta il puntatore all'inizio del quarto parametro.
  //
  p = p + sizeof (long long int);
  //
```

```
  // Mostra il quarto parametro.
  //
  printf ("z = %i\n", *((int *)p));
  //
  return;
}

int main (int argc, char *argv[])
{
  f (10, (long double) 12.34, (long long int) 13, 14);
  return 0;
}
```

Come si vede, per raggiungere gli argomenti successivi al primo, conoscendo le loro caratteristiche, si scandisce in pratica la memoria occupata dalla pila dei dati, prendendo come riferimento l'indirizzo del primo parametro, il quale costituisce il riferimento certo. Si misura la dimensione del primo parametro e si aggiusta il puntatore in modo da posizionarsi dopo la fine di questo, sapendo che da lì in poi si trovano gli argomenti successivi. Il puntatore è di tipo '**`char *`**', in modo da poterlo gestire a unità di «caratteri», conformemente al valore prodotto dall'operatore '**`sizeof`**'. Se tutto funziona come previsto, il programma mostra correttamente il messaggio seguente:

```
w = 10; x = 12.340000; y = 13; z = 14
```

Il modo corretto di estrapolare i valori dei parametri non dichiarati richiede l'uso di alcune macroistruzioni della libreria standard, contenute nel file di intestazione '`stdarg.h`'. Si osservi come va trasformato l'esempio già apparso per rispettare la formalità standard:

Listato 66.247. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/22g6f9rU* , *http://ideone.com/Xuy6s* .

```
#include <stdio.h>
#include <stdarg.h>

void f (int w,...)
{
  //
  // Dichiara le variabili che servono a contenere
  // gli argomenti privi di parametri formali.
  //
  long double x;
  long long int y;
  int z;
  //
  // Dichiara il puntatore ai parametri.
  //
  va_list ap;
  //
  // Posiziona il puntatore dopo il primo parametro,
  // ovvero dopo l'ultimo parametro dichiarato
  // esplicitamente.
  //
  va_start (ap, w);
  //
  // Estrapola il secondo argomento della chiamata (portando
  // avanti il puntatore di conseguenza.
  //
  x = va_arg (ap, long double);
  //
  // Mostra il valore del primo e del secondo argomento
  // ottenuto dalla chiamata della funzione.
```

```
  //
  printf ("w = %i; ", w);
  printf ("x = %Lf; ", x);
  //
  // Estrapola il terzo argomento.
  //
  y = va_arg (ap, long long int);
  //
  // Mostra il terzo argomento.
  //
  printf ("y = %lli; ", y);
  //
  // Estrapola il quarto argomento.
  //
  z = va_arg (ap, int);
  //
  // Mostra il quarto e ultimo argomento.
  //
  printf ("z = %i\n", z);
  //
  // Conclude la scansione degli argomenti.
  //
  va_end (ap);
  //
  return;
}

int main (int argc, char *argv[])
{
  f (10, (long double) 12.34, (long long int) 13, 14);
  return 0;
}
```

Come si vede, è necessario incorporare il file di intestazione

‘stdarg.h’ della libreria standard. All’inizio della funzione si dichiara una variabile di tipo ‘**va_list**’ per scandire l’elenco di parametri: si tratta evidentemente di un puntatore (molto probabilmente al tipo ‘**char**’). Subito dopo si inizializza la variabile da usare per la scansione con la macroistruzione ‘**va_start**’ che ha l’apparenza di una funzione. A ‘**va_start**’ viene passata la variabile da usare come puntatore per gli argomenti e l’ultimo parametro dichiarato espressamente nella funzione, allo scopo di aggiornare il puntatore e di portarlo all’inizio del primo argomento privo di un parametro esplicito. Successivamente si utilizza la macroistruzione ‘**va_arg**’, anche questa con l’apparenza di una funzione, per estrapolare l’argomento a cui punta la variabile di tipo ‘**va_list**’, usata per lo scopo, aggiornando conseguentemente la variabile-puntatore, in modo da essere pronta per l’argomento successivo. Al termine si usa ‘**va_end**’, la quale può essere indifferentemente una macroistruzione o una funzione vera e propria, allo scopo di concludere l’uso del puntatore dichiarato per la scansione dei parametri.

Le macroistruzioni ‘**va_start**’ e ‘**va_arg**’ non potrebbero essere realizzate in forma di funzioni. Infatti, ‘**va_start**’ utilizza apparentemente come argomento l’ultimo parametro della funzione, ma per calcolare la posizione del parametro successivo servirebbe invece l’indirizzo di tale variabile. In modo analogo, la macroistruzione ‘**va_arg**’ richiede l’indicazione del tipo di dati da estrarre, mentre una funzione vera potrebbe accettare solo la dimensione restituita dall’operatore ‘**sizeof**’; inoltre restituisce un valore dello stesso tipo, mentre una funzione vera può restituire un solo tipo prestabilito.

Nell'esempio non si vede cosa accade quando si trasmette un argomento costituito da un carattere ('**char**'). In tal caso bisogna tenere in considerazione l'effetto della promozione a intero; pertanto, la macroistruzione ***va_arg*** va usata indicato un tipo '**int**' (e non un tipo '**char**'). Lo stesso dicasi per i valori in virgola mobile, che vanno estratti prevedendo un formato '**double**', anche se nell'argomento originale dovesse trattarsi di '**float**' (e ammesso che l'argomento non sia espresso in un formato ancora più grande).

66.6.4 Annotazioni su «printf()» e altre funzioni simili

Da quanto descritto a proposito della promozione dei valori numerici, interi o in virgola mobile, si comprende che le rappresentazioni di valori numerici vanno fatte preferibilmente a partire da interi di tipo '**int**' o da valori in virgola mobile di tipo '**double**'. Si osservino gli esempi seguenti:

- ```
  printf ("%hd\n", 123);
  ```
  in linea di principio, lo specificatore di conversione '**%hd**' attende un valore di tipo '**short int**', ma il valore 123 che gli viene fornito è implicitamente di tipo '**int**';

- ```
  printf ("%hd\n", (short int) 123);
  ```
  esattamente come nell'esempio precedente e a nulla serve il tentativo di indicare un cast nell'argomento della chiamata alla funzione;

- ```
  printf ("%c\n", 'A');
  ```
  lo specificatore di conversione '**%c**' si attende un valore di tipo '**char**' (con o senza segno), ma il carattere '**A**' che gli viene fornito è implicitamente di tipo '**int**'.
  ```

Nel caso della funzione ***scanf()***, questi problemi non ci sono, perché gli argomenti variabili sono costituiti tutti da puntatori ad aree di memoria che devono essere in grado di contenere le informazioni da inserire.

66.6.5 Costante predefinita «__func__»

«

Lo standard del linguaggio prescrive che, se all'interno di una funzione viene usato il nome '**__func__**', questo si deve tradurre nel nome della funzione che lo contiene. In pratica, il compilatore che incontra questo nome, dichiara automaticamente, all'interno della funzione, la costante seguente:

```
static const char __func__[] = "nome_funzione";
```

L'esempio seguente mostra in che modo se ne potrebbe fare uso.

Listato 66.251. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/1I0KIqiq* , *http://ideone.com/A9est* .

```
#include <stdio.h>
void f (void)
{
  printf ("Sono nella funzione \"%s\".\n", __func__);
}
int main (int argc, char *argv[])
{
  f ();
  return 0;
}
```

Una volta compilato il programma, eseguendolo si ottiene:

```
Sono nella funzione "f".
```

66.7 Struttura, unione, campo, enumerazione, costante composta

Fino a questo punto sono stati incontrati solo i tipi di dati primitivi, oltre agli array di questi (incluse le stringhe). Nel linguaggio C, come in altri, è possibile definire dei tipi di dati aggiuntivi, derivati dai tipi primitivi.

Qui si usa la convenzione di nominare le strutture, le unioni e le enumerazioni con una lettera iniziale maiuscola. Per quanto riguarda invece i tipi di dati derivati, ottenuti con l'istruzione '**`typedef`**', si segue l'uso comune di aggiungere l'estensione '**`_t`**'.

66.7.1 Enumerazioni

È possibile dichiarare una variabile di tipo enumerativo, costituita tecnicamente da un intero, la quale può rappresentare solo un insieme prestabilito di valori, indicati simbolicamente attraverso delle definizioni. I valori simbolici che possono essere rappresentati sono tradotti in un numero intero, ma il programmatore non dovrebbe avere la necessità di avere a che fare direttamente con tali valori numerici corrispondenti. In altri termini, il tipo enumerativo è una forma di rappresentazione di un intero attraverso costanti mnemoniche.

```
enum nome { costante [, costante] ...}
```

La sintassi indicata mostra il modo in cui si definisce un tipo del genere: all'interno di parentesi graffe si elencano i nomi delle costanti

che possono essere assegnate a una variabile di questo tipo. Tuttavia, alle costanti si può associare un valore intero in modo esplicito; pertanto, la costante può essere espressa così:

nome_simbolico [=***n***]

Si osservi l'esempio seguente che comunque non rappresenta un programma completo:

```
…
enum Colore { nero, marrone, rosso, arancio, giallo, verde,
              blu, viola, grigio, bianco, argento=100,
              oro };
…
enum Colore c;     // Dichiara la variabile «c».
…
c = marrone + 1;   // Assegna a «c» il valore successivo a
                   // «marrone»; in pratica assegna il valore
                   // «rosso».
…
if (c <= rosso);                          // Se il colore va dal
  {                                       // nero al rosso,
    printf ("Non mi piace: %i\n", c); // visualizza un
  }                                       // messaggio e mostra
                                          // anche il numero
                                          // corrispondente.
…
```

All'inizio viene dichiarato il tipo enumerativo '**Colore**', come insieme di colori principali, definiti simbolicamente per nome. Va osservato che nel caso dell'argento, viene associato espressamente il valore 100.

In mancanza di associazioni esplicite tra il valore simbolico e valore numerico, il compilatore associa al primo dei simboli il valore zero e dà a quelli successivi un numero ottenuto incrementando di una unità quello precedente. Nel caso dell'esempio, nero corrisponde a zero, marrone a uno, rosso a due e così di seguito fino al bianco. Il colore argento è definito espressamente (quindi dal nove del bianco si salta al 100 dell'argento) e il colore dell'oro viene determinato implicitamente come pari a ***argento***+1, ovvero uguale a 101.

Seguendo l'esempio si vede la dichiarazione della variabile ***c*** di tipo '**`enum Colore`**'. In pratica, viene dichiarata una variabile di tipo intero, in grado di contenere i valori dell'enumerazione '**`Colore`**'.

Successivamente si assegna alla variabile ***c*** la somma tra la costante ***marrone*** (pari a uno) e il numero uno. In pratica si assegna il valore due, ma in base al contesto si intende di avere assegnato ***rosso***.

Alla fine dell'esempio si vede un confronto tra la variabile ***c*** e un colore di quelli definiti simbolicamente. Di fatto si sta confrontando il valore della variabile con il numero due, ma in pratica sembra di valutare la cosa solo sul piano della sequenza ideale che è stata attribuita a quei colori.

La dichiarazione di una variabile enumerativa coincide quindi con la dichiarazione di un insieme di costanti simboliche, le quali non possono essere ridefinite. Pertanto, non è possibile dichiarare due variabili diverse che condividono costanti simboliche con lo stesso nome, a meno di essere in un campo di azione differente:

```
enum Colori { nero, marrone, rosso, arancio, giallo, verde,
              blu, viola, grigio, bianco };

enum Bianco_e_nero { nero, bianco };    // Non si può.
```

Le costanti simboliche definite attraverso le enumerazioni, possono essere usate anche al di fuori delle variabili dichiarate espressamente per questo scopo, purché possano ragionevolmente contenerne il valore. È anche evidente che al posto delle enumerazioni definite in questo modo sia possibile gestire direttamente le costanti. L'esempio seguente riporta i passi equivalenti di quanto già visto all'inizio della sezione:

```
…
const int nero    = 0;
const int marrone = 1;
const int rosso   = 2;
const int arancio = 3;
const int giallo  = 4;
const int verde   = 5;
const int blu     = 6;
const int viola   = 7;
const int grigio  = 8;
const int bianco  = 9;
const int argento = 100;
const int oro     = 101;
…
int c;              // Dichiara la variabile «c».
…
c = marrone + 1;    // Assegna a «c» il valore successivo a
                    // «marrone»; in pratica assegna il valore
                    // «rosso».
…
if (c <= rosso);                        // Se il colore va dal
```

```
   {                                       // nero al rosso,
     printf ("Non mi piace: %i\n", c); // visualizza un
   }                                       // messaggio e mostra
                                           // anche il numero
                                           // corrispondente.
…
```

66.7.2 Strutture

Gli array sono sequenze di elementi uguali, tutti adiacenti nel modello di rappresentazione della memoria, ideale o reale che sia. In modo simile si possono definire strutture di dati più complesse in cui gli elementi adiacenti siano di tipo differente. Gli elementi che compongono una struttura sono i suoi ***membri***. In pratica, una struttura è una sorta di mappa di accesso a un'area di memoria, attraverso i suoi membri.

La variabile contenente una struttura si comporta in modo analogo alle variabili di tipo primitivo, per cui, la variabile che è stata creata a partire da una struttura, rappresenta tutta la zona di memoria occupata dalla struttura stessa e non solo il riferimento al suo inizio. Questa distinzione è importante, per non fare confusione con il comportamento relativo agli array che sono sostanzialmente solo dei puntatori.

La dichiarazione di una struttura si articola in due fasi: la dichiarazione del tipo e la dichiarazione delle variabili che utilizzano quella struttura.

```
struct Datario { int giorno; int mese; int anno; };
```

L'esempio mostra la dichiarazione della struttura '**`Datario`**' (ovvero del tipo '**`struct Datario`**') composta da tre interi dedicati

a contenere rispettivamente: il giorno, il mese e l'anno. In questo caso, trattandosi di tre elementi dello stesso tipo, sarebbe stato possibile utilizzare un array, ma come è possibile vedere in seguito, una struttura può essere conveniente anche in queste situazioni.

È importante osservare che le parentesi graffe sono parte dell'istruzione di dichiarazione della struttura e non rappresentano un blocco di istruzioni. Per questo motivo appare il punto e virgola finale, cosa che potrebbe sembrare strana, specialmente quando la struttura si articola su più righe come nell'esempio seguente:

```
struct Datario {
    int giorno;
    int mese;
    int anno;
};              // Il punto e virgole finale è necessario.
```

La dichiarazione delle variabili che utilizzano la struttura può avvenire contestualmente con la dichiarazione della struttura, oppure in un momento successivo. L'esempio seguente mostra la dichiarazione del tipo '**`struct Datario`**', seguito da un elenco di variabili che utilizzano quel tipo: ***inizio*** e ***fine***.

```
struct Datario {
    int giorno;
    int mese;
    int anno;
} inizio, fine;
```

Tuttavia, il modo più elegante per dichiarare delle variabili a partire da una struttura è quello seguente:

```
struct Datario inizio, fine;
```

Quando una variabile è stata definita come organizzata secondo una

certa struttura, si accede ai suoi componenti attraverso l'indicazione del nome della variabile stessa, seguita dall'operatore punto ('.') e dal nome dell'elemento particolare.

```
inizio.giorno = 1;
inizio.mese = 1;
inizio.anno = 2012;
…
fine.giorno = inizio.giorno;
fine.mese = inizio.mese +1;
fine.anno = inizio.anno;
```

Una struttura può essere dichiarata in modo anonimo, definendo immediatamente tutte le variabili che fanno uso di quella struttura. La differenza sta nel fatto che la struttura non viene nominata nel momento della dichiarazione e, dopo la definizione dei suoi elementi, devono essere elencate tutte le variabili in questione. Evidentemente, non c'è la possibilità di riutilizzare questa struttura per altre variabili definite in un altro punto, ma soprattutto, come viene mostrato in seguito, diventa impossibile indicare il tipo di struttura come parametro formale di una funzione.

```
struct {
    int giorno;
    int mese;
    int anno;
} inizio, fine;
```

66.7.3 Assegnamento, inizializzazione, campo di azione e puntatori delle strutture

Nella sezione precedente si è visto come accedere ai vari componenti della struttura, attraverso una notazione che utilizza l'operatore

punto. Volendo è possibile assegnare a una variabile di questo tipo l'intero contenuto di un'altra che appartiene alla stessa struttura:

```
inizio.giorno = 1;
inizio.mese = 1;
inizio.anno = 2012
…
fine = inizio;
fine.mese++;
```

L'esempio mostra l'assegnamento alla variabile ***fine*** di tutta la variabile ***inizio***. Questo è ammissibile solo perché si tratta di variabili dello stesso tipo, cioè di strutture di tipo '**`Datario`**' (come deriva dagli esempi precedenti). Se invece si trattasse di variabili costruite a partire da strutture differenti, anche se realizzate nello stesso modo, con gli stessi membri, ciò non sarebbe ammissibile.

```
…
struct Datario {int giorno; int mese; int anno;};
struct Giorno  {int giorno; int mese; int anno;};
…
struct Datario ingresso = {31, 12, 2007};
struct Giorno  uscita;
uscita = ingresso;      // Errore: i dati sono incompatibili
…
```

Nel momento della dichiarazione di una struttura, è possibile anche inizializzarla utilizzando una forma simile a quella disponibile per gli array:

```
struct Datario inizio = { 1, 1, 2012 };
```

Oppure, per essere precisi e non dipendere dall'ordine dei campi nella struttura:

```
struct Datario inizio = { .giorno=1, .mese=1, .anno=2012 };
```

Dal momento che le strutture sono tipi di dati nuovi, per poterne fare uso occorre che la dichiarazione relativa sia accessibile a tutte le parti del programma che hanno bisogno di accedervi. Probabilmente, il luogo più adatto è al di fuori delle funzioni, eventualmente anche in un file di intestazione realizzato appositamente.

Ciò dovrebbe bastare a comprendere che le variabili che contengono una struttura vengono passate regolarmente attraverso le funzioni, purché la dichiarazione del tipo corrispondente sia precedente ed esterno alla descrizione delle funzioni stesse.

```
…
struct Datario { int giorno; int mese; int anno; };
…
void elabora (struct Datario oggi)
{
    …
}
```

L'esempio seguente che rappresenta un programma completo, serve a dimostrare che, nella chiamata di una funzione, la struttura viene passata per valore (e non per riferimento come avviene con gli array).

Listato 66.267. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/YZqBySyx* , *http://ideone.com/ffO5U* .

```
#include <stdio.h>

struct Datario {int giorno; int mese; int anno;};

void f (struct Datario d)
```

```
{
    unsigned int indirizzo = (int) &d;
    d.giorno = 28;
    d.mese = 2;
    d.anno = 2007;
    printf ("data %i-%i-%i inserita all'indirizzo %u\n",
            d.giorno, d.mese, d.anno, indirizzo);
}

int main (void)
{
    struct Datario data = {31, 12, 2007};
    unsigned int ind = (int) &data;
    f (data);
    printf ("data %i-%i-%i inserita all'indirizzo %u\n",
            data.giorno, data.mese, data.anno, ind);
    return 0;
}
```

Se si esegue il programma si ottiene un messaggio simile a quello seguente, dove si vede che gli l'indirizzi delle variabili contenenti la struttura, prima della chiamata della funzione e all'interno della stessa, sono differenti:

```
data 28-2-2007 inserita all'indirizzo 3212916960
data 31-12-2007 inserita all'indirizzo 3212916992
```

D'altro canto, se la variabile fosse la stessa, le modifiche fatte all'interno della funzione sarebbero visibili anche dopo la chiamata.

Così come nel caso dei tipi primitivi, anche con le strutture si possono creare dei puntatori. La loro dichiarazione avviene in modo intuitivo, come nell'esempio seguente:

```
struct Datario *p_data_fattura;
…
p_data_fattura = &inizio;
…
```

Quando si utilizza un puntatore a una struttura, diventa un po' più difficile fare riferimento ai vari componenti della struttura stessa, perché l'operatore punto ('`.`') che serve a unire il nome della struttura a quello dell'elemento, ha priorità rispetto all'asterisco che si utilizza per dereferenziare il puntatore:

```
*p_data_fattura.giorno = 15; // Non è valido!
```

L'esempio appena mostrato, non è ciò che sembra, perché l'asterisco posto davanti viene valutato dopo l'elemento '**`p_data_fattura.giorno`**', il quale non esiste. Per risolvere il problema si possono usare le parentesi, come nell'esempio seguente:

```
(*p_data_fattura).giorno = 15; // Corretto.
```

In alternativa si può usare l'operatore '**`->`**', fatto espressamente per i puntatori a una struttura:

```
p_data_fattura->giorno = 15; // Corretto.
```

L'esempio seguente è una variante di quello già presentato in precedenza per dimostrare il passaggio per valore delle variabili che contengono una struttura. Ma in questo caso, il passaggio dei dati avviene esplicitamente per riferimento.

Listato 66.273. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/vW8ktxAg* , *http://ideone.com/dYBHZ* .

```
#include <stdio.h>
```

```
struct Datario {int giorno; int mese; int anno;};

void f (struct Datario *d)
{
    unsigned int indirizzo = (int) d;
    d->giorno = 28;
    d->mese = 2;
    d->anno = 2007;
    printf ("data %i-%i-%i inserita all'indirizzo %u\n",
             d->giorno, d->mese, d->anno, indirizzo);
}

int main (void)
{
    struct Datario data = {31, 12, 2007};
    unsigned int ind = (int) &data;
    f (&data);
    printf ("data %i-%i-%i inserita all'indirizzo %u\n",
             data.giorno, data.mese, data.anno, ind);
    return 0;
}
```

In tal caso, gli indirizzi della struttura appaiono uguali e le modifiche applicate all'interno della funzione si riflettono nella variabile originale:

```
data 28-2-2007 inserita all'indirizzo 3214580384
data 28-2-2007 inserita all'indirizzo 3214580384
```

66.7.4 Scostamento all'interno delle strutture

Il file '`strdef.h`' della libreria standard definisce una macroistruzione che, attraverso la parvenza di una funzione, consente di misurare lo scostamento di un membro della struttura, rispetto all'inizio della stessa:

```
offsetof (tipo, membro)
```

Si osservi l'esempio seguente.

Listato 66.275. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/wW7KtJB2* , *http://ideone.com/QJrQv* .

```
#include <stdio.h>
#include <stddef.h>

struct Elenco {
    char  uno;
    short due;
    int   tre;
};

int main (int argc, char *argv[])
{
    size_t offset = offsetof (struct Elenco, due);
    printf ("Il membro \"due\" si trova %i byte dopo "
            "l'inizio della struttura.\n", offset);
    return 0;
}
```

Come si può vedere, la macroistruzione ***offsetof*** produce un risultato

di tipo '**size_t**'. Supponendo che il compilatore allinei i membri della struttura secondo multipli di due byte, il messaggio emesso dal programma potrebbe essere così:

```
Il membro "due" si trova 2 byte dopo l'inizio della struttura.
```

Pertanto, in questo caso, dopo il membro '**uno**' c'è un byte inutilizzato prima del membro '**due**'.

È il caso di ribadire che '**offsetof**' è una macroistruzione, ottenuta tramite le funzionalità del precompilatore. Diversamente, è probabile che sia impossibile realizzare una funzione che si comporti nello stesso modo apparente.

66.7.5 Unioni

«

L'unione permette di definire un tipo di dati accessibile in modi diversi, gestendolo come se si trattasse contemporaneamente di tipi differenti. La dichiarazione è simile a quella della struttura; quello che bisogna tenere a mente è che si fa riferimento alla stessa area di memoria; pertanto, lo spazio occupato è pari a quello del membro più grande.

```
union Livello {
    char c;
    int  i;
};
```

Si immagini, per esempio, di voler utilizzare indifferentemente una serie di lettere alfabetiche, oppure una serie di numeri, per definire un livello di qualcosa («A» equivalente a uno, «B» equivalente a due, ecc.). Le variabili generate a partire da questa unione, possono

essere gestite nei modi stabiliti, come se fossero una struttura, ma condividendo la stessa area di memoria.

```
union Livello carburante;
```

L'esempio mostra in che modo si possa dichiarare una variabile di tipo '**union Livello**', riferita all'omonima unione. Il bello delle unioni sta però nella possibilità di combinarle con le strutture.

```
struct Livello {
    char tipo;
    union {
        char c;          // Usato se tipo == 'c'.
        int  i;          // Usato se tipo == 'n'.
    };
};
```

L'esempio non ha un grande significato pratico, ma serve a chiarire le possibiltà. La variabile ***tipo*** serve ad annotare il tipo di informazione contenuta nell'unione, se di tipo carattere o numerico. L'unione viene dichiarata in modo anonimo come appartenente alla struttura.

L'esempio successivo, che è completo, permette di verificare l'ordine con cui vengono memorizzati i byte in memoria. L'unione dichiarata parte dal presupposto che un numero '**short int**' utilizzi l'equivalente di due caratteri:

Listato 66.280. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/K6to4aa3* , *http://ideone.com/YJk2f* .

```
#include <stdio.h>

union Little_big {
    short int i;         // 16 bit
```

```
    char c[2];          // 8 bit, 8 bit
};

int main (void)
{
    union Little_big lb;
    lb.i = 0x1234;
    printf ("%x %x%x\n", lb.i, lb.c[0], lb.c[1]);
    return 0;
}
```

Eseguendo il programma in un elaboratore con architettura *little endian* si ottiene il risultato seguente:

```
1234 3412
```

66.7.6 Campi

«

All'interno di una struttura è possibile definire l'accesso a ogni singolo bit di un tipo di dati determinato, oppure a gruppetti di bit. In pratica viene dato un nome a ogni bit o gruppetto.

```
struct Luci {
    unsigned char
        b0      :1,
        b1      :1,
        b2      :1,
        b3      :1,
        b4      :1,
        b5      :1,
        b6      :1,
        b7      :1;
};
```

L'esempio mostra l'abbinamento di otto nomi ai bit di un tipo '**char**'. Il primo, ***b0***, rappresenta il bit più a destra, ovvero quello meno significativo. Se il tipo '**char**' occupasse una dimensione maggiore di 8 bit, la parte eccedente verrebbe semplicemente sprecata.

```
struct Luci salotto;
…
salotto.b2 = 1;
```

L'esempio mostra la dichiarazione della variabile ***salotto*** come appartenente alla struttura mostrata sopra, quindi l'assegnamento del terzo bit a uno, probabilmente per «accendere» la lampada associata.

Volendo indicare un gruppo di bit maggiore, basta aumentare il numero indicato a fianco dei nomi dei campi, come nell'esempio seguente:

```
struct Prova {
    unsigned char
        b0      :1,
        b1      :1,
        b2      :1,
        stato   :4;
};
```

Nell'esempio appena mostrato, si usano i primi tre bit in maniera singola (per qualche scopo) e altri quattro per contenere un'informazione «più grande». Ciò che resta (probabilmente solo un bit) viene semplicemente ignorato.

66.7.7 Istruzione «typedef»

«

L'istruzione '**typedef**' permette di definire un nuovo di tipo di dati, in modo che la sua dichiarazione sia più agevole. Lo scopo di tutto ciò sta nell'informare il compilatore; '**typedef**' non ha altri effetti. La sintassi del suo utilizzo è molto semplice:

```
typedef tipo nuovo_tipo;
```

Si osservi l'esempio seguente:

```
typedef int numero_t;
numero_t x, y, z;
```

In questo modo viene definito il nuovo tipo '**numero_t**', corrispondente in pratica a un tipo intero, con il quale si dichiarano tre variabili: ***x***, ***y*** e ***z***. Le tre variabili sono di tipo '**numero_t**'. L'esempio seguente riguarda le enumerazioni:

```
typedef enum Colore { nero, marrone, rosso, arancio,
                      giallo, verde, blu, viola, grigio,
                      bianco } colore_t;
colore_t c, d;
```

In questo caso si definisce il tipo '**colore_t**', corrispondente a un'enumerazione con i nomi dei colori principali. Le variabili ***c*** e ***d*** vengono dichiarate con questa modalità. Dal momento che si usa '**typedef**', si potrebbe definire l'enumerazione in modo anonimo:

```
typedef enum { nero, marrone, rosso, arancio, giallo,
               verde, blu, viola, grigio, bianco } colore_t;
colore_t c, d;
```

L'esempio successivo riguarda le strutture:

```
struct Datario {
    int giorno;
    int mese;
    int anno;
};
typedef struct Datario data_t;
data_t inizio, fine;
```

Attraverso '**typedef**' è stato definito il tipo '**data_t**', facilitando così la dichiarazione delle variabili ***inizio*** e ***fine***. Ma in questo caso, si presta di più una struttura anonima:

```
typedef struct {
    int giorno;
    int mese;
    int anno;
} data_t;
data_t inizio, fine;
```

Tradizionalmente, i nomi dei tipi di dati creati con l'istruzione '**typedef**' hanno estensione '**_t**'.

66.7.8 Costanti letterali composte

È possibile rappresentare un array o una struttura attraverso una costante letterale, nota come ***costante letterale composta***. Formalmente si definisce la costante letterale composta secondo il modello seguente, dove le parentesi graffe fanno parte della definizione:

(***tipo***) { ***valore*** [, ***valore***] }

Per comprenderne l'utilizzo servono degli esempi e il caso più semplice riguarda la definizione degli array:

```
int *p = (int []) {3, 5, 76};
```

In questo modo si dichiara un array di interi, contenente rispettivamente i valori 3, 5 e 76, il cui indirizzo iniziale viene assegnato al puntatore ***p***. La variante seguente fa sì che il contenuto dell'array non possa essere modificato, ma per questo deve rendere altrettanto invariabile il contenuto raggiunto attraverso il puntatore:

```
const int *p = (const int []) {3, 5, 76};
```

Un array in forma letterale può essere trasmesso a una funzione. Quello che segue è un programma completo per dimostrare tale possibilità.

Listato 66.292. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/EkefhL40* , *http://ideone.com/dx4fQ* .

```
#include <stdio.h>

void f (int i[])
{
  printf ("i: %i %i %i\n", i[0], i[1], i[2]);
}

int main (int argc, char *argv[])
{
  f ((int []) {1, 3, 7});
  return 0;
}
```

In pratica, la funzione ***f()*** viene chiamata passando come argomento un array di tre interi, il quale logicamente viene trasmesso solo attraverso il puntatore al primo dei suoi elementi.

In modo analogo si possono rappresentare le strutture, ma in tal caso occorre disporre di un modello di riferimento, come si può vedere nell'esempio seguente che costituisce un altro programma completo.

Listato 66.293. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/1aTWnv01* , *http://ideone.com/PLEi0* .

```
#include <stdio.h>

struct Elenco {
    char  uno;
    short due;
    int   tre;
};

int main (int argc, char *argv[])
{
  struct Elenco e;
  e = (struct Elenco) { 33, 55, 77 };
  printf ("struttura: %i %i %i\n", e.uno, e.due, e.tre);
  return 0;
}
```

Ma naturalmente, i valori della struttura possono essere abbinati esplicitamente ai componenti a cui appartengono:

```
…
  e = (struct Elenco) { .uno=33, .tre=77, .due=55 };
…
```

Come per il caso degli array, anche le strutture rappresentate in forma letterale possono essere usate tra gli argomenti di una funzione. L'esempio seguente fa la stessa cosa di quello appena mostrato, con

la differenza che si avvale di una funzione per ottenere lo scopo.

Listato 66.295. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/4DUDe1bG* , *http://ideone.com/33Gx33G* .

```
#include <stdio.h>

struct Elenco {
    char  uno;
    short due;
    int   tre;
};

void f (struct Elenco e)
{
  printf ("struttura: %i %i %i\n", e.uno, e.due, e.tre);
}

int main (int argc, char *argv[])
{
  f ((struct Elenco) { 33, 55, 77 });
  return 0;
}
```

Anche in questo caso, naturalmente, si possono rendere espliciti i componenti della struttura a cui si attribuiscono i valori:

```
…
  f ((struct Elenco) { .uno=33, .tre=77, .due=55 });
…
```

A differenza dell'array, la struttura che si trova tra gli argomenti di una funzione viene passata integralmente; volendo trasmettere solo il suo indirizzo, si può usare l'operatore '**&**', come nell'esempio

seguente.

Listato 66.297. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/FJnL564M* , *http://ideone.com/mcodf* .

```
#include <stdio.h>

struct Elenco {
    char  uno;
    short due;
    int   tre;
};

void f (struct Elenco *e)
{
  printf ("struttura: %i %i %i\n", e->uno, e->due, e->tre);
}

int main (int argc, char *argv[])
{
  f (&(struct Elenco) {.uno=33, .tre=77, .due=55});
  return 0;
}
```

66.8 Tipi di dati speciali, di uso comune

Il linguaggio C prevede un insieme di tipi di dati tradizionali, a cui ci si riferisce con maggiore frequenza, e vari altri tipi, alcuni dei quali è bene conoscere.

66.8.1 Tipo «_Bool»

«

Lo standard C prevede un tipo particolare per la rappresentazione di valori logici, ovvero solo per i valori zero e uno. Nella tradizione del linguaggio, questo tipo manca e di norma si è rimediato rimpiazzandolo semplicemente con un valore intero, dal tipo '**`char`**' in su. Dal momento che è frequente l'uso di un tipo personalizzato (o di una macro-variabile del precompilatore) denominato ***bool***, lo standard ha inserito il tipo logico con il nome '**`_Bool`**', allo scopo di evitare conflitti con il codice esistente.

Il tipo '**`_Bool`**' può contenere solo i valori zero e uno; pertanto, la conversione di un numero di tipo diverso in un tipo '**`_Bool`**' avviene traducendo qualunque valore diverso da zero con il numero uno (*Vero*), mentre lo zero mantiene il suo valore invariato (*Falso*).

Lo standard non stabilisce come deve essere rappresentato in memoria il tipo '**`_Bool`**', anche se si tratta molto probabilmente di un byte intero che viene a essere sacrificato per lo scopo. Data la particolarità di questo tipo, non è detto che si possa utilizzare un puntatore per raggiungere l'area di memoria corrispondente.

Comprendendo il motivo per il quale questo tipo ha ricevuto un nome così particolare, diventa evidente che se lo si vuole utilizzare convenga creare una macro-variabile o un tipo derivato. D'altra parte, lo stesso file '`stdbool.h`' prescrive la definizione della macro-variabile '**`bool`**'.

In conclusione, se si desidera utilizzare un tipo di dati booleano, conviene fare riferimento alla macro-variabile ***bool***, la quale potrebbe anche essere ridefinita localmente nel proprio programma, se quello che si vuole non è conforme alle previsioni dello standard o delle

librerie del proprio compilatore.

66.8.2 Tipo «void»

Il tipo '**void**' rappresenta un'eccezione tra i tipi di dati usati nel linguaggio, in quanto rappresenta formalmente una variabile di rango nullo, e come tale incapace di contenere qualunque valore. La situazione più frequente di utilizzo del tipo '**void**' riguarda le funzioni, quando non devono restituire alcun valore: in tal caso si dichiara che sono di tipo '**void**'.

```
void procedura (int x)
{
  …
  return;
}
```

L'esempio mostra una funzione che, non dovendo restituire alcun valore, viene dichiarata di tipo '**void**'. Come si vede, l'istruzione '**return**' va usata, in questo caso, senza l'indicazione di un valore.

Quando una funzione non richiede parametri, si deve indicare esplicitamente questo fatto con la parola chiave '**void**', come dire che esiste sì un parametro, ma di rango nullo e come tale privo di qualunque informazione:

```
int funzione (void)
{
  …
}
```

In questo esempio, la funzione restituisce un valore intero, ma non fa uso di alcun parametro.

Il cast di tipo '**void**' può servire per annullare il risultato di un'espressione, quando ciò che interessa della stessa sono solo i suoi «effetti collaterali». In altri termini, quando un'espressione esegue qualche tipo di operazione, ma complessivamente si vuole scartare il risultato che viene generato, si può usare un cast di tipo '**void**'. Per esempio, quando si vuole usare una funzione, la quale restituirebbe un valore, del quale non si vuole fare alcun uso, si può indicare nella chiamata un cast al tipo '**void**', anche se di norma ciò non è necessario:

```
…
     (void) mia_funzione (…);
…
```

È possibile definire un puntatore generico al tipo '**void**', sapendo che questo è convertibile in tutti gli altri tipi di puntatore, con un cast appropriato e che è sempre possibile fare anche l'inverso:

```
     …
     void *p;
     …
     p = (void *) &a;
     …
```

Il puntatore nullo può essere definito, sia come un valore intero pari a zero, sia come tale valore tradotto in un puntatore generico, ovvero '**void ***':

```
     …
     int NULL = 0;
     …
```

```
...
void *NULL = (void *) 0;
...
```

Si osservi che un puntatore generico (‘**void ***’) non può essere incrementato o decrementato, perché fa riferimento a un’unità di memoria di dimensione nulla. Pertanto, per usare un puntatore del genere, quando si vuole scandire la memoria, prima va convertito in un puntatore di rango appropriato.

66.8.3 Tipo «size_t»

Secondo lo standard il tipo ‘**size_t**’ è definito nel file ‘`stddef.h`’, ma in pratica, dal momento che viene usato dall’operatore ‘**sizeof**’, potrebbe essere incorporato direttamente nel compilatore, tra i tipi fondamentali. A ogni modo si tratta normalmente di un tipo equivalente a un ‘**unsigned long int**’, destinato però a contenere la dimensione di qualcosa, intesa come intervallo tra due indirizzi (tra due puntatori), ma espressa come valore assoluto.

Listato 66.304. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/65fb65zpFR* , *http://ideone.com/9UNUA* .

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[])
{
    double a = 1.1;
    double b = 2.2;
    char * A = (char *) &a;
    char * B = (char *) &b;
    size_t s = abs (A - B);
    printf ("distanza: %i\n", s);
    return 0;
}
```

L'esempio mostra la dichiarazione di due variabili e di due puntatori alle variabili. Tuttavia, i puntatori sono di tipo '**char ***', in modo che la sottrazione tra i due dia la distanza in byte. Volendo, per non fare riferimento a un tipo particolare di puntatore, si potrebbe usare il tipo '**void**', ottenendo lo stesso risultato.

Listato 66.305. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/GFoNtYU5* , *http://ideone.com/bp9Qi* .

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[])
{
    double a = 1.1;
    double b = 2.2;
    void * A = (void *) &a;
    void * B = (void *) &b;
    size_t s = abs (A - B);
    printf ("distanza: %i\n", s);
    return 0;
}
```

Va osservato che il risultato mostrato dall'esecuzione dell'esempio compilato, dipende dal compilatore. In pratica, è il compilatore che decide come collocare in memoria le variabili; se si presume che siano adiacenti, si dovrebbe ottenere una distanza di otto byte.

66.8.4 Tipo «ptrdiff_t»

Per rappresentare la differenza tra due indirizzi, tenendo conto del segno, si usa il tipo '**ptrdiff_t**', definito anch'esso nel file '`stddef.h`'. Molto probabilmente si tratta di un tipo equivalente a un '**long int**'. Viene ripreso l'esempio già mostrato, senza calcolare il valore assoluto della differenza tra indirizzi.

Listato 66.306. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/WaunziA8* , *http://ideone.com/pzJC8* .

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
int main (int argc, char *argv[])
{
    double a = 1.1;
    double b = 2.2;
    void * A = (void *) &a;
    void * B = (void *) &b;
    ptrdiff_t s = (A - B);
    printf ("differenza: %i\n", s);
    return 0;
}
```

66.8.5 Tipo «va_list»

«

Il tipo '**va_list**' è definito dallo standard nel file di intestazione '`stdarg.h`', allo scopo di agevolare la scansione degli argomenti variabili, passati alle funzioni. Lo standard è vago sul significato che deve avere il tipo '**va_list**', ma in pratica dovrebbe trattarsi di un puntatore al tipo '**char**'.[23] Tuttavia il suo utilizzo rimane relegato alla scansione degli argomenti variabili, come descritto nella sezione 66.6.3. Viene comunque riportata qui la copia di un esempio che ne mostra l'uso.

Listato 66.307. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/5kUJnQxn* , *http://ideone.com/2W05Y* .

```
#include <stdio.h>
#include <stdarg.h>

void
f (int w,...)
{
    long double x;      // Dichiara le variabili che servono
    long long int y;    // a contenere gli argomenti per i
    int z;              // quali mancano i parametri formali.

    va_list ap;         // Dichiara il puntatore agli
                        // argomenti.

    va_start (ap, w);   // Posiziona il puntatore dopo la
                        // fine di «w».

    x = va_arg (ap, long double); // Estrae l'argomento
                                  // successivo portando
                                  // avanti il puntatore
                                  // di conseguenza.

    printf ("w = %i; ", w);      // Mostra il valore del
                                 // primo parametro.
    printf ("x = %Lf; ", x);     // Mostra il valore
                                 // dell'argomento successivo.

    y = va_arg (ap, long long int);  // Estrapola e mostra
    printf ("y = %lli; ", y);        // il terzo argomento.

    z = va_arg (ap, int);            // Estrapola e mostra
    printf ("z = %i\n", z);          // il quarto argomento.
```

```
    va_end (ap);                    // Conclude la scansione.

    return;
}

int main (int argc, char *argv[])
{
    f (10, (long double)12.34, (long long int)13, 14);
    return 0;
}
```

66.8.6 Tipo «wchar_t»

«

Per rappresentare un carattere esteso, ovvero un carattere dell'insieme universale, non è sufficiente il tipo '**char**' e per questo esiste invece il tipo '**wchar_t**' (*wide character type*), definito nel file 'stddef.h'.

Il tipo '**wchar_t**' è un intero, usato generalmente senza segno, di rango sufficiente a rappresentare tutti i caratteri che si intende di poter ammettere. È da osservare che per rappresentare l'insieme completo dei caratteri già definiti sono necessari anche più di 32 bit.

Il tipo '**wchar_t**' si usa sostanzialmente come il tipo '**char**', anche per ciò che riguarda gli array e le stringhe (che per essere tali devono essere terminate con il carattere nullo), ma si tratta sempre di una gestione interna, perché la rappresentazione richiede invece una trasformazione nella forma prevista dalla configurazione locale (sezione 66.9).

66.8.7 Tipo «wint_t»

Molte delle funzioni standard che in qualche modo hanno a che fare con un carattere singolo (perché ne ricevono il valore come argomento o perché restituiscono il valore di un carattere), lo fanno trattando il carattere come un tipo '**int**', ovvero, trattando il carattere senza segno e promuovendolo al rango di un intero normale. Questo sistema permette di distinguere tra tutti i caratteri dell'insieme ridotto e un valore ulteriore, rappresentato dalla macro-variabile ***EOF***, usata per rappresentare un errore in base al contesto.

Nella gestione dei caratteri estesi ci sono funzioni analoghe che svolgono lo stesso tipo di adattamento, ma in tal caso il valore del carattere viene gestito in qualità di '**wint_t**', il quale può rappresentare tutti i caratteri che sono ammessi dal tipo '**wchar_t**', con l'aggiunta del valore corrispondente a '**WEOF**' (diverso da tutti gli altri).

Il tipo '**wint_t**' e la macro-variabile ***WEOF*** sono definiti nel file '`wchar.h`'. Il tipo '**wint_t**' è, evidentemente, un intero di rango tale da consentire la rappresentazione di tutti i valori necessari.

66.8.8 Tipo «time_t»

Diverse funzioni dichiarate nel file '`time.h`' fanno riferimento al tipo '**time_t**' che rappresenta la quantità di unità di tempo trascorsa a partire da un'epoca di riferimento.

Frequentemente si tratta di un valore numerico intero che rappresenta la quantità di secondi trascorsi dall'epoca di riferimento (nei sistemi Unix è di norma l'ora zero del 1 gennaio 1970); inoltre, in un elaboratore che gestisca correttamente i fusi orari, è normale che questo valore sia riferito al tempo universale coordinato.

66.8.9 Tipo «struct tm»

«

La libreria standard, nel file ‘`time.h`’, prescrive che sia definito il tipo ‘**`struct tm`**’, con il quale è possibile rappresentare tutte le informazioni relative a un certo tempo, secondo le convenzioni umane:

```
struct tm {
    int tm_sec;     // Secondi:                  da 0 a 60.
    int tm_min;     // Minuti:                   da 0 a 59.
    int tm_hour;    // Ora:                      da 0 a 23.
    int tm_mday;    // Giorno del mese:          da 1 a 31.
    int tm_mon;     // Mese dell'anno:           da 0 a 11.
    int tm_year;    // Anno dal 1900.
    int tm_wday;    // Giorno della settimana: da 0 a 6
                    // con lo zero corrispondente alla
                    // domenica.
    int tm_yday;    // Giorno dell'anno:         da 0 a 365.
    int tm_isdst;   // Ora estiva. Contiene un valore
                    // positivo se è in vigore l'ora estiva;
                    // zero se l'ora è quella «normale»
                    // ovvero quella invernale;  un valore
                    // negativo se l'informazione non è
                    // disponibile.
};
```

66.8.10 Tipo «FILE»

«

Il tipo ‘**`FILE`**’ rappresenta una variabile strutturata con tutte le informazioni necessarie a individuare un flusso di file aperto. Di norma vengono usati puntatori, ovvero variabili di tipo ‘**`FILE *`**’, per tutte le operazioni di accesso relative a flussi di file aperti, tanto che nel gergo comune si confondono le cose e tali puntatori sono chiamati generalmente *stream*.

66.8.11 Tipo «fpos_t»

Alcune funzioni individuano la posizione di accesso ai file attraverso un insieme di dati. In quei casi, per rappresentare tale insieme di dati si usano variabili strutturate di tipo '**fpos_t**'.

66.9 Configurazione locale

La libreria standard del linguaggio C prevede la gestione della configurazione locale, attraverso l'indicazione di una stringa da associare a una ***categoria***, dove la categoria rappresenta il contesto particolare della configurazione locale a cui si vuole fare riferimento.

La stringa con cui si indica il tipo di configurazione desiderato, contiene le informazioni sulla lingua, la nazionalità e soprattutto la codifica da usare per la rappresentazione delle ***sequenze multibyte***. La codifica scelta condiziona l'insieme di caratteri che possono essere gestiti, sia attraverso le sequenze multibyte, sia attraverso i caratteri estesi.

66.9.1 Configurazione locale nei sistemi Unix e simili

In un sistema Unix o simile, la configurazione locale viene definita impostando alcune variabili di ambiente. Si tratta precisamente di variabili il cui nome inizia per '**LC_**...', dove in particolare la variabile '**LC_ALL**', se usata, prevale su tutte, mentre la variabile '**LANG**' (se '**LC_ALL**' non viene usata) serve per la configurazione predefinita di tutte le altre variabili '**LC_**...' che non fossero state dichiarate espressamente. A queste variabili di ambiente si associa una stringa secondo il formato seguente:

lingua_nazionalità . codifica

Per esempio, la configurazione '**de_CH.UTF-8**' rappresenta la configurazione di lingua tedesca per la Svizzera, con una codifica UTF-8.

Ogni variabile di ambiente '**LC_**...', esclusa '**LC_ALL**', rappresenta una categoria, ovvero un contesto particolare a cui applicare la configurazione locale. Per esempio, pur volendo gestire i numeri con una rappresentazione europea (con la virgola per i decimali), si potrebbe voler gestire le valute in dollari americani. Pertanto ci potrebbe essere un uso contrastante delle variabili '**LC_NUMERIC**' e '**LC_MONETARY**'.

66.9.2 Configurazione locale nel linguaggio C

Il linguaggio C non gestisce la configurazione locale attraverso le variabili di ambiente, perché non è detto che il sistema in cui si trova a operare il programma le preveda. Tuttavia definisce le categorie della configurazione locale attraverso macro-variabili (dichiarate nel file '`locale.h`') con gli stessi nomi e significati usati per le variabili di ambiente dei sistemi Unix e simili (vale anche il fatto che la macro-variabile ***LC_ALL*** si riferisca simultaneamente a tutte le categorie previste). Le macro-variabili in questione riguardano solo le categorie ***LC_***..., mentre la variabile di ambiente ***LANG*** non ha alcun corrispondente nel linguaggio e non rappresenta precisamente una categoria, ma solo un valore predefinito.

La configurazione locale di partenza per un programma scritto in linguaggio C è proprio la configurazione '**C**', la quale coincide so-

stanzialmente con la modalità di funzionamento tradizionale del linguaggio, con una codifica ASCII o equivalente. Per impostare la configurazione locale si usa la funzione ***setlocale()*** secondo il modello seguente:

```
char *setlocale (int categoria, const char *configurazione);
```

Il primo parametro è un numero intero che si indica normalmente attraverso una macro-variabile ***LC_***...; il secondo è una stringa, contenente la definizione della configurazione, per esempio '**it_IT.UTF-8**'. Se la funzione è nelle condizioni di accettare la configurazione richiesta, restituisce un puntatore alla stringa che definisce la configurazione stessa; altrimenti dà solo il puntatore nullo.

Come accennato, all'avvio ogni programma si trova a funzionare come se fosse stata usata la configurazione '**C**', ovvero come se fosse stata usata la funzione ***setlocale()*** così:

```
setlocale (LC_ALL, "C");
```

Per richiedere una configurazione più attuale e più utile, conviene specificare qualcosa che preveda la codifica UTF-8, con la quale è possibile rappresentare qualunque carattere della codifica universale:

```
setlocale (LC_ALL, "fr_CH.UTF-8");
```

Tuttavia, se il sistema operativo ha una gestione della configurazione locale, così come avviene nei sistemi Unix e simili, è meglio far sì che il programma erediti tale configurazione. Per ottenere questo, si usa la funzione ***setlocale()*** lasciando una stringa nulla (nel senso di

vuota) al posto della configurazione richiesta:

```
setlocale (LC_ALL, "");
```

Per interrogare la configurazione locale attiva per una certa categoria (o per tutte se si fa riferimento a '**`LC_ALL`**'), è sufficiente fornire il puntatore nullo al posto della stringa. L'esempio seguente è completo e si vede anche l'incorporazione del file '`locale.h`', contenente il prototipo della funzione ***setlocale()*** e la dichiarazione delle macro-variabili ***LC_***….

Listato 66.312. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/iomnt08Q* , *http://ideone.com/1Vk3V* .

```
#include <stdio.h>
#include <locale.h>
int main (int argc, char *argv[])
{
  setlocale (LC_ALL, "");
  char *loc;
  loc = setlocale (LC_ALL, NULL);
  printf ("LC_ALL: \"%s\"\n", loc);
  return 0;
}
```

Il programma potrebbe emettere il risultato seguente:

```
LC_ALL: "it_IT.UTF-8"
```

66.9.3 Caratteri multibyte e caratteri estesi

«

All'origine del linguaggio C esisteva una corrispondenza biunivoca tra carattere e byte. Attualmente, questa corrispondenza riguarda solo i caratteri dell'insieme minimo, il quale di norma coincide con quello della codifica ASCII. Per rappresentare caratteri che vanno al

di fuori dell'insieme minimo, si usano due metodi nel linguaggio: le sequenze multibyte, in cui un carattere è rappresentato attraverso una sequenza di più byte o comunque attraverso l'inserzione di codici che cambiano di volta in volta il sottoinsieme di riferimento, e i caratteri estesi che richiedono una unità di memorizzazione con un rango maggiore del byte. L'esempio seguente mostra l'uso di una stringa multibyte:

```
    printf ("€àèìòαδϕ\n");
```

È il contesto che fa capire la natura della stringa. In pratica, il file sorgente che contiene i caratteri deve essere scritto utilizzando una qualche codifica che preveda l'uso di più byte per rappresentare un carattere. La stessa codifica è quella che il programma deve usare durante il funzionamento per interpretare correttamente la stringa multibyte fornita.

In questo caso particolare, la funzione ***printf()*** non ha nemmeno bisogno di rendersi conto della codifica; semplicemente, se il programma funziona secondo la configurazione corretta, la visualizzazione del messaggio avviene come previsto.

Esistono diversi modi di gestire delle sequenze multibyte per rappresentare caratteri particolari, ma alcune sono più difficili da amministrare, perché richiedono il passaggio a sottoinsiemi di caratteri differenti attraverso l'uso di codici speciali, a cui si fa riferimento con il termine *shift*. In pratica, in tali condizioni, quando deve essere interpretata una stringa contenente sequenze multibyte, le funzioni devono tenere traccia dello stato di questa interpretazione, per sapere a quale sottoinsieme particolare di caratteri si sta facendo riferimento. Pertanto, l'interruzione e la ripresa di tale interpretazione devono

essere motivo di preoccupazione per il programmatore. Fortunatamente la tendenza è quella di usare la codifica UTF-8 per la rappresentazione dell'insieme universale dei caratteri, per tutte le lingue e tutte le nazionalità. Tale codifica ha il vantaggio di non richiedere la conservazione di uno stato (*shift status*), in quanto l'interpretazione di ogni carattere è indipendente dai precedenti: quello che è importante è evitare di spezzare l'interpretazione di un carattere a metà, ma anche se fosse, i caratteri successivi verrebbero individuati correttamente.

Dall'esempio mostrato si intende che una stringa multibyte si rappresenta letteralmente nello stesso modo di una stringa normale, con la differenza che la sua lunghezza in «caratteri», nel senso di unità '**`char`**', è maggiore dei caratteri che rappresenta. quindi, eventualmente, nel dimensionare un array di caratteri, occorre tenere conto di questo particolare.

Per rappresentare un carattere che va al di fuori dell'insieme minimo del linguaggio C, si può usare un carattere esteso, ovvero un valore intero di rango maggiore rispetto al tipo '**`char`**'. Si tratta precisamente del tipo '**`wchar_t`**' (*wide char*) che in condizioni normali va dai 16 ai 32 bit;.

Evidentemente, il rango del tipo '**`wchar_t`**' condiziona la quantità di caratteri che possono essere rappresentati. Per una rappresentazione abbastanza completa dell'insieme universale serve almeno un tipo '**`wchar_t`**' da 32 bit.

Si può rappresentare una costante letterale di tipo '**`wchar_t`**' mettendo anteriormente il prefisso '**`L`**'. Per esempio, '**`L'€'`**' viene con-

vertito dal compilatore in un carattere esteso che rappresenta numericamente il simbolo dell'euro. In modo analogo è possibile costruire array di elementi '**wchar_t**', per contenere stringhe estese (stringhe di caratteri '**wchar_t**' concluse da un valore nullo di terminazione, come per le stringhe normali). Anche per rappresentare le stringhe estese in modo letterale si può usare il prefisso '**L**'. Per esempio, '**L"àèìòù"**' viene tradotto dal compilatore in una stringa estesa.

Listato 66.315. Per provare il codice attraverso un servizio *pastebin*: *http://ideone.com/eWHAz* .

```
#include <stdio.h>
#include <locale.h>
#include <wchar.h>
int main (int argc, char *argv[])
{
  setlocale (LC_ALL, "en_US.UTF-8");
  wchar_t wc = L'ß';
  wchar_t wcs[] = L"€àèìòαστφ";
  printf ("%lc, %ls\n", wc, wcs);
  return 0;
}
```

L'esempio mostra l'uso delle costanti letterali riferite a caratteri e stringhe estese. In particolare, va osservato l'uso della funzione ***printf()***, in cui si indicano lo specificatore di conversione '**%lc**' per tradurre un carattere esteso e '**%ls**' per una stringa estesa. Ecco il risultato che si attende di visualizzare da quel programma:

```
ß, €àèìòαστφ
```

A questo punto è bene sia chiaro un concetto logico ma non sempre evidente: per gestire caratteri al di fuori dell'insieme minimo, è necessario definire la configurazione locale con una codifica che sia

tale da permetterlo. Pertanto, se non si usa la funzione ***setlocale()*** (così come invece avviene nell'esempio), si sta lavorando con la configurazione predefinita '**C**', per la quale non ci sono sequenze multibyte e diventa inutile l'uso del tipo '**wchar_t**'. Pertanto, se nell'esempio mancasse l'uso appropriato della funzione ***setlocale()***, non si otterrebbe la visualizzazione del testo come previsto.

66.9.4 Concatenamento eterogeneo

«

Il concatenamento di stringhe espresse in forma di costanti letterali, avviene, per le stringhe estese, esattamente come per le stringhe tradizionali, con l'eccezione che il concatenamento eterogeneo è ammissibile e implica sempre l'interpretazione di stringhe estese:

```
...
    wcp = "ciao amore" L"€àèìòαστφ";
...
```

In questo caso, la variabile ***wcp*** riceve il puntatore a una stringa estesa contenente precisamente la sequenza «ciao amore€àèìòαστφ», conclusa in modo appropriato.

Questo meccanismo consente, tra le altre cose, di concatenare delle macro-variabili che si espandono in stringhe letterali normali, in ogni circostanza, senza doverle duplicare per distinguerle in base al contesto.

66.9.5 Conversione tra caratteri multibyte e caratteri estesi

«

Un gruppo di funzioni dichiarate come prototipo nel file '`stdlib.h`' è importante per gestire la conversione tra caratteri multibyte e caratteri estesi. Le funzioni più importanti sono precisamente ***mbstowcs()***

(*Multibyte string to wide character string*) e ***wcstombs()*** (*Wide character string to multibyte string*), con lo scopo di convertire stringhe da multibyte a caratteri estesi e viceversa.

```
size_t mbstowcs (wchar_t *restrict wcs,
                 const char *restrict s,
                 size_t n);
```

```
size_t wcstombs (char *restrict s,
                 const wchar_t *restrict wcs,
                 size_t n);
```

La funzione '**mbstowcs**' si usa per convertire una stringa contenente sequenze multibyte in una stringa estesa, ovvero un array di elementi '**wchar_t**'. L'ultimo parametro rappresenta la quantità massima di caratteri estesi che devono essere inseriti nella stringa estesa di destinazione, contando anche il carattere nullo di terminazione. Il valore restituito è la quantità di caratteri che sono stati inseriti, escludendo il carattere nullo di terminazione, se c'è.

Listato 66.318. Per provare il codice attraverso un servizio *pastebin*: *http://ideone.com/Nx3eA* .

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
  setlocale (LC_ALL, "en_US.UTF-8");
  wchar_t wca[] = L"€€€€€€€€€€";
  wchar_t wcb[] = L"€€€€€€€€€€";
```

```
  size_t q;

  q = mbstowcs (wca, "äåâ", 3);
  printf ("mbstowcs: %i: \"%ls\"\n", q, wca);

  q = mbstowcs (wcb, "äåâ", 6);
  printf ("mbstowcs: %i: \"%ls\"\n", q, wcb);

  return 0;
}
```

L'esempio mostra la dichiarazione di due stringhe estese contenenti 10 caratteri estesi (oltre al carattere di terminazione della stringa). La funzione ***mbstowcs()*** viene usata la prima volta per tradurre la stringa multibyte '**`L"äåâ"`**' nei caratteri estesi corrispondenti all'inizio della prima delle due stringhe estese. Però, viene posto il limite al trasferimento di soli tre caratteri. Così facendo, il carattere di terminazione della stringa multibyte non viene convertito. Nel secondo caso, invece, si richiede il trasferimento di sei caratteri estesi, ma questo si ferma quando viene incontrato il carattere nullo di terminazione.

Entrambe le chiamate alla funzione ***mbstowcs()*** restituiscono il valore tre, perché sono solo tre i caratteri trasferiti, che siano diversi da quello di terminazione, ma nel secondo caso si può apprezzare la differenza nella stringa estesa risultante:

```
mbstowcs: 3: "äåâ€€€€€€€€"
mbstowcs: 3: "äåâ"
```

La funzione ***wcstombs()*** funziona in modo opposto, per convertire una stringa estesa in una stringa multibyte. In questo caso, l'ultimo

parametro rappresenta la quantità di byte che si vogliono ottenere con il trasferimento, incluso quello che rappresenta la terminazione della stringa. Logicamente, come nel caso dell'altra funzione, si ottiene la quantità di byte ottenuti dal trasferimento, ma senza contare il carattere nullo di terminazione.

Listato 66.320. Per provare il codice attraverso un servizio *pastebin*: *http://ideone.com/inlVK* .

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
  setlocale (LC_ALL, "en_US.UTF-8");
  char mba[] = "**********";
  char mbb[] = "**********";
  size_t n;

  n = wcstombs (mba, L"äåâ", 6);
  printf ("wcstombs: %i: \"%s\"\n", n, mba);

  n = wcstombs (mbb, L"äåâ", 9);
  printf ("wcstombs: %i: \"%s\"\n", n, mbb);

  return 0;
}
```

Questo nuovo esempio è analogo al precedente, ma invertendo il ruolo delle stringhe: questa volta la stringa estesa viene convertita in una stringa multibyte. Nel caso particolare della codifica UTF-8, ognuna delle lettere che si vedono nella stringa estesa si traduce in una sequenza di due byte; pertanto, la conversione richiede che

siano convertiti almeno sette byte, per includere anche il carattere nullo di terminazione. Si può vedere che nel primo caso il carattere nullo non viene convertito, pertanto la stringa di destinazione continua ad apparire della lunghezza originale, pur con la prima parte sovrascritta. Naturalmente, rimangono solo quattro asterischi perché la sequenza multibyte necessaria a rappresentare quelle tre lettere è complessivamente di sei byte.

```
wcstombs: 6: "äåâ****"
wcstombs: 6: "äåâ"
```

La conversione, in un verso o nell'altro, può fallire. Se queste funzioni incontrano dei problemi, restituiscono l'equivalente di −1 tradotto secondo il tipo '**`size_t`**' (in pratica, utilizzando una rappresentazione dei valori negativi in complemento a due, si ottiene il valore positivo massimo che la variabile possa rappresentare, essendo '**`size_t`**' senza segno).

Ci sono altri dettagli sull'uso di queste funzioni, ma si possono approfondire leggendo la sezione 69.9.11 e le pagine di manuale *mbstowcs(3) wcstombs(3)*.

66.10 Organizzazione dei file sorgenti

«

Quando si scrive un programma che non sia estremamente banale, diventa importante organizzare i file dei sorgenti in un modo gestibile. Se l'esperienza di programmazione da cui si proviene, quando ci si rivolge al C, è quella dei linguaggi interpretati, si può essere tentati di scrivere tutto il proprio programma in un file solo, ma questo approccio può essere controproducente. D'altra parte, per dividere il lavoro in più file, occorre che tale suddivisione abbia un senso pratico, conforme alla filosofia del linguaggio.

66.10.1 File di intestazione

La direttiva '**`#include`**' del precompilatore consente di incorporare un altro file, scritto secondo le regole del linguaggio, come se il suo contenuto facesse parte del file incorporante. Tradizionalmente questi file che vengono incorporati sono «file di intestazione», a cui si dà un'estensione diversa, '`.h`', proprio per distinguerne lo scopo.

Un file di intestazione, perché sia utile e non serva a creare maggiore confusione, può contenere la dichiarazione di macro-variabili, di macroistruzioni, di tipi derivati, di prototipi di funzione e di variabili pubbliche. Non ha senso inserire il codice completo delle funzioni all'interno di un file di intestazione, perché queste verrebbero replicate inutilmente nei file-oggetto, ogni volta che viene incorporato il file stesso.

Se si rispetta questo principio, un file di intestazione può essere incorporato in diversi file, garantendo un uso uniforme di quanto dichiarato al suo interno, senza duplicazioni inutili nel risultato della compilazione, anche se ciò che contiene tale file viene usato solo parzialmente o non viene usato affatto.

Un file di intestazione deve contenere ciò che serve alla soluzione di un certo tipo di problematica, ben delimitata. In particolare, dovrebbe contenere tutti i prototipi delle funzioni che servono, o possono servire, per quel tale problema.

66.10.2 Funzioni pubbliche

Le funzioni che devono poter essere usate in varie parti del programma è bene siano pubbliche (come avviene in modo predefinito) e che siano descritte come prototipo in un file di intestazione appropria-

to. Per quanto possibile, le funzioni potrebbero essere scritte in file indipendenti, ovvero: un file distinto per ogni funzione.

Dal momento che le funzioni potrebbero avere bisogno di usare macro-variabili o macroistruzioni definite nel file di intestazione che ne dichiara i prototipi, nei file di queste funzioni dovrebbe apparire l'inclusione del file di intestazione rispettivo.

66.10.3 Funzioni e variabili private

Le funzioni dichiarate con la parola chiave '**`static`**' sono visibili solo all'interno del file-oggetto in cui vanno a finire. Queste funzioni statiche sono utili in quanto vengono chiamate da una sola o da poche funzioni; in tal caso, questo gruppo di funzioni è costretto a convivere nello stesso file.

Lo stesso problema riguarda le variabili che devono essere utilizzate da più funzioni, ma che non devono essere visibili alle altre, perché anche in questo caso si rende necessario il mettere tale insieme nello stesso file.

66.10.4 Esempio di «stdlib.h»

Per comprendere il senso di quanto appena descritto in modo così sintetico, è utile osservare l'organizzazione della libreria C standard, anche se poi nella realtà i contenuti dei file che la compongono non sono sempre facili da interpretare. A ogni modo, qui viene proposto il caso di quella parte della libreria C che fa capo al file di intestazione '`stdlib.h`'.

Per cominciare, già dal nome del file scelto come esempio, va osservato che un file di intestazione realizzato in modo conforme alla filosofia del linguaggio rappresenta una «libreria» di qualcosa, anche

se, per le funzioni, contiene solo i prototipi. Ecco, in breve, come potrebbe essere fatto il file ‘`stdlib.h`’, omettendo alcune porzioni ridondanti per i fini della spiegazione:

```
#ifndef _STDLIB_H
#define _STDLIB_H       1
#define NULL 0
typedef unsigned long int size_t;
typedef unsigned int wchar_t;
#include <limits.h>
typedef struct {int quot; int rem;} div_t;
…
#define RAND_MAX        INT_MAX
…
int   atoi    (const char *nptr);
…
int   rand    (void);
void  srand   (unsigned int seed);
void *malloc  (size_t size);
void *realloc (void *ptr, size_t size);
void  free    (void *ptr);
#define calloc(nmemb, size) (malloc ((nmemb) * (size)))
…
#endif
```

Si può osservare che l’interpretazione del contenuto del file è subordinata al fatto che la macro-variabile ***_STDLIB_H*** non sia già stata dichiarata, mentre altrimenti viene dichiarata. In pratica, con questo meccanismo, se per qualunque ragione un file si trova a incorporare più volte il file di intestazione, il compilatore considera quel contenuto solo la prima volta.

Nell’esempio si vedono dichiarazioni di macro-variabili, di macroi-

struzioni (***calloc()*** è, in questo caso, una macroistruzione), di tipi di dati derivati. Secondo il buon senso, tutte queste cose devono servire alle funzioni di cui sono presenti i prototipi, ma soprattutto per ciò che riguarda i prototipi. Per esempio, la macro-variabile ***NULL*** viene dichiarata nel file di intestazione perché è il valore che potrebbe essere restituito da funzioni come ***malloc()*** e deve essere uniformato; il tipo derivato '**`size_t`**' viene dichiarato perché viene usato dalla funzione ***malloc()*** e da altre; il file '`limits.h`' viene incorporato perché definisce il valore della macro-variabile ***INT_MAX*** che in questo caso viene usato per definire ***RAND_MAX***, la quale deve essere uniformata per l'uso con la funzione ***rand()***.

La funzione ***atoi()*** è utile per dimostrare in che modo mettere ogni funzione nel proprio file indipendente. Per esempio, quello che segue potrebbe essere il file '`atoi.c`':

```
#include <stdlib.h>:
#include <ctype.h>:
int
atoi (const char *nptr)
{
  int i;
  int sign = +1;
  int n;

  for (i = 0; isspace (nptr[i]); i++)
    {
      ;       // Si limita a saltare gli spazi iniziali.
    }

  if (nptr[i] == '+')
    {
      sign = +1;
```

```
      i++;
    }
  else if (nptr[i] == '-')
    {
      sign = -1;
      i++;
    }

  for (n = 0; isdigit (nptr[i]); i++)
    {
      n = (n * 10) + (nptr[i] - '0'); // Accumula il valore.
    }

  return sign * n;
}
```

Come si vede, questa versione di ***atoi()*** si avvale delle funzioni ***isspace()*** e ***isdigit()***, dichiarate nel file '`ctype.h`' che viene aggiunto di conseguenza all'elenco delle inclusioni. Questa inclusione non è stata fatta nel file di intestazione '`stdlib.h`', perché l'uso delle funzioni ***isspace()*** e ***isdigit()*** è dovuto soltanto a una scelta realizzativa di ***atoi()*** e non perché la libreria '`stdlib.h`' dipenda necessariamente da '`ctype.h`'.

Per realizzare le funzioni ***rand()*** e ***srand()*** deve essere condivisa una variabile, la quale può essere nascosta prudentemente al resto del programma. Pertanto serve un file unico che incorpori entrambe le funzioni:

```
#include <stdlib.h>
static unsigned int _srand = 1; // Il rango di «_srand»
                                // deve essere maggiore o
                                // uguale a quello di
                                // «RAND_MAX» e di
                                // «unsigned int».
int
rand (void)
{
  _srand = _srand * 1234567 + 12345;
  return _srand % ((unsigned int) RAND_MAX + 1);
}

void
srand (unsigned int seed)
{
  _srand = seed;
}
```

66.10.5 Parametri delle macroistruzioni

«

Quando si dichiara una macroistruzione, si usano delle macro-variabili interne che rappresentano i parametri per la «chiamata» di questa specie di funzione. Dal momento che il codice che costituisce la macroistruzione può avvalersi di altre macro-variabili già dichiarate e dato che di norma queste hanno nomi che utilizzano lettere maiuscole, è bene che quelle interne siano scritte con sole lettere minuscole. In pratica, conviene fare come nella macroistruzione già apparsa nella sezione precedente:

```
#define calloc(nmemb, size) (malloc ((nmemb) * (size)))
```

Al contrario, facendo come nell'esempio successivo, il rischio che sia già stata dichiarata la macro-variabile ***SIZE*** oppure ***NMEMB*** è

più alto:

```
#define calloc(NMEMB, SIZE) (malloc ((NMEMB) * (SIZE)))
```

66.10.6 Compilazione

I vari file con estensione '`.c`' possono essere compilati separatamente, per ottenere altrettanti file-oggetto da collegare successivamente (i file '`.h`' devono essere incorporati da file '`.c`', pertanto non vanno compilati da soli). Per esempio, per un certo gruppo di file collocato in una certa directory, si potrebbe usare un file-make simile a quello seguente:

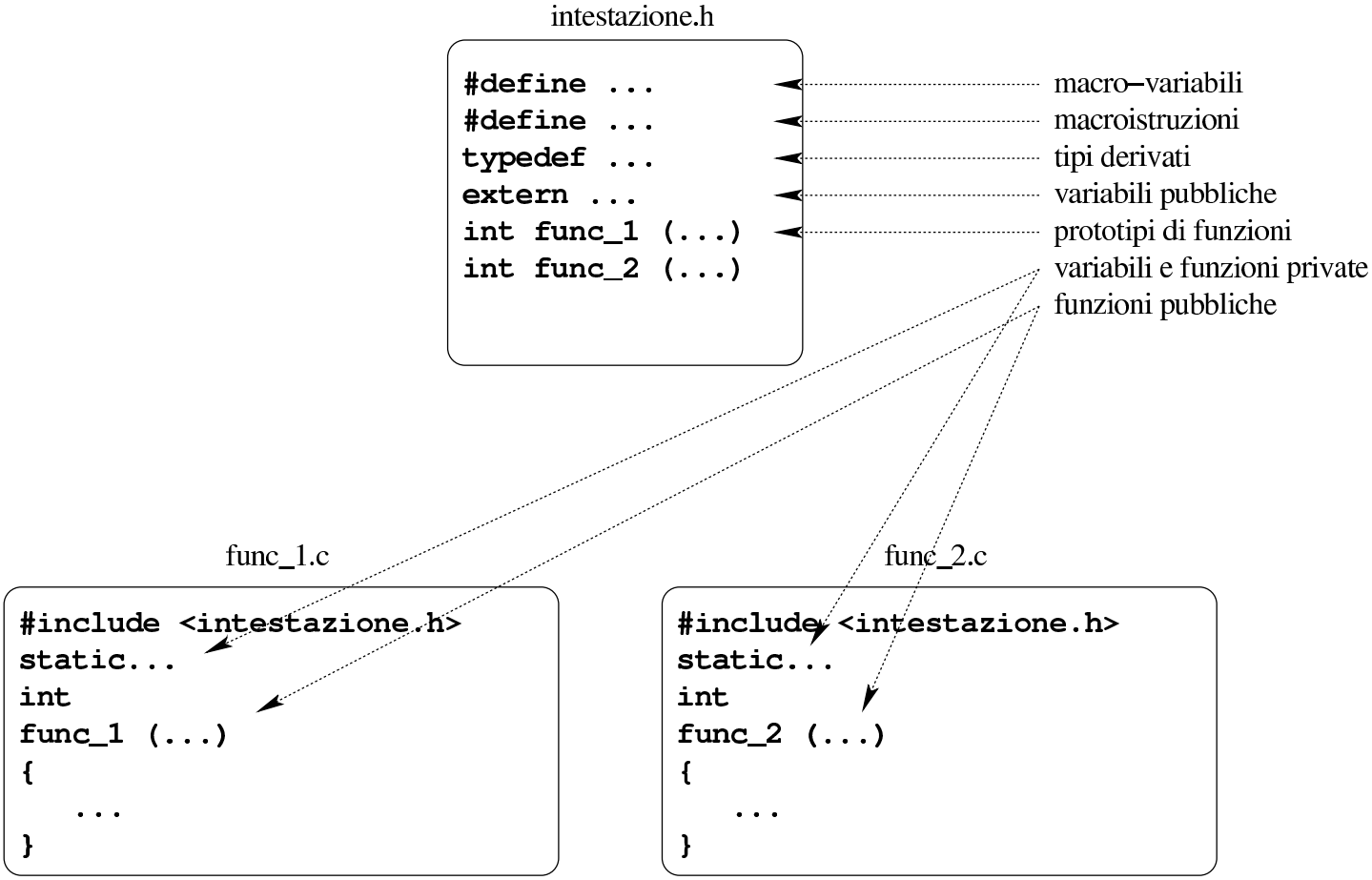
```
sorgenti = uno due tre
#
all: $(sorgenti)
#
clean:
        @rm *.o 2> /dev/null
#
$(sorgenti):
        @echo $@.c
        @gcc -Wall -Werror -o $@.o -c $@.c -I../include
```

In pratica, si presume che nella directory in cui si trova il file-make, ci siano i file '`uno.c`', '`due.c`' e '`tre.c`', per i quali si vogliono ottenere altrettanti file-oggetto, con l'estensione appropriata. Si presume anche che i file di intestazione a cui i sorgenti fanno riferimento si trovino nella directory '`../include/`'.

Compilando in questo modo i file che contengono il minimo indispensabile (possibilmente una sola funzione per ciascuno), quando si verificano errori è più semplice concentrare l'attenzione per correggerli.

Quando si dispone dei file-oggetto si può passare al collegamento (*link*), ma anche in questa fase possono emergere dei problemi di tipo diverso: di solito si tratta di una funzione che viene chiamata, della quale esiste solo il prototipo e quindi non si trova in alcun file-oggetto. Naturalmente, il collegamento deve avvenire una volta sola, con tutti i file-oggetto che compongono il programma.

Figura 66.328. Indicazioni generali per la stesura di un insieme di file sorgenti ordinato.



66.11 K&R

Il linguaggio C, nella sua versione originale, nota come «K&R» (Kernigham e Ritchie), aveva delle caratteristiche che, fortunatamente, sono state perdute. Generalmente non è necessario conoscere

le particolarità del vecchio linguaggio C, ma può capitare di leggere del vecchio codice, oppure può succedere di dover usare un vecchio compilatore.

Qui si annotano le caratteristiche principali della sintassi K&R, rispetto al linguaggio C, nella sua forma attuale.

66.11.1 Prototipi e chiamate delle funzioni

La differenza più importante tra la sintassi K&R e il linguaggio C attuale, sta nel modo di dichiarare i prototipi delle funzioni. Il prototipo di una funzione K&R non contiene la definizione dei tipi dei parametri (e tantomeno permette di attribuire loro dei nomi). Per esempio, il prototipo

```
int funzione (char a, short b, int c, long d, float e, double f);
```

si riduce, secondo la sintassi K&R, semplicemente nella dichiarazione seguente:

```
int funzione ();
```

Nella sintassi K&R, la mancanza di un prototipo vero e proprio, fa sì che nella chiamata di una funzione occorra essere molto precisi con i tipi degli argomenti; in altri termini, tutto quello che non ha il rango di '**int**', va controllato attentamente. Per esempio, supponendo che il rango di '**long int**' sia effettivamente maggiore di quello di '**int**', la chiamata seguente provoca certamente dei problemi:

```
x = funzione ('a', 123, 456, 789, 12.3, 45.6);
```

Si comprende che l'argomento attuale 789 sia effettivamente di tipo '**int**', mentre la funzione si attende un rango maggiore, con risultati non prevedibili. Per quanto riguarda invece i tipi '**char**' e '**short**

`int`’, va osservato che la sintassi K&R prevede la promozione automatica a ‘**`int`**’, inoltre, per il tipo ‘**`float`**’ è prevista la promozione a ‘**`double`**’.

Come si può intuire, **anche la quantità prevista degli argomenti di una chiamata non è determinabile per il compilatore**, con le conseguenze che si possono immaginare.

66.11.1.1 La macroistruzione «_PROTOTYPE» di Minix

«

Il codice del sistema operativo Minix è nato in un momento in cui si potevano incontrare sia compilatori C che riconoscevano e richiedevano l’uso di prototipi di funzione con l’indicazione dei parametri, sia di compilatori che potevano accettare solo la sintassi K&R. Per ovviare a questo problema, il codice del sistema Minix adotta l’uso di una macroistruzione, denominata ***_PROTOTYPE***, dichiarata così:

```
#if _ANSI
…
#define _PROTOTYPE(function, params)    function params
…
#else
…
#define _PROTOTYPE(function, params)    function()
…
#endif
```

Successivamente, quando viene il momento di dichiarare un prototipo, questo viene scritto come nell’esempio seguente:

```
_PROTOTYPE( int printf, (const char *_format, ...)              );
_PROTOTYPE( int scanf,  (const char *_format, ...)              );
```

66.11.2 Dichiarazione delle funzioni

Il modello sintattico che descrive la dichiarazione delle funzioni secondo il C di K&R, potrebbe essere espresso come nello schema seguente:

```
tipo nome_funzione (par_1 [, par_2] ...)
tipo par_1;
[tipo par_2;]
...
{
    ...
}
```

L'esempio seguente mostra la dichiarazione di una certa funzione, secondo la sintassi attuale del linguaggio C:

```
int
funzione (int i, int j)
{
    int k;
    k = i + j;
    return (k);
}
```

Così sarebbe invece secondo la sintassi K&R:

```
int
funzione (i, j)
int i;
int j;
{
    int k;
    k = i + j;
    return (k);
}
```

Tra l'altro, ciò può far incorrere in un errore, che il compilatore non segnala:

```
int
funzione (i, j)
int i;
int j;
{
    int i;
    int k;
    k = i + j;
    return (k);
}
```

In questo caso, la variabile ***i*** viene dichiarata anche nel corpo della funzione, oscurando il contenuto del parametro ***i***.

66.11.3 Operatori composti di assegnamento

«

Nel linguaggio C comune si possono utilizzare degli operatori di assegnamento composti, come nell'esempio seguente in cui si vuole incrementare di due unità la variabile ***i***:

```
i += 2;
```

Nella sintassi K&R, scrivere '**+=**' oppure '**+ =**' non fa differenza, mentre nello standard attuale del linguaggio ciò non è più ammissibile.

Nelle primissime versioni della sintassi K&R, gli operatori composti erano invertiti, pertanto, avrebbe potuto essere scritto:

```
i =+ 2; /* da non fare mai! */
```

Si può osservare che nella sintassi K&R non è possibile usare il segno '**+**' al di fuori della somma, perché non avrebbe alcuna utilità (dal momento che «***+x***» è uguale a ***x***); pertanto, il fatto che si possa anche scrivere '**i = + 2;**', non dovrebbe creare difficoltà. Tuttavia, scrivendo l'istruzione seguente, c'è da domandarsi cosa si intenda veramente:

```
i =- 2; /* da non fare mai! */
```

La variabile ***i*** viene ridotta di due unità, oppure le viene assegnato semplicemente il valore −2?

66.11.4 Tipi numerici

Nella sintassi K&R, le costanti numeriche in ottale possono contenere anche le cifre 8 e 9, senza che il compilatore si allarmi di ciò. Inoltre, le costanti numeriche rappresentano sempre un numero di tipo intero normale ('**int**'), a meno che gli si aggiunga la lettera '**L**', per indicare che si tratta di un tipo '**long int**'.

I tipi numerici disponibili sono minori rispetto allo standard attuale del linguaggio C, mancando il tipo '**long long int**' e il tipo '**long double**'. Inoltre, nella sintassi K&R non è previsto l'uso dello specificatore '**unsigned**'.

66.11.5 Tipo «void *»

«

Per la sintassi K&R, il tipo '**`void *`**' è equivalente al tipo '**`char *`**'. Pertanto, l'incremento di un tale puntatore porta ai byte successivi, mentre così non può avvenire secondo la sintassi attuale.

66.11.6 Direttive del preprocessore

«

Nella sintassi K&R, le direttive del preprocessore devono avere il cancelletto ('**`#`**') esattamente nella prima colonna; inoltre non sono ammissibili direttive nulle, in cui il cancelletto sta da solo.

Le direttive '**`#elif`**', '**`#error`**' e '**`#pragma`**' non sono disponibili. Gli operandi '**`defined`**', '**`#`**' e '**`##`**' non sono disponibili.

66.11.7 Altre osservazioni su K&R

«

- Sulla funzione ***main()*** non si specifica cosa debba o possa restituire.
- Non sono disponibili le sequenze triplici, o *trigraph*, e di conseguenza non viene riconosciuta la sequenza '**`\?`**' nelle costanti carattere o nelle stringhe.
- La porzione significativa dei nomi degli identificatori (nomi di funzioni, di variabili, ecc.) è di soli otto caratteri.
- Le sequenze '**`\a`**', '**`\?`**' e '**`\x`**', nelle costanti carattere e nelle stringhe, non sono riconosciute; al contrario, sono ammissibili le sequenze '**`\8`**' e '**`\9`**', che invece non dovrebbero, trattandosi di riferimenti a valori in ottale.
- Le costanti stringa, potrebbero risultare modificabili, mentre la sintassi attuale del linguaggio non lo deve consentire.

- L'operatore '`&`', usato per ottenere il puntatore a una variabile, non può essere usato con i nomi degli array.
- L'inizializzazione di una variabile, mentre viene dichiarata, può essere fatta omettendo il segno '`=`'. Per esempio, al posto di scrivere '`int x = 3 + 4;`', si può scrivere '`int x 3 + 4;`'.
- Nella struttura di controllo '`switch`', l'espressione che viene valutata per la scelta dell'azione da compiere deve essere di tipo '`int`'.
- Le enumerazioni non sono disponibili.

66.11.8 Unproto

Se per qualche ragione si deve usare un compilatore C che è rimasto a standard precedenti al 1987, viene in aiuto il programma Unproto,[24] di Wietse Venema, che va inserito tra il preprocessore C e il compilatore vero e proprio.

Unproto è in grado di trasformare il risultato prodotto dal preprocessore in un codice C compatibile con la sintassi K&C, sia per la questione legata ai prototipi e la dichiarazione delle funzioni, sia per altri problemi meno appariscenti.

Unproto è anche incluso nella distribuzione Dev86, ovvero gli strumenti di sviluppo per 8088/8086, dove il compilatore BCC di Bruce Evans se ne avvale automaticamente.

66.12 Riferimenti

- Brian W. Kernigham, Dennis M. Ritchie, *The C programming language*, Prentice-Hall 1978, prima edizione ISBN 0-13-110163-3; seconda edizione 0-13-110362-8, 0-13-110370-9; edizione italiana, Pearson, ISBN 88-7192-200-X, *http://cm.bell-labs.com/cm/cs/cbook/*).
- Eric Giguere, *The ANSI Standard: A Summary for the C Programmer*, 1987, *http://www.ericgiguere.com/articles/ansi-c-summary.html*
- Open Standards, *C - Approved standards*, *http://www.open-std.org/jtc1/sc22/wg14/www/standards*
- *ISO/IEC 9899:TC2*, *http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf*
- Richard Stallman e altri, *GNU coding standards*, *http://www.gnu.org/prep/standards/*
- Autori vari, *GCC manual*, *http://gcc.gnu.org/onlinedocs/gcc/* , *http://gcc.gnu.org/onlinedocs/gcc.pdf*
- Douglas Walls, *How to use the restrict qualifier in C*, *http://developers.sun.com/solaris/articles/cc_restrict.html*
- *SUPER-UX C Programmer's Guide, DIFFERENCES BETWEEN SUPER-UX ANSI C AND K&R C*, *http://static.cray-cyber.org/Documentation/NEC_SX_R10_1/G1AF02E/CHAP1.HTML#1.3*
- Wietse Zweitze Venema, *Wietse's tools and papers, Unproto*, *ftp://ftp.porcupine.org/pub/security/index.html* , *ftp://ftp.porcupine.org/pub/unix/unproto5.shar.Z*

- Robert de Bath, *Dev86: a cross development C compiler, assembler and linker environment for the production of 8086 executables*, http://homepage.ntlworld.com/robert.debath/dev86/

[1] È bene osservare che un'istruzione composta, ovvero un raggruppamento di istruzioni tra parentesi graffe, non è concluso dal punto e virgola finale.

[2] In particolare, i nomi che iniziano con due trattini bassi ('__'), oppure con un trattino basso seguito da una lettera maiuscola ('_*X*') sono riservati.

[3] Tuttavia, le estensioni POSIX prevedono la possibilità di avere tre parametri: '**`int main (int argc, char *argv[], char *envp[])`**'.

[4] Il linguaggio C, puro e semplice, non comprende alcuna funzione, benché esistano comunque molte funzioni più o meno standardizzate, come nel caso di ***printf()***.

[5] Quando il linguaggio C viene usato secondo lo standard POSIX, ovvero ciò che definisce le caratteristiche di un sistema operativo Unix, il byte deve essere precisamente di 8 bit, senza altre possibilità.

[6] Sono esistiti anche elaboratori in grado di indirizzare il singolo bit in memoria, come il Burroughs B1900, ma rimane il fatto che il linguaggio C si interessi di raggiungere un byte intero alla volta.

[7] Il qualificatore '**`signed`**' si può usare solo con il tipo '**`char`**', dal momento che il tipo '**`char`**' puro e semplice può essere con o senza segno, in base alla realizzazione particolare del linguaggio che dipende dall'architettura dell'elaboratore e dalle convenzioni del

sistema operativo.

[8] La distinzione tra valori con segno o senza segno, riguarda solo i numeri interi, perché quelli in virgola mobile sono sempre espressi con segno.

[9] Come si può osservare, la dimensione è restituita dall'operatore '**`sizeof`**', il quale, nell'esempio, risulta essere preceduto dalla notazione '**`(int)`**'. Si tratta di un cast, perché il valore restituito dall'operatore è di tipo speciale, precisamente si tratta del tipo '**`size_t`**'. Il cast è solo precauzionale perché generalmente tutto funziona in modo regolare senza questa indicazione.

[10] Per la precisione, il linguaggio C stabilisce che il «byte» corrisponda all'unità di memorizzazione minima che, però, sia anche in grado di rappresentare tutti i caratteri di un insieme minimo. Pertanto, ciò che restituisce l'operatore ***sizeof()*** è, in realtà, una quantità di byte, solo che non è detto si tratti di byte da 8 bit.

[11] Gli operandi di '**`? :`**' sono tre.

[12] Lo standard prevede il tipo di dati '**`_Bool`**' che va inteso come un valore numerico compreso tra zero e uno. Ciò significa che il tipo '**`_Bool`**' si presta particolarmente a rappresentare valori logici (binari), ma ciò sempre secondo la logica per la quale lo zero corrisponde a *Falso*, mentre qualunque altro valore corrisponde a *Vero*.

[13] Per la precisione, i parametri di una funzione corrispondono alla dichiarazione di variabili di tipo automatico.

[14] Questa descrizione è molto semplificata rispetto al problema del campo di azione delle variabili in C; in particolare, quelle che qui vengono chiamate «variabili globali», non hanno necessariamente un campo di azione esteso a tutto il programma, ma in condizioni

normali sono limitate al file in cui sono dichiarate. La questione viene approfondita in modo più adatto a questo linguaggio nella sezione 66.3.

[15] In pratica, ***EXIT_SUCCESS*** equivale a zero, mentre ***EXIT_FAILURE*** equivale a uno.

[16] Lo standard non impone che si tratti di file veri e propri; tuttavia, in un sistema Unix o in qualunque altro sistema operativo analogo, questi sarebbero file da cercare secondo criteri stabiliti, come viene descritto.

[17] In fase di collegamento (*link*) può darsi che il programma che svolge questo compito richieda che i file-oggetto siano indicati secondo una certa sequenza logica, ma questo problema, se esiste, è al di fuori della competenza del linguaggio C.

[18] Si ricorda che, in questo contesto, per «file» si intende il risultato dell'elaborazione da parte del precompilatore, il quale a sua volta potrebbe avere fuso assieme diversi file.

[19] Una variabile potrebbe rappresentare un registro del microprocessore e in tal caso non si potrebbe costruire un puntatore alla stessa. Pertanto, l'argomento sui puntatori parte dal presupposto che le variabili a cui eventualmente si vuole fare riferimento tramite un puntatore siano allocate in memoria.

[20] Per dereferenziare un puntatore si usa generalmente l'asterisco davanti al nome, pertanto il valore a cui punta la variabile ***p*** è accessibile attraverso l'espressione ****p***. Tuttavia esiste un altro modo che viene chiarito a proposito dell'aritmetica dei puntatori, per cui lo stesso valore si raggiunge con l'espressione '***p*`[0]`'.

[21] In contesti particolari è ammissibile che ***argc*** sia pari a zero, a

indicare che non viene fornita alcuna informazione; oppure, se gli argomenti vengono forniti ma il nome del programma è assente, ***argv[0][0]*** deve essere pari a *<NUL>*, ovvero al carattere nullo.

[22] L'indirizzo gestito da un puntatore a una funzione, riguarda potenzialmente uno «spazio di indirizzamento» differente rispetto a quello usato per le variabili. Per esempio, il puntatore ***p1***, riferito a una certa funzione, potrebbe avere lo stesso contenuto numerico del puntatore ***p2*** riferito a una variabile, ma nella memoria reale, i due puntatori raggiungerebbero posizioni differenti. Ciò serve per comprendere che la gestione dei puntatori alle funzioni non può essere confusa con quella dei dati, perché riguarda domini di indirizzamento diversi.

[23] È improbabile che sia utilizzato un tipo '`void *`', perché non sarebbe possibile scandire la memoria, salvo convertirlo ogni volta in un formato '`char *`'.

[24] **Unproto** software libero

Capitolo 67

Gestione dei flussi di file in C

67.1 Concetti generali 822
67.1.1 Dal file al flusso di file 822
67.1.2 File di testo e file binari 826
67.1.3 Fine del file 828
67.1.4 Memoria tampone 829
67.1.5 Flussi standard 830
67.1.6 Orientamento 831
67.2 Utilizzo comune dei file 832
67.2.1 Apertura e chiusura 832
67.2.2 Lettura e scrittura 835
67.2.3 Indicatore interno al file 840
67.2.4 File di testo 843
67.2.5 I/O standard 845
67.2.6 Ridirezione 846
67.2.7 Controllo degli errori 847
67.3 Conversione di input e output 850
67.3.1 Composizione dell'output 851
67.3.2 Rappresentazione degli specificatori di composizione per l'emissione dei dati 853
67.3.3 Funzioni per la composizione dell'output 860
67.3.4 Concatenamento di stringhe 861
67.3.5 Interpretazione dell'input 862

67.3.6 Rappresentazione degli specificatori di conversione 866

67.3.7 Funzioni per l'interpretazione dell'input 869

67.4 Riferimenti 871

EOF 828 errno 847 fclose() 832 fgets() 843 FILE 822 832 fopen() 832 fputs() 843 fread() 835 fseek() 840 ftell() 840 fwrite() 835 printf() 860 puts() 843 reopen() 846 scanf() 869 stderr 845 stdio 845 stdio.h 822 stdout 845 vprintf() 860 vscanf() 869 WEOF 828 %+... 853 %...c 853 866 %...d 853 866 %...e 853 866 %...f 853 866 %...g 866 %...hd 853 866 %...hhd 866 %...hhi 866 %...hho 866 %...hhu 866 %...hhx 866 %...hi 853 866 %...ho 853 866 %...hu 853 866 %...hx 853 866 %...i 853 866 %...lc 853 866 %...ld 853 866 %...Le 853 866 %...Lf 853 866 %...Lg 866 %...li 866 %...lld 853 866 %...lli 866 %...llo 853 866 %...llu 853 866 %...llx 853 866 %...lo 853 866 %...ls 853 866 %...lu 853 866 %...lx 853 866 %...o 853 866 %...s 853 866 %...u 853 866 %...x 853 866 %0... 853 %-... 853

67.1 Concetti generali

«

Il linguaggio C ha un proprio modo per gestire i file che, per poter essere compreso, richiede l'introduzione di alcuni concetti, presentati in questo capitolo. Va osservato che lo standard del linguaggio C prevede i flussi di file, i quali però, in un sistema che si rifà al modello di Unix sono gestiti attraverso i descrittori di file. Per scrivere codice C che sia compatibile nel modo migliore con qualunque sistema operativo, occorre avvalersi soltanto dei flussi, a cui qui ci si riferisce.

67.1.1 Dal file al flusso di file

Dal punto di vista del programma scritto in linguaggio C, il file viene utilizzato in qualità di ***flusso logico di dati*** (*stream*), ovvero flusso di file. Per la precisione, un file viene aperto attribuendogli un puntatore che rappresenta il flusso di file relativo; quando poi il flusso viene chiuso, l'associazione con il file si conclude.

La gestione del flusso di file avviene in modo trasparente, con l'ausilio di funzioni standard, ma ciò implica la presenza di una sorta di tabellina contenente una serie di informazioni legate all'accesso al file. Questa tabellina è formata in modo differente, a seconda del contesto in cui ci si trova a compilare il programma, ma in generale dovrebbe contenere almeno alcune informazioni basilari: il riferimento a un array di caratteri usato in qualità di memoria tampone, assieme ai vari puntatori necessari alla sua gestione; il tipo di accesso al file; i riferimenti per accedere al file secondo le caratteristiche del sistema operativo.

Quella tabellina che raccoglie tutte le informazioni su un certo flusso di file è definita da una variabile strutturata, dalla quale deriva un tipo di dati dichiarato nel file di intestazione '`stdio.h`'. Il tipo di dati in questione è denominato '**`FILE`**'.

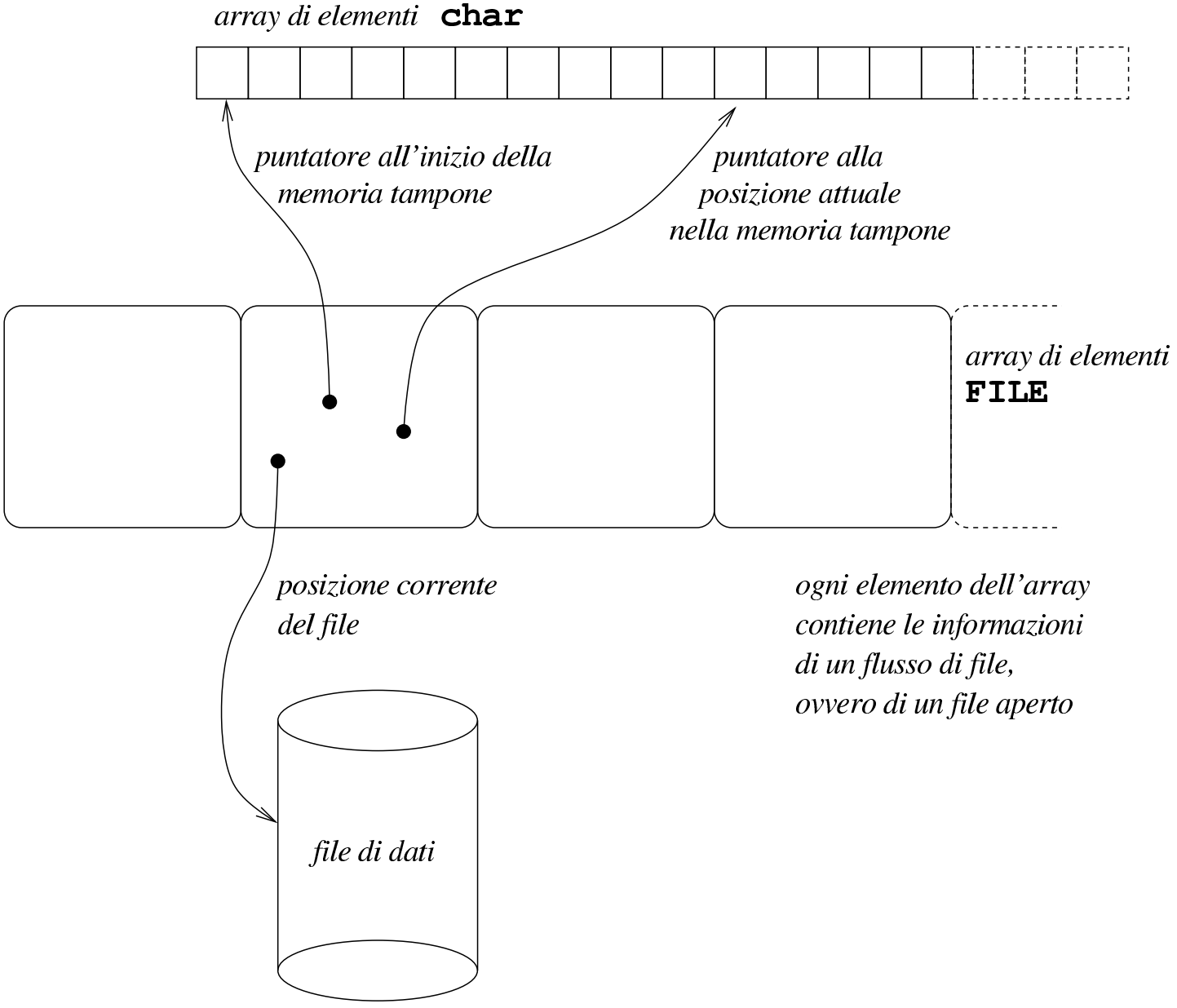
L'apertura di un file, attraverso le funzioni standard, coincide con l'ottenimento di un puntatore al tipo '**`FILE`**'; pertanto, questo puntatore rappresenta il flusso di file e tutti i riferimenti a tale flusso si fanno con quel puntatore.

La modalità di accesso al file distingue tra lettura, scrittura e scrittura in aggiunta, utilizzando una simbologia particolare per esprimerla. Lo specchietto successivo sintetizza le operazioni consentite in base

alla modalità utilizzata:[1]

`r`	`w`	`a`	`r+`	`w+`	`a+`	Annotazioni
X			X			Per aprire un file in lettura, con queste modalità, è necessario che esista già.
	X			X		Quando si apre un file in scrittura, con queste modalità, se il file non esiste viene creato al volo, se invece esiste già, il suo contenuto precedente viene eliminato.
X			X	X	X	Con queste modalità è possibile leggere il contenuto del file.
	X	X	X	X	X	Con queste modalità è possibile modificare il contenuto del file.
		X			X	Con queste modalità è possibile scrivere nel file soltanto aggiungendo dati in coda.

Figura 67.2. Rappresentazione intuitiva dell'associazione tra una variabile strutturata di tipo '**FILE**' e il file a cui fa riferimento. Qui viene ipotizzato un array di elementi di tipo '**FILE**', ma non è detto che l'organizzazione della libreria standard che si utilizza sia conforme a questa organizzazione.



67.1.2 File di testo e file binari

«

Il linguaggio C nasce per il sistema Unix, dove il file di testo ha una conformazione particolare che non è condivisa universalmente. Il file di testo in un sistema Unix o derivato è composto da una sequenza di caratteri (tradotti in byte),[2] dove la separazione tra le righe è segnalata dal codice *new-line*, corrispondente a <*LF*>, ovvero la sequenza '**\n**'.

Nei sistemi Dos e MS-Windows si ha una rappresentazione simile, dove però il codice di interruzione di riga è rappresentato dalla sequenza <*CR*><*LF*>. In altri sistemi si usano codice di interruzione di riga differenti e sono ammissibili forme molto diverse per rappresentare un file di testo.

Per questa ragione, il linguaggio C distingue l'accesso ai file attraverso due tipologie fondamentali: file di testo e file binari. In questo modo, quando si prevede un accesso in modalità testuale, la lettura e la scrittura del file avvengono attraverso una mediazione, tale da consentire al programmatore di trattare il file come se avesse la stessa rappresentazione di un sistema Unix. Naturalmente, in un sistema Unix e in qualunque altro sistema equivalente e conforme alla tradizione, non c'è distinzione tra l'accesso testuale ai file e quello binario.

Da quanto esposto vanno considerate due cose: quando si interviene su un file di testo, il codice corrispondente alla sequenza '**\n**' va inteso genericamente come codice di interruzione di riga; inoltre, il modo in cui si tiene traccia della posizione corrente all'interno di un file di testo non è predeterminabile, soprattutto perché non si può sapere quanti byte separano la fine di una riga dall'inizio della

successiva.

Il testo seguente è citato dalla documentazione standard *ISO/IEC 9899:TC2* e può servire per comprendere meglio il significato attribuito ai concetti di file di testo e di file binario:

> *A text stream is an ordered sequence of characters composed into lines, each line consisting of zero or more characters plus a terminating new-line character. Whether the last line requires a terminating new-line character is implementation-defined. Characters may have to be added, altered, or deleted on input and output to conform to differing conventions for representing text in the host environment. Thus, there need not be a one- to-one correspondence between the characters in a stream and those in the external representation. Data read in from a text stream will necessarily compare equal to the data that were earlier written out to that stream only if: the data consist only of printing characters and the control characters horizontal tab and new-line; no new-line character is immediately preceded by space characters; and the last character is a new-line character. Whether space characters that are written out immediately before a new-line character appear when read in is implementation-defined.*
>
> *A binary stream is an ordered sequence of characters that can transparently record internal data. Data read in from a binary stream shall compare equal to the data that were earlier written out to that stream, under the same implementation. Such a stream may, however, have an implementation-defined number of null characters appended to the end of the stream.*

67.1.3 Fine del file

«

Nei documenti che trattano del linguaggio C si fa spesso riferimento alla macro-variabile ***EOF*** (dichiarata nel file '`stdio.h`'), in qualità di valore che si ottiene quando si tenta di leggere oltre la fine del file. La macro-variabile ***EOF*** corrisponde a un valore negativo che solitamente è −1, trattato come intero normale. Generalmente si può ottenere un valore di questo genere quando la lettura avviene carattere per carattere (inteso nel senso del tipo '**`char`**', corrispondente al byte), perché in questi casi il carattere letto viene convertito in un valore senza segno, esteso alla dimensione di un intero normale. In questo modo, nessun carattere potrebbe confondersi con un valore negativo di un intero di tipo '**`int`**'.

Quando però la lettura di un file avviene attraverso funzioni che leggono un carattere esteso alla volta (l'equivalente di un carattere '**`wchar_t`**'), queste restituiscono un valore di tipo differente ('**`wint_t`**') con cui si può rappresentare sia un carattere esteso, sia il valore rappresentato dalla macro-variabile ***WEOF*** che non individua alcun carattere esteso e rappresenta il raggiungimento della fine del file. A differenza di ***EOF***, il valore di ***WEOF*** potrebbe essere positivo o negativo indifferentemente, perché conta solo che si tratti di un valore che non corrisponde ad alcun carattere esteso.

Di norma, il raggiungimento della fine di un file viene annotato all'interno della variabile strutturata che controlla il flusso (a cui ci si riferisce con un puntatore di tipo '**`FILE *`**') e può essere interrogata con una funzione apposita. Naturalmente, l'uso di una funzione che porti alla modifica della posizione corrente, va ad azzerare tale indicazione.

67.1.4 Memoria tampone

I flussi di file possono disporre di una memoria tampone (*buffer*) che di norma è costituita da un array di caratteri ed è gestita da puntatori annotati all'interno delle variabili strutturate di tipo '**`FILE`**' associate ai flussi stessi.

Il programmatore ha la possibilità di controllare l'uso della memoria tampone, definendone la dimensione o arrivando a escluderla del tutto. In particolare, se si utilizza la memoria tampone, si può distinguere tra una gestione completa e una gestione a righe di testo.

L'uso della memoria tampone implica che le operazioni di scrittura possono avvenire con un certo ritardo. In generale, alla chiusura di un flusso di file si ottiene anche lo scarico della memoria tampone per ciò che riguarda le operazioni di scrittura ancora sospese; eventualmente è disponibile anche una funzione per richiedere espressamente l'esecuzione della scrittura in qualunque altro momento.

Va osservato che gli accessi ai file si prevedono in modo esclusivo; pertanto la gestione della memoria tampone è interna al programma. Per un accesso condiviso ai file la memoria tampone non può essere usata e comunque occorrono delle accortezze che le funzioni standard non possono offrire.

67.1.5 Flussi standard

«

Il linguaggio C prevede che ogni programma disponga, in modo predefinito, di tre flussi di file già costituiti: standard input, standard output e standard error. Il primo è predisposto per la lettura e di norma è collegato alla tastiera; il secondo e il terzo consentono solo la scrittura e sono collegati normalmente allo schermo.

Il fatto di disporre di tre flussi già in essere implica che ci siano tre puntatori di tipo '`FILE *`' già predisposti e associati correttamente alle strutture rispettive, per il controllo dei flussi di competenza. Va osservato che mentre i flussi standard non possono essere costituiti esplicitamente, potrebbero invece essere chiusi, oppure potrebbero essere riassegnati associandoli a file (o dispositivi) differenti.

L'associazione iniziale dei flussi standard a file o dispositivi dipende da ciò che succede in fase di avvio del programma (una shell potrebbe ridirigere i flussi a file diversi da quelli consueti). In condizioni normali, lo standard error è privo di memoria tampone, perché ciò che viene segnalato attraverso questo canale deve essere recepito il più presto possibile; per quanto riguarda invece gli altri due flussi, se questi non sono associati a dispositivi interattivi, di norma sono provvisti di memoria_tampone.

Rimane da chiarire in che modo il file corrispondente al flusso sia aperto: l'associazione a una modalità di accesso binaria o testuale dovrebbe dipendere dal contesto e precisamente da ciò che determina il sistema operativo. È comunque possibile cambiare espressamente tale modalità, nel caso ciò fosse auspicabile.

67.1.6 Orientamento

I dati scritti e letti da un file vengono gestiti sempre attraverso sequenze di byte. Quando si devono rappresentare «caratteri estesi», tali da non poter essere espressi in un solo byte, si usano delle sequenze multibyte, secondo una codifica che normalmente dipende dalla configurazione locale.

La codifica multibyte utilizzata può essere priva di stato, in quanto ogni carattere esteso ha la propria sequenza indipendente, oppure può richiedere, di volta in volta, la selezione di un sottoinsieme di caratteri differente (attraverso quello che viene chiamato *shift state*). In ogni caso, sia la scrittura, sia la lettura, richiede di tenere traccia dello stato di completamento e, se necessario, della modalità di interpretazione in corso (*shift state*). Queste informazioni possono essere raccolte in un'area di memoria organizzata secondo il tipo '**`mbstate_t`**' (*Multibyte state*) che di solito è strutturata in più componenti.

Nella variabile strutturata di tipo '**`FILE`**' che rappresenta un flusso aperto, usata per gestire l'accesso al file relativo, deve essere presente un componente di tipo '**`mbstate_t`**' per poter seguire lo stato di interpretazione di una sequenza multibyte.

Onde evitare confusione, un flusso di file (aperto in modo binario o testuale, indifferentemente), deve essere ***orientato***, nel senso che occorre stabilire se vada gestito a caratteri normali o estesi. In mancanza di una dichiarazione esplicita, l'orientamento viene definito in base all'uso del flusso attraverso funzioni specializzate per il trattamento di stringhe normali o di stringhe estese. Per esempio, si ottiene un orientamento orientato al byte (*byte-oriented*) se si utilizza la

funzione ***fprintf()*** (*file print formatted*), mentre si ottiene un orientamento esteso (*wide-oriented*) se si usa la funzione ***fwprintf()*** (*file wide print formatted*).

Una volta impostato l'orientamento, anche solo attraverso l'uso iniziale di una funzione invece di un'altra, questo può essere cambiato solo in modo esplicito, eventualmente riaprendo il flusso. Ma se questo cambiamento esplicito non viene eseguito, non è possibile utilizzare il flusso attraverso funzioni che non siano conformi all'orientamento esistente.

Si osservi che anche i tre flussi standard, all'inizio dell'esecuzione del programma, sono ancora privi di orientamento.

67.2 Utilizzo comune dei file

«

Nel linguaggio C, i file aperti sono flussi di file e l'apertura coincide con la predisposizione automatica di una variabile strutturata di tipo '**FILE**', a cui, di conseguenza, si fa riferimento attraverso un puntatore (di tipo '**FILE ***'). Di solito, questo puntatore viene chiamato discorsivamente «puntatore al file», ovvero *file pointer*.

Quando si vuole accedere a un file, così come per poter usare le funzioni che consentono l'input e l'output elementare, è necessario includere il file '`stdio.h`', dove, tra l'altro, è dichiarato il tipo '**FILE**'.

67.2.1 Apertura e chiusura

«

L'apertura dei file viene ottenuta normalmente con la funzione ***fopen()*** che restituisce il puntatore al file, oppure il puntatore nullo, '**NULL**', in caso di fallimento dell'operazione. L'esempio seguen-

te mostra l'apertura del file '`mio_file`' contenuto nella directory corrente, con una modalità di accesso in sola lettura.

```
#include <stdio.h>
…
int main (void)
  {
    FILE *fp_mio_file;
    …
    fp_mio_file = fopen ("mio_file", "r");
    …
  }
```

Come si vede dall'esempio, è normale assegnare il puntatore ottenuto a una variabile adatta, che da quel momento identifica il file, finché questo resta aperto.

La chiusura del file avviene in modo analogo, attraverso la funzione ***fclose()***, che restituisce zero se l'operazione è stata conclusa con successo, oppure il valore rappresentato da ***EOF***. L'esempio seguente ne mostra l'utilizzo.

```
…
    fclose (fp_mio_file);
…
```

La chiusura del file conclude l'attività con questo, dopo avere scritto tutti i dati eventualmente ancora rimasti in sospeso (se il file è stato aperto in scrittura).

Normalmente, un file aperto viene definito come flusso di file, o *stream*; così, nello stesso modo viene identificata la variabile puntatore che vi si riferisce. In effetti, lo stesso file potrebbe anche essere aperto più volte con puntatori differenti, quindi è corretto distinguere tra file fisici su disco e file aperti, o flussi.

Seguono gli schemi sintattici di ***fopen()*** e ***fclose()***, in forma di prototipo di funzione:

```
FILE *fopen (char *file, char *modalità);
```

```
int fclose (FILE *flusso_di_file);
```

La funzione ***fopen()*** richiede come secondo argomento una stringa contenente l'informazione della modalità di accesso. Questa può essere composta utilizzando i simboli seguenti, dove la lettera '**b**' richiede espressamente un accesso binario, mentre la mancanza di tale lettera indica un accesso con le convenzioni dei file di testo:

Stringa	Descrizione
`r` `rb`	apre il file in sola lettura, posizionandosi all'inizio del file;
`r+` `rb+` | `r+b`	apre il file in lettura e scrittura, posizionandosi all'inizio del file;

Stringa	Descrizione
`w` `wb`	apre il file in sola scrittura, creandolo se necessario, o troncandone a zero il suo contenuto se questo esiste già;
`w+` `wb+` | `w+b`	apre il file in scrittura e lettura, creandolo se necessario, o troncandone a zero il suo contenuto se questo esiste già;
`a` `ab`	apre il file in scrittura in aggiunta (*append*), creandolo se necessario, o aggiungendovi dati a partire dalla fine e, di conseguenza, posizionandosi alla fine dello stesso;
`a+` `ab+` | `a+b`	apre il file in scrittura in aggiunta e in lettura, creandolo se necessario, o aggiungendovi dati a partire dalla fine e, di conseguenza, posizionandosi alla fine dello stesso.

La funzione ***fclose()*** restituisce zero in caso di successo, oppure il valore corrispondente alla macro-variabile ***EOF*** (annotando anche un valore appropriato nella variabile ***errno***).

67.2.2 Lettura e scrittura

L'accesso al contenuto dei file avviene generalmente a livello di byte e le operazioni di lettura e scrittura dipendono da un indicatore riferito a una posizione, espressa in byte, del contenuto del file stesso. Naturalmente, tale indicatore fa parte delle informazioni che si conservano nella variabile strutturata di tipo '**`FILE`**', a cui si fa riferimento per identificare il flusso di file.

A seconda di come viene aperto il file, questo indicatore viene posizionato nel modo più logico, come descritto a proposito della fun-

zione ***fopen()***. Questo indicatore viene spostato automaticamente a seconda delle operazioni di lettura e scrittura che si compiono, tuttavia, quando si passa da una modalità di accesso all'altra, è necessario spostare l'indicatore attraverso le istruzioni opportune, in modo da non creare ambiguità.

Per la lettura generica di un file in modo binario (nel senso di una lettura tale e quale del file) si può usare la funzione ***fread()*** che legge una quantità di byte trattandoli come un array. Per la precisione, si tratta di definire la dimensione di ogni elemento, espressa in byte, quindi la quantità di tali elementi. Il risultato della lettura viene inserito in un array, i cui elementi hanno la stessa dimensione. Si osservi l'esempio seguente:

```
…
    char ca[100];
    FILE *fp;
    int i;
    …
    i = fread (ca, 1, 100, fp);
    …
```

In questo modo si intende leggere 100 elementi della dimensione di un solo byte, collocandoli nell'array ***ca***, organizzato nello stesso modo. Naturalmente, non è detto che la lettura abbia successo, o quantomeno non è detto che si riesca a leggere la quantità di elementi richiesta. Il valore restituito dalla funzione rappresenta la quantità di elementi letti effettivamente. Se si verifica un qualsiasi tipo di errore che impedisce la lettura, la funzione si limita a restituire zero.

Quando il file viene aperto in lettura, l'indicatore interno viene posizionato all'inizio del file; quindi, ogni operazione di lettura sposta

in avanti il puntatore, in modo che la lettura successiva avvenga a partire dalla posizione immediatamente seguente:

```
…
     char ca[100];
     FILE *fp;
     int i;
     …
     fp = fopen ("mio_file", "rb");
     …
     while (1)          /* Ciclo senza fine */
       {
         i = fread (ca, 1, 100, fp);
         if (i == 0)
           {
             break;       /* Termina il ciclo */
           }
         …
       }
     …
```

In questo modo, come mostra l'esempio, viene letto tutto il file a colpi di 100 byte alla volta, tranne l'ultima in cui si ottiene solo quello che resta da leggere.

Analogamente, la scrittura può essere eseguita con la funzione ***fwrite()*** che scrive una quantità di byte trattandoli come un array, nello stesso modo già visto con la funzione ***fread()***. La scrittura procede a partire dalla posizione corrente riferita al file.

```
…
    char ca[100];
    FILE *fp;
    int i;
    …
    i = fwrite (ca, 1, 100, fp);
    …
```

L'esempio, come nel caso di ***fread()***, mostra la scrittura di 100 elementi di un solo byte, prelevati da un array. Il valore restituito dalla funzione è la quantità di elementi che sono stati scritti con successo. Se si verifica un qualsiasi tipo di errore che impedisce la scrittura, la funzione si limita a restituire zero.

Anche in scrittura è importante l'indicatore della posizione interna del file. Di solito, quando si crea un file o lo si estende, l'indicatore si trova sempre alla fine. L'esempio seguente mostra lo scheletro di un programma che crea un file, copiando il contenuto di un altro (non viene utilizzato alcun tipo di controllo degli errori).

```
#include <stdio.h>
…
int main (void)
{
    char ca[1024];
    FILE *fp_in;
    FILE *fp_out;
    int i;
    …
    fp_in = fopen ("file_in", "r");
    …
    fp_out = fopen ("file_out", "w");
    …
    while (1)                    // Ciclo senza fine.
```

```
        {
          i = fread (ca, 1, 1024, fp_in);
          if (i == 0)
            {
              break;                    // Termina il ciclo.
            }
          …
          fwrite (ca, 1, i, fp_out);
          …
        }
      …
      fclose (fp_in);
      fclose (fp_out);
      …
      return 0;
}
```

Seguono i modelli sintattici di ***fread()*** e ***fwrite()***, espressi in forma di prototipi di funzione:

```
size_t fread (void *restrict ptr,
              size_t dimensione,
              size_t quantità,
              FILE *restrict stream);
```

```
size_t fwrite (const void *restrict ptr,
               size_t dimensione,
               size_t quantità,
               FILE *stream);
```

Il tipo di dati '**`size_t`**' serve a garantire la compatibilità con qualun-

que tipo intero, mentre il tipo '**void**' per l'array permette l'utilizzo di qualunque tipo per i suoi elementi, anche se negli esempi è sempre stato visto il trattamento di sole sequenze di byte.

67.2.3 Indicatore interno al file

«

Lo spostamento diretto dell'indicatore interno della posizione di un file aperto è un'operazione necessaria quando il file è stato aperto simultaneamente in lettura e in scrittura, e da un tipo di operazione si vuole passare all'altro. Per questo si utilizza la funzione ***fseek()*** ed eventualmente anche ***ftell()*** per conoscere la posizione attuale. La posizione e gli spostamenti sono espressi in byte.

La funzione ***fseek()*** esegue lo spostamento a partire dall'inizio del file, oppure dalla posizione attuale, oppure dalla posizione finale. Per questo utilizza un parametro che può avere tre valori identificati rispettivamente da tre macro-variabili: ***SEEK_SET***, ***SEEK_CUR*** e ***SEEK_END***. l'esempio seguente mostra lo spostamento del puntatore, riferito al flusso di file ***fp***, in avanti di 10 byte, a partire dalla posizione attuale.

```
…
i = fseek (fp, 10, SEEK_CUR);
…
```

La funzione ***fseek()*** restituisce zero se lo spostamento avviene con successo, altrimenti si ottiene un valore negativo.

L'esempio seguente mostra lo scheletro di un programma, senza controlli sugli errori, che, dopo aver aperto un file in lettura e scrittura, lo legge a blocchi di dimensioni uguali, modifica questi blocchi e li riscrive nel file.

```
#include <stdio.h>

static const int dim = 100; // Dimensione del record logico.

int main (void)
{
    char ca[dim];
    FILE *fp;
    int qta;
    int posizione_1;
    int posizione_2;

    fp = fopen ("mio_file", "r+b");  // Lettura e scrittura.

    while (1)                        // Ciclo senza fine.
      {
        //
        // Salva la posizione del puntatore interno al file
        // prima di eseguire la lettura.
        //
        posizione_1 = ftell (fp);
        qta = fread (ca, 1, dim, fp);

        if (qta == 0)
          {
            break;                   // Termina il ciclo.
          }
        //
        // Salva la posizione del puntatore interno al file
        // dopo la lettura.
        //
        posizione_2 = ftell (fp);
        //
        // Sposta il puntatore alla posizione precedente
```

```
            // alla lettura.
            //
            fseek (fp, posizione_1, SEEK_SET);
            //
            // Esegue qualche modifica nei dati, per esempio
            // mette un punto esclamativo all'inizio.
            //
            ca[0] = '!';
            //
            // Riscrive il record modificato.
            //
            fwrite (ca, 1, qta, fp);
            //
            // Riporta il puntatore interno al file alla
            // posizione corretta per eseguire la lettura
            // successiva.
            //
            fseek (fp, posizione_2, SEEK_SET);
          }

        fclose (fp);
        return 0;
}
```

Segue il modello sintattico per l'uso della funzione ***fseek()***, espresso attraverso il suo prototipo:

```
int fseek (FILE *stream, long int spostamento,
           int punto_di_partenza);
```

Il valore dello spostamento, fornito come secondo parametro, rappresenta una quantità di byte che può essere anche negativa, indicando in tal caso un arretramento dal punto di partenza. Il valore restitui-

to da ***fseek()*** è zero se l'operazione viene completata con successo, altrimenti viene restituito un valore diverso.

Segue il modello sintattico per l'uso della funzione ***ftell()***, espresso attraverso il suo prototipo:

```
long int ftell (FILE *stream)
```

La funzione ***ftell()*** permette di conoscere la posizione dell'indicatore interno al file a cui fa riferimento il flusso di file fornito come parametro. Se si tratta di un file per il quale si esegue un accesso binario, la posizione ottenuta è assoluta, ovvero riferita all'inizio del file.

Il valore restituito in caso di successo è positivo, a indicare appunto la posizione dell'indicatore. Se si verifica un errore viene restituito un valore negativo: -1.

67.2.4 File di testo

I file di testo possono essere gestiti in modo più semplice attraverso due funzioni: ***fgets()*** e ***fputs()***. Queste permettono rispettivamente di leggere e scrivere un file una riga alla volta, intendendo come riga una porzione di testo che termina con il codice di interruzione di riga, secondo l'astrazione usata dal linguaggio.

La funzione ***fgets()*** permette di leggere una riga di testo di una data dimensione massima. Si osservi l'esempio seguente:

```
…
fgets (ca, 100, fp);
…
```

In questo caso, viene letta una riga di testo di una dimensione massima di 99 caratteri, dal file rappresentato dal puntatore ***fp***. Questa riga viene posta all'interno dell'array ***ca***, con l'aggiunta di un carattere '`\0`' finale. Questo fatto spiega il motivo per il quale il secondo parametro corrisponde a 100, mentre la dimensione massima della riga letta è di 99 caratteri. In pratica, l'array di destinazione è sempre una stringa, terminata correttamente.

Nello stesso modo funziona ***fputs()***, che però richiede solo la stringa e il puntatore del file da scrivere. Dal momento che una stringa contiene già l'informazione della sua lunghezza perché possiede un carattere di conclusione, non è prevista l'indicazione della quantità di elementi da scrivere.

```
…
fputs (ca, fp);
…
```

Seguono i modelli sintattici delle funzioni ***fputs()*** e ***fgets()***, in forma di prototipi di funzione:

```
char *fgets (char *stringa, int dimensione_max, FILE *stream);
```

```
int fputs (const char *stringa, FILE *stream)
```

Se l'operazione di lettura riesce, ***fgets()*** restituisce un puntatore corrispondente alla stessa stringa (cioè l'array di caratteri di destinazione), altrimenti restituisce il puntatore nullo, '**`NULL`**', per esempio quando è già stata raggiunta la fine del file.

La funzione ***fputs()*** permette di scrivere una stringa in un file di

testo. La stringa viene scritta senza il codice di terminazione finale, '`\0`', ma anche senza aggiungere il codice di interruzione di riga. Il valore restituito è un valore positivo in caso si successo, altrimenti ***EOF***.

In alternativa a ***fgets()*** e a ***fputs()*** si possono considerare anche le funzioni ***gets()*** e ***puts()***, le quali però utilizzano rispettivamente lo standard input e lo standard output. Ma la funzione ***gets()*** legge tutto quello che trova fino alla fine della riga o, in mancanza di questo, fino alla fine del file, mentre ***puts()*** **aggiungere automaticamente il codice di interruzione di riga** alla fine della stringa che viene scritta nel file.

```
char *gets (char *stringa);
```

```
int puts (const char *stringa)
```

67.2.5 I/O standard

Ci sono tre flussi di file che risultano aperti in modo predefinito, all'avvio del programma:

- standard input, corrispondente normalmente alla tastiera;
- standard output, corrispondente normalmente allo schermo del terminale;
- standard error, anch'esso corrispondente normalmente allo schermo del terminale.

Spesso si utilizzano questi flussi di file attraverso funzioni apposite (come nel caso di ***gets()*** e ***puts()***) che vi fanno riferimento in modo

implicito, ma si potrebbe accedere anche attraverso funzioni generalizzate, utilizzando come puntatori i nomi: '**stdio**', '**stdout**' e '**stderr**'.

67.2.6 Ridirezione

«

È possibile associare un flusso di file già in essere, a un file differente, attraverso la funzione ***freopen()***, oppure è possibile modificarne la modalità di accesso. Evidentemente questo tipo di operazione richiede la chiusura del flusso di file, prima di associarvi un file differente o di cambiare la modalità, cosa che comunque tenta di eseguire automaticamente la stessa funzione ***freopen()***:

```
FILE *freopen (const char *restrict nome_file_nuovo,
               const char *restrict modalità_di_accesso,
               FILE *restrict flusso di file);
```

La funzione, se riesce a eseguire il proprio compito, restituisce il puntatore allo stesso flusso di file indicato come terzo argomento, ovvero quello a cui viene applicata la ridirezione o la modifica dei permessi (o entrambe le cose). Per limitare l'effetto alla sola modifica della modalità di accesso, è sufficiente indicare il puntatore nullo al posto del nome del file. Viene mostrato un esempio che ridirige lo standard output:

```
#include <stdio.h>
int main (void)
{
    printf ("ciao 1\n");
    freopen ("mio", "w", stdout);
    printf ("ciao 2\n");
    freopen ("/dev/tty", "w", stdout);
    printf ("ciao 3\n");
    return 0;
}
```

In questo caso, dal momento che la funzione ***printf()*** scrive automaticamente attraverso lo standard output, quando il flusso di file '**`stdout`**' viene ridiretto nel file '`mio`', il testo '**`ciao 2`**' viene scritto in tale file. Ipotizzando di operare in un sistema Unix o in un sistema equivalente, il file di dispositivo '`/dev/tty`' dovrebbe corrispondere allo schermo del terminale utilizzato in quel momento (anche se fosse un terminale grafico); pertanto, il messaggio '**`ciao 3`**' dovrebbe apparire nuovamente sullo schermo.

Logicamente, quando si riapre un file e si cambia la modalità, da binaria a testo o viceversa, può essere appropriato un riposizionamento, con l'aiuto di ***fseek()***.

67.2.7 Controllo degli errori

Molte funzioni, quando si verifica un errore, annotano quanto accaduto, in forma di numero intero, in una variabile globale nota con il nome ***errno***. In generale, il nome ***errno*** è un'espressione che si traduce nell'accesso, a un'area di memoria, condiviso dal programma, ed eventualmente distinto in base al thread, ovvero il flusso di controllo. Il significato del valore attribuito alla variabile ***errno*** è descritto da macro-variabili definite nel file '`errno.h`', nel qua-

le viene anche dichiarata la variabile ***errno***, o l'espressione che la rappresenta.

La lettura della variabile ***errno*** porta alla conoscenza dell'ultimo errore che si è presentato e non è previsto il suo azzeramento automatico.

La variabile strutturata che si utilizza per fare riferimento a un flusso di file prevede anche l'annotazione di uno stato di errore. In pratica, le funzioni che accedono ai file, oltre che aggiornare la variabile globale ***errno***, gestiscono l'indicazione di questo stato, azzerandolo quando non è più significativo. Per verificare la presenza di uno stato di errore ancora valido, a proposito di un flusso di file, si usa la funzione ***ferror()*** che restituisce un valore diverso da zero se questo stato esiste effettivamente:

```
int ferror (FILE *flusso_di_file);
```

Per interpretare l'errore annotato nella variabile ***errno*** e visualizzare direttamente un messaggio attraverso lo standard error, si può usare la funzione ***perror()***:

```
void perror (const char *s);
```

La funzione ***perror()*** mostra un messaggio in modo autonomo, aggiungendo davanti la stringa che può essere fornita come primo argomento (diversamente si può indicare il puntatore nullo o una stringa nulla, in quanto contenente solo il carattere di terminazione).

L'esempio seguente mostra un programma completo e molto semplice, in cui si crea un errore, tentando di scrivere un messaggio

attraverso lo standard input, cosa che produce un errore. Se effettivamente si rileva un errore associato a quel flusso di file, attraverso la funzione ***ferror()***, allora si passa alla sua interpretazione con la funzione ***perror()***.

Listato 67.15. Per provare il codice attraverso un servizio *pastebin*: *http://ideone.com/JvbUI* .

```
#include <stdio.h>
#include <errno.h>
int main (void)
{
    fprintf (stdin, "Ciao amore!\n");
    if (ferror (stdin))
      {
        perror ("Attenzione");
      }
    return 0;
}
```

Come si vede, è necessario includere anche il file ‘`errno.h`’, senza il quale la variabile ***errno*** non risulterebbe accessibile. Avviando questo programma in un sistema GNU/Linux si potrebbe ottenere il messaggio seguente:

```
Attenzione: Bad file descriptor
```

In alternativa alla funzione ***perror()*** si può usare anche ***strerror()*** (dal file ‘`string.h`’), con la quale si ottiene la stringa contenente il messaggio di errore:

```
char *strerror (int n_errore);
```

Si può modificare leggermente l’esempio già apparso, in modo da

usare la funzione ***strerror()*** per produrre lo stesso risultato.

Listato 67.17. Per provare il codice attraverso un servizio *pastebin*: *http://ideone.com/qDCYr* .

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
int main (void)
{
    char *cp;
    fprintf (stdin, "Ciao amore!\n");
    if (ferror (stdin))
      {
        cp = strerror (errno);
        fprintf (stderr, "Attenzione: %s\n", cp);
      }
    return 0;
}
```

67.3 Conversione di input e output

«

Il linguaggio C rappresenta in memoria i valori numerici in modo binario secondo una modalità diversa rispetto a quella usata per le stringhe che servono invece per l'interazione umana. In altri termini, un conto è il valore 100, un altro è la sequenza dei caratteri numerici con cui questo valore viene rappresentato sullo schermo o su carta.

Il linguaggio C non svolge automaticamente conversioni da valori numerici binari a stringhe di cifre numeriche e viceversa; per questo è necessario invece avvalersi di funzioni di conversione. Per la precisione esistono due gruppi di funzioni, ...***printf()*** e ...***scanf()***, con cui è possibile comporre (nel senso tipografico) le informazioni in uscita, oppure interpretarle in senso inverso le informazioni in ingresso.

67.3.1 Composizione dell'output

Le funzioni del gruppo ...***printf()*** consentono di comporre una stringa (da memorizzare o da visualizzare), partendo da un'altra stringa contenente il formato di composizione e utilizzando un elenco variabile di argomenti:

```
...printf ( ... stringa_di_composizione [, argomento] ... )
```

Il modello sintattico dà solo una visione di massima: a seconda della funzione ci possono essere dei parametri che non vengono chiariti nello schema, quindi appare sempre la stringa di composizione, la quale può essere seguita da altri argomenti le cui caratteristiche non sono precisate nel prototipo della funzione.[3]

La stringa di composizione è una stringa normale, in cui si inseriscono delle sequenze precedute dal simbolo '`%`', note come ***specificatori di conversione***. Conviene partire da un esempio, proprio con la funzione ***printf()***, la quale emette la stringa generata dalla composizione attraverso lo standard output (attraverso il flusso di file associato allo standard output):

```
...
printf ("Il capitale di %i al tasso %f%% dà l'interesse %i",
        1000, 0.5, 5);
...
```

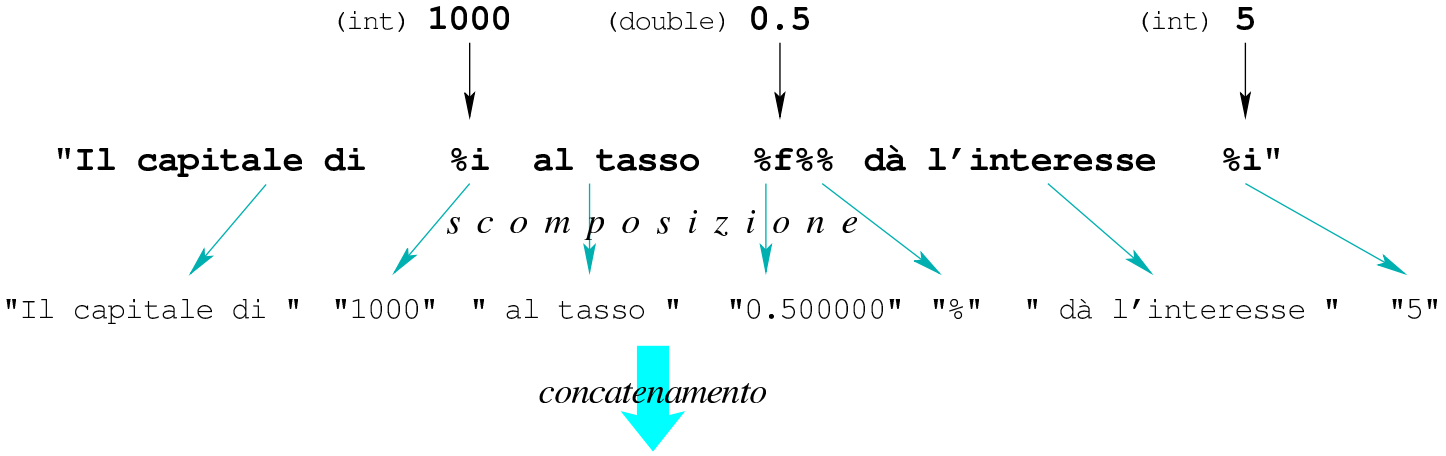
Da questa istruzione si ottiene la visualizzazione della frase seguente:

```
Il capitale di 1000 al tasso 0.500000% dà l'interesse 5
```

In pratica, al posto del primo specificatore '`%i`' è stato inserito il va-

lore 1000 dopo averlo convertito in modo da essere rappresentato da quattro caratteri (‘`'1'`’, ‘`'0'`’, ‘`'0'`’, ‘`'0'`’), al posto del secondo specificatore ‘`%f`’ è stato inserito il valore 0.5 dopo un’opportuna conversione in caratteri, al posto del terzo specificatore ‘`%%`’ è stato inserito un carattere di percentuale, infine, al posto del quarto specificatore ‘`%i`’ è stato inserito il valore 5.

Figura 67.20. Schematizzazione della trasformazione di una stringa di composizione in una stringa finale.



Il capitale di 1000 al tasso di 0.500000% dà l’interesse 5

Lo specificatore di conversione ha due compiti: indicare che tipo di informazione viene prelevato dagli argomenti (ammesso che si prelevi effettivamente un valore) e come questa deve essere rappresentata. Nel caso dell’esempio, il primo specificatore ‘`%i`’ indica che il valore da prelevare dagli argomenti è di tipo ‘`int`’; il secondo specificatore ‘`%f`’ indica un tipo ‘`double`’; il terzo non preleva alcun valore; il quarto indica ancora un altro ‘`int`’.

Una stringa di composizione che non contenga degli specificatori rimane evidentemente intatta e non richiede alcun dato aggiuntivo. La funzione ***printf()*** (che è stata usata nell’esempio) viene usata spesso come mezzo generico per emettere un messaggio attraverso lo stan-

dard output, anche quando non c'è alcun bisogno di comporre dei dati. Questo è lecito, ma non va dimenticato il contesto, pertanto, scrivere l'istruzione seguente sarebbe sbagliato:

```
…
printf ("Il capitale di 1000 al tasso 0.5% dà l'interesse 5");
…
```

Il modo giusto è quello seguente:

```
…
printf ("Il capitale di 1000 al tasso 0.5%% dà l'interesse 5");
…
```

67.3.2 Rappresentazione degli specificatori di composizione per l'emissione dei dati

Di norma, la scelta dello specificatore determina il tipo di dati dell'argomento e il tipo di trasformazione che deve ricevere. La tabella 67.23 elenca alcuni degli specificatori di conversione utilizzabili, nella loro forma più semplice. È bene ricordare che per rappresentare il simbolo di percentuale si usa uno specificatore fittizio composto dalla sequenza di due segni percentuali: '`%%`'.

Tabella 67.23. Alcuni specificatori di conversione.

Simbolo	Corrispondenza
`%…c`	Un carattere singolo.
`%…s`	Una stringa.
`%…d`	Un intero con segno da rappresentare in base dieci.

Simbolo	Corrispondenza
`%...u`	Un intero senza segno da rappresentare in base dieci.
`%...o`	Un intero senza segno da rappresentare in ottale.
`%...x`	Un intero senza segno da rappresentare in esadecimale.
`%...e`	Un numero a virgola mobile normale (‘**`double`**’), da rappresentare in notazione esponenziale.
`%...f`	Un numero a virgola mobile normale (‘**`double`**’), da rappresentare in notazione decimale fissa.

Leggendo la tabella si può osservare che la composizione dei dati in uscita può riguardare anche dati che sono già in forma di stringa (lo specificatore ‘**`%...s`**’), pertanto si usa questo metodo anche per il concatenamento delle stringhe.

Gli specificatori di conversione possono contenere indicazioni ulteriori tra il simbolo di percentuale e la lettera che definisce il tipo di trasformazione. Si tratta di inserire un simbolo composto da un carattere singolo, seguito eventualmente da altre informazioni aggiuntive, secondo la sintassi seguente:

```
% [simbolo] [n_ampiezza] [.n_precisione] [hh|h|l|ll|j|z|t|L] tipo
```

Alcuni di questi simboli sono rappresentati dalla tabella 67.24. In presenza di valori numerici, si può indicare il numero di cifre decimali intere (ampiezza), aggiungendo eventualmente il numero di decimali (precisione), se si tratta di rappresentare un numero a virgola mobile. Quando è necessario modificare il tipo di dati provenienti dagli argomenti, ciò può essere precisato con una sigla, come

descritto nella tabella 67.25.

Tabella 67.24. Alcuni simboli per la conversione di valori numerici.

Simbolo	Corrispondenza
`%+`...	Il segno «+», usato all'inizio di uno specificatore di conversione, fa sì che i numeri positivi siano rappresentati con il segno in modo esplicito, mentre altrimenti il segno viene mostrato solo per quelli negativi.
`%0`***ampiezza***... `%+0`***ampiezza***...	Lo zero, usato all'inizio di uno specificatore di conversione, fa sì che si inseriscano degli zeri per allineare a destra un valore numerico, nell'ambito dell'ampiezza specificata. Lo zero può combinarsi con il segno «+».
`%`***ampiezza***... `%+`***ampiezza***...	In mancanza di uno zero iniziale, in presenza dell'indicazione dell'ampiezza, il valore viene allineato a destra usando degli spazi.
`%-`... `%-+`...	Il segno meno richiede un allineamento a sinistra rispetto al campo, usando degli spazi a destra. Si può combinare con il segno «+», ma non con lo zero.

Tabella 67.25. Alcuni modificatori dell'estensione che ha in memoria il valore da estrarre e comporre.

Simbolo	Corrispondenza
`%...ld` `%...lu` `%...lo` `%...lx`	La lettera '**`l`**', prima di quella che specifica il tipo di composizione, quando si tratta della conversione di un numero intero, specifica che il dato va trattato in qualità di '**`long int`**', con o senza segno.

Simbolo	Corrispondenza
`%…lc` `%…ls`	La lettera '**`l`**', prima di quella che specifica il tipo di composizione, quando si tratta della conversione di un carattere o di una stringa, specifica che il dato va trattato in qualità di carattere esteso (*wide char*) o di stringa estesa (*wide string*).
`%…lld` `%…llu` `%…llo` `%…llx`	La coppia di lettere '**`ll`**', prima di quella che specifica il tipo di composizione, quando si tratta della conversione di un numero intero, specifica che il dato va trattato in qualità di '**`long long int`**', con o senza segno.
`%…Le` `%…Lf`	La lettera '**`L`**', prima di quella che specifica il tipo di composizione, quando si tratta della conversione di un valore in virgola mobile, specifica che il dato va trattato in qualità di '**`long double`**'.

Nella stringa di composizione possono apparire anche sequenze di escape come già mostrato nella tabella 66.17. Si veda anche la pagina di manuale *printf(3)*.

Tabella 67.26. Esempi di utilizzo degli specificatori di conversione di ***printf()***. Le costanti numeriche utilizzate negli esempi sono interpretate secondo le convenzioni del linguaggio, pertanto: 123 e −123 sono costanti di tipo '**`int`**'; mentre 123.456 e −123.456 sono costanti di tipo '**`double`**'.

Codice	Risultato emesso attraverso la funzione
`printf ("[%i]", 123);`	`[123]`
`printf ("[%i]", -123);`	`[-123]`

Codice	Risultato emesso attraverso la funzione
`printf ("[%2d]", 123);`	`[123]` L'indicatore '**`%2d`**' specifica che si devono usare almeno due cifre, ma se le cifre della parte intera sono in numero maggiore, queste vanno indicate tutte ugualmente.
`printf ("[%6d]", 123);`	`[   123]`
`printf ("[%6d]", -123);`	`[  -123]`
`printf ("[%+6d]", 123);`	`[  +123]`
`printf ("[%06d]", 123);`	`[000123]`
`printf ("[%06d]", -123);`	`[-00123]`
`printf ("[%+06d]", 123);`	`[+00123]`
`printf ("[%-6d]", 123);`	`[123   ]`
`printf ("[%u]", 123);`	`[123]`
`printf ("[%u]", -123);`	`[4294967173]` Evidentemente si ottiene la rappresentazione del valore binario, tale e quale, secondo la notazione usata per i valori negativi.
`printf ("[%6x]", 123);`	`[    7b]`
`printf ("[%06x]", 123);`	`[00007b]`
`printf ("[%x]", 123);`	`[7b]`

Codice	Risultato emesso attraverso la funzione
`printf ("[%x]", -123);`	`[ffffff85]` Evidentemente si ottiene la rappresentazione del valore binario, tale e quale, secondo la notazione usata per i valori negativi.
`printf ("[%6x]", 123);`	`[    7b]`
`printf ("[%06x]", 123);`	`[00007b]`
`printf ("[%o]", 123);`	`[173]`
`printf ("[%o]", -123);`	`[37777777605]` Evidentemente si ottiene la rappresentazione del valore binario, tale e quale, secondo la notazione usata per i valori negativi.
`printf ("[%6o]", 123);`	`[   173]`
`printf ("[%06o]", 123);`	`[000173]`
`printf ("[%f]", 123.456);`	`[123.456000]`
`printf ("[%f]", -123.456);`	`[-123.456000]`
`printf ("[%12f]", 123.456);`	`[  123.456000]`
`printf ("[%.4f]", 123.456);`	`[123.4560]`
`printf ("[%12.4f]", 123.456);`	`[    123.4560]`
`printf ("[%12.4f]", -123.456);`	`[   -123.4560]`
`printf ("[%+12.4f]", 123.456);`	`[   +123.4560]`
`printf ("[%012.4f]", 123.456);`	`[0000123.4560]`
`printf ("[%012.4f]", -123.456);`	`[-000123.4560]`
`printf ("[%+012.4f]", 123.456);`	`[+000123.4560]`

Codice	Risultato emesso attraverso la funzione
`printf ("[%-12.4f]", 123.456);`	`[123.4560    ]`
`printf ("[%e]", 123.456);`	`[1.234560e+02]`
`printf ("[%e]", -123.456);`	`[-1.234560e+02]`
`printf ("[%15e]", 123.456);`	`[   1.234560e+02]`
`printf ("[%.4e]", 123.456);`	`[1.2346e+02]`
`printf ("[%15.4e]", 123.456);`	`[     1.2346e+02]`
`printf ("[%15.4e]", -123.456);`	`[    -1.2346e+02]`
`printf ("[%+15.4e]", 123.456);`	`[    +1.2346e+02]`
`printf ("[%015.4e]", 123.456);`	`[000001.2346e+02]`
`printf ("[%015.4e]", -123.456);`	`[-00001.2346e+02]`
`printf ("[%+015.4e]", 123.456);`	`[+00001.2346e+02]`
`printf ("[%-15.4e]", 123.456);`	`[1.2346e+02     ]`
`printf ("[%s]", "ciao amore");`	`[ciao amore]`
`printf ("[%7s]", "ciao amore");`	`[ciao amore]` La stringa è più lunga di sette caratteri, ma viene visualizzata completamente.
`printf ("[%.7s]", "ciao amore");`	`[ciao am]` La stringa viene troncata se è più lunga del valore della precisione.
`printf ("[%.14s]", "ciao amore");`	`[ciao amore]`
`printf ("[%14s]", "ciao amore");`	`[    ciao amore]`
`printf ("[%14.7s]", "ciao amore");`	`[       ciao am]`
`printf ("[%-14s]", "ciao amore");`	`[ciao amore    ]`

Codice	Risultato emesso attraverso la funzione
`printf ("[%-14.7s]", "ciao amore");`	`[ciao am       ]`

67.3.3 Funzioni per la composizione dell'output

Tutte le funzioni standard il cui nome finisce per '**`printf`**' interpretano una stringa di composizione secondo le modalità descritte nel capitolo, ovvero in modo analogo a ***printf()*** che, in particolare, emette il risultato della composizione attraverso lo standard output. In particolare, la funzione ***fprintf()*** scrive il risultato attraverso il flusso di file che costituisce il parametro ***stream*** (il primo argomento) e la funzione ***sprintf()*** copia il risultato, come stringa, a partire dal puntatore ***s*** (sempre il primo argomento).

```
int printf (const char *restrict composizione, ...);
```

```
int fprintf (FILE *restrict stream,
             const char *restrict composizione, ...);
```

```
int sprintf (char *restrict s,
             const char *restrict composizione, ...);
```

```
int snprintf (char *restrict s,
              size_t n,
              const char *restrict composizione, ...);
```

Le funzioni di cui è appena stato mostrato il modello sintattico, leggono gli argomenti successivi alla stringa di composizione in base a quanto indicato con gli specificatori di composizione. Altre funzioni equivalenti, con il nome che inizia con la lettera «v», hanno bisogno di un puntatore di tipo '**va_list**':

```
int vprintf (const char *restrict composizione,
             va_list arg);
```

```
int vfprintf (FILE *restrict stream,
              const char *restrict composizione,
              va_list arg);
```

```
int vsprintf (char *restrict s,
              const char *restrict composizione,
              va_list arg);
```

```
int vsnprintf (char *restrict s,
               size_t n,
               const char *restrict composizione,
               va_list arg);
```

67.3.4 Concatenamento di stringhe

Il linguaggio C, di per sé, non agevola l'uso delle stringhe; al massimo si può contare sul fatto che una sequenza di stringhe letterali venga considerata una stringa sola, concatenata. Per il concatenamento

delle stringhe sono disponibili le funzioni ***strcat()*** e ***strncat()***, ma l'uso delle funzioni previste per la composizione dell'output è molto più comodo, considerata la facilità con cui si inseriscono anche dati diversi dalle stringhe.

Listato 67.27. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/Nzbovidr* , *http://ideone.com/Jgn8z* .

```
#include <stdio.h>
int main (void)
{
    char s[] = "Ciao amore";
    printf ("%s... Ti %s.\n", s, "voglio tanto bene");
    return 0;
}
```

L'esempio appena mostrato dovrebbe dimostrare questa maggiore facilità. Il messaggio che viene visualizzato è: «Ciao amore... Ti voglio tanto bene.»

67.3.5 Interpretazione dell'input

«

Quando un programma interagisce con l'essere umano, scambia dati in forma grafica, nel senso che un numero appare e viene inserito come sequenza di caratteri grafici. Così come per la rappresentazione umana dei dati si usano comunemente le funzioni ...***printf()***, per l'immissione dei dati si usano le funzioni ...***scanf()*** che hanno il ruolo opposto:

```
...scanf ( ... stringa_di_conversione [ , argomento ] ... )
```

Il modello sintattico dà solo una visione di massima: a seconda della

funzione ci possono essere dei parametri che non vengono chiariti nello schema, quindi appare sempre la stringa di conversione, la quale può essere seguita da altri argomenti costituiti da puntatori, le cui caratteristiche particolari non sono precisate nel prototipo della funzione.[4] Viene proposto un esempio con la funzione ***scanf()*** che riceve i dati in ingresso (da interpretare) dallo standard input:

```
…
printf ("Inserisci l'importo: ");
scanf ("%i", &i_importo);
…
```

Il pezzo di codice mostrato emette la frase seguente e resta in attesa dell'inserimento di un valore numerico intero, seguito da [*Invio*]:

```
Inserisci l'importo: _
```

Questo valore viene inserito nella variabile ***i_importo***. Si deve osservare il fatto che i parametri successivi alla stringa di conversione sono dei puntatori, per cui, avendo voluto inserire il dato nella variabile ***i_importo***, questa è stata indicata preceduta dall'operatore '**&**' in modo da fornire alla funzione l'indirizzo corrispondente (si veda la sezione 66.5 sulla gestione dei puntatori).

Con una stessa funzione di questo tipo è possibile inserire dati per diverse variabili, come si può osservare dall'esempio seguente, ma in questo caso, per ogni dato viene richiesta la pressione di [*Invio*] o l'inserimento di spazi tra un dato e l'altro.

```
…
printf ("Inserisci il capitale e il tasso: ");
scanf ("%i%f", &i_capitale, &i_tasso);
…
```

La stringa di conversione è il parametro più delicato di queste funzioni. Come visto negli esempi, una stringa del genere contiene principalmente degli specificatori di conversione che, come già accennato, si comportano in modo molto simile agli specificatori di composizione delle funzioni *...printf()*. Quello che segue è lo schema sintattico generale per la definizione di uno specificatore di conversione:

```
%[*] [n_ampiezza] [hh|h|l|ll|j|z|t|L] tipo
```

Come si può vedere, all'inizio è previsto un solo tipo di simbolo, costituito da un asterisco, il cui scopo è quello di annullare l'assegnamento del valore a una variabile. In pratica, con l'asterisco il dato corrispondente allo specificatore viene interpretato, ma poi non viene salvato in alcuna variabile.

Successivamente può apparire un numero che rappresenta l'ampiezza del dato da interpretare, in byte, il cui scopo è quello di limitare la lettura fino a un certo carattere (inteso come '**char**', pertanto le sequenze multibyte contano per più di una unità singola). In questo caso non esiste la possibilità di indicare una precisione.

Dopo può apparire una sigla, composta da una o più lettere, il cui scopo è quello di modificare la dimensione predefinita della variabile di destinazione. In altri termini, senza questo modificatore si intende che la variabile ricevente debba essere di una certa grandezza, ma con l'aggiunta del «modificatore di lunghezza» si precisa invece qualcosa di diverso. In pratica, il modificatore di lunghezza usato da queste funzioni è equivalente a quello delle funzioni di composizione dell'output.

Al termine dello specificatore di conversione appare una lettera che

dichiara come deve essere interpretato il dato in ingresso e, in mancanza del modificatore di lunghezza, indica anche la dimensione predefinita della variabile ricevente.

Secondo la documentazione standard, il contenuto delle stringhe di conversione si suddivide in «direttive» che, in linea di massima, dovrebbero comporsi secondo il modello seguente:

[*spazi*] *carattere_multibyte* | %...

Pertanto, una direttiva può contenere degli spazi, un carattere (inteso in senso tipografico e quindi può occupare più di un byte) oppure uno specificatore di conversione. Visto da un altro punto di vista, la stringa di conversione è composta principalmente da specificatori di conversione che però possono essere alternati da spazi o altri caratteri: gli spazi indicano che in quella posizione possono esserci spazi che vengono ignorati; altri caratteri devono invece corrispondere esattamente nell'input e vengono poi ignorati. Tuttavia ci sono altre situazioni in cui gli spazi sono ugualmente esclusi in modo predefinito, come nell'esempio già visto, dove la stringa di conversione è composta solo da specificatori di conversione. Nell'esempio seguente, invece, si dimostra l'uso di caratteri estranei agli specificatori di conversione:

```
…
printf ("Inserisci la data: ");
scanf ("%i/%i/%i", &giorno, &mese, &anno);
…
```

In questo caso la digitazione della data richiede anche l'inserzione delle barre oblique, senza le quali il riconoscimento fallisce.

Purtroppo, la sintassi per la scrittura delle stringhe di conversione non è molto soddisfacente ed è difficile avere un'idea chiara del loro utilizzo. Pertanto, è consigliabile di utilizzare sempre solo modelli molto semplici.

67.3.6 Rappresentazione degli specificatori di conversione

«

Di norma, la scelta dello specificatore di conversione determina il tipo di dati dell'argomento (ovvero il tipo di variabile a cui l'argomento punta) e il modo in cui deve essere interpretato. La tabella successiva elenca alcuni degli specificatori di conversione utilizzabili, nella loro forma più semplice. È bene ricordare che anche in questo caso si può usare uno specificatore costituito dall'unione di due caratteri percentuali ('`%%`'), il quale identifica semplicemente un carattere di percentuale singolo proveniente dai dati in ingresso, ma da ignorare.

Tabella 67.32. Tipi di conversione principali.

Simbolo	Corrispondenza
%...c	Corrisponde a un carattere, oppure, se viene specificata una lunghezza, a una sequenza di caratteri; pertanto, in condizioni normali il puntatore deve essere di tipo '**`char *`**' e, a partire dalla posizione indicata dallo stesso, ci deve essere abbastanza spazio per contenere tutti i caratteri previsti per l'immissione.

Simbolo	Corrispondenza
`%…s`	Corrisponde a una sequenza di caratteri diversi da quelli che producono uno spazio e a questa sequenza viene aggiunto automaticamente il carattere *<NUL>*, ovvero '**`\0`**'; pertanto, in condizioni normali il puntatore deve essere di tipo '**`char *`**' e, a partire dalla posizione indicata dallo stesso, ci deve essere abbastanza spazio per contenere la stringa da immettere, incluso il carattere nullo conclusivo.
`%…d`	Corrisponde a un numero intero (con o senza segno), espresso in base dieci; pertanto, in condizioni normali il puntatore deve essere di tipo '**`int *`**', con o senza segno.
`%…i`	Corrisponde a un numero intero (con o senza segno), espresso in base otto, in base dieci o in base sedici (purché la base di numerazione sia riconoscibile dal prefisso usato); pertanto, in condizioni normali il puntatore deve essere di tipo '**`int *`**', con o senza segno.
`%…u`	Corrisponde a un numero intero senza segno, espresso in base dieci; pertanto, in condizioni normali il puntatore deve essere di tipo '**`unsigned int *`**'.
`%…x`	Corrisponde a un numero intero senza segno, espresso in base sedici; pertanto, in condizioni normali il puntatore deve essere di tipo '**`unsigned int *`**'.
`%…o`	Corrisponde a un numero intero senza segno, espresso in base otto; pertanto, in condizioni normali il puntatore deve essere di tipo '**`unsigned int *`**'.

Simbolo	Corrispondenza
`%…e` `%…f` `%…g`	Corrisponde a un numero a virgola mobile rappresentato in notazione decimale fissa o in notazione esponenziale; pertanto, in condizioni normali il puntatore deve essere di tipo '**`double *`**'.

Tabella 67.33. Alcuni modificatori dell'ampiezza del valore da immettere.

Simbolo	Corrispondenza
`%…hd` `%…hi` `%…hu` `%…ho` `%…hx`	La lettera '**`h`**', prima di quella che specifica il tipo di conversione, quando si tratta dell'inserimento di un numero intero, specifica che il dato va trattato in qualità di '**`short int *`**', con o senza segno, in base al contesto.
`%…ld` `%…lu` `%…lo` `%…lx`	La lettera '**`l`**', prima di quella che specifica il tipo di conversione, quando si tratta dell'inserimento di un numero intero, specifica che il dato va trattato in qualità di '**`long int *`**', con o senza segno, in base al contesto.

Simbolo	Corrispondenza
`%…lc` `%…ls`	La lettera '**`l`**', prima di quella che specifica il tipo di conversione, quando si tratta dell'inserimento di un carattere o di una stringa, specifica che il dato va trattato in qualità di carattere esteso (*wide char*) o di stringa estesa (*wide string*). Di conseguenza, si intende che l'argomento sia di tipo '**`wchar_t *`**'.
`%…Le` `%…Lf` `%…Lg`	La lettera '**`L`**', prima di quella che specifica il tipo di conversione, quando si tratta dell'inserimento di un valore in virgola mobile, specifica che il dato va trattato in qualità di '**`long double *`**'.

67.3.7 Funzioni per l'interpretazione dell'input

Tutte le funzioni standard il cui nome finisce per '**`scanf`**' interpretano dei dati in ingresso attraverso una stringa di conversione, secondo le modalità descritte nel capitolo, ovvero in modo analogo a ***scanf()*** che, in particolare, legge i dati da interpretare dallo standard input. In particolare, la funzione ***fscanf()*** legge l'input attraverso il flusso di file che costituisce il parametro ***stream*** (il primo argomento) e la funzione ***sscanf()*** legge l'input da una stringa (che costituisce sempre il primo argomento).

```
int fscanf (FILE *restrict stream,
            const char *restrict conversione, ...);
```

```
int sscanf (const char *restrict s,
            const char *restrict conversione, ...);
```

```
int scanf  (const char *restrict conversione, ...);
```

Le funzioni di cui è appena stato mostrato il modello sintattico, utilizzano gli argomenti successivi alla stringa di conversione in base a quanto indicato con gli specificatori di conversione. Altre funzioni equivalenti, con il nome che inizia con la lettera «v», hanno bisogno di un puntatore di tipo '**va_list**':

```
int vfscanf (FILE *restrict stream,
             const char *restrict conversione,
             va_list arg);
```

```
int vsscanf (const char *restrict s,
             const char *restrict conversione,
             va_list arg);
```

```
int vscanf  (const char *restrict conversione,
             va_list arg);
```

67.4 Riferimenti

«

- *Rationale for American National Standard for Information Systems - Programming Language - C: Input/Output*, http://www.lysator.liu.se/c/rat/d9.html
- *ISO/IEC 9899:TC2*, http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf

[1] Nella tabella, in questa fase, non si distingue ancora tra accessi a file di testo rispetto a quelli relativi a file binari, pertanto non appare mai la sigla '**b**'.

[2] Può trattarsi anche di sequenze multibyte, ovvero di rappresentazioni dei caratteri che usano più byte per carattere.

[3] Questa è una semplificazione, perché ci sono altre funzioni dello stesso gruppo, che iniziano con la lettera '**v**', le quali alla fine hanno un puntatore di tipo '**va_list**'.

[4] Questa è una semplificazione, perché ci sono altre funzioni dello stesso gruppo, che iniziano con la lettera '**v**', le quali alla fine hanno un puntatore di tipo '**va_list**'.

Esempi di programmazione in C

Problemi elementari 873

Somma tra due numeri positivi 874
Moltiplicazione di due numeri positivi attraverso la somma 876
Divisione intera tra due numeri positivi 877
Elevamento a potenza 879
Radice quadrata 881
Fattoriale 883
Massimo comune divisore 884
Numero primo 886
Successione di Fibonacci 887

Scansione di array 890

Ricerca sequenziale 890
Ricerca binaria 893

Algoritmi tradizionali 896

Bubblesort 896
Torre di Hanoi 899
Quicksort 900
Permutazioni 904

In questo capitolo vengono mostrati alcuni algoritmi elementari o comuni portati in C. Per la spiegazione di questi, se non sono già conosciuti, conviene leggere quanto riportato a partire dalla sezione 62.2.

«

Somma tra due numeri positivi

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/somma.c* .

Listato u3.1. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/0fuGkfkz* , *http://ideone.com/SQBOl* .

```
#include <stdio.h>

int
somma (int x, int y)
{
    int z = x;
    int i;

    for (i = 1; i <= y; i++)
      {
        z++;
      };

    return z;
}

int
main (int argc, char *argv[])
{
    int x;
    int y;
    int z;

    // Converte le stringhe ottenute dalla riga di comando in
    // numeri interi e li assegna alle variabili x e y.
```

```
    sscanf (argv[1], "%i", &x);
    sscanf (argv[2], "%i", &y);

    z = somma (x, y);

    printf ("%i + %i = %i\n", x, y, z);

    return 0;
}
```

In alternativa si può tradurre il ciclo '**`for`**' in un ciclo '**`while`**'.

Listato u3.2. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/TKwppy25* , *http://ideone.com/K33pD33* .

```
int
somma (int x, int y)
{
    int z = x;
    int i = 1;

    while (i <= y)
      {
        z++;
        i++;
      };

    return z;
}
```

Moltiplicazione di due numeri positivi attraverso la somma

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/moltiplica.c* .

Listato u3.3. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/xjV7MZte* , *http://ideone.com/b5Wxx* .

```
#include <stdio.h>

int
moltiplica (int x, int y)
{
    int z = 0;
    int i;

    for (i = 1; i <= y; i++)
      {
        z = z + x;
      }

    return z;
}

int
main (int argc, char *argv[])
{
    int x;
    int y;
    int z;

    // Converte le stringhe ottenute dalla riga di comando
    // in numeri interi e li assegna alle variabili x e y.

    sscanf (argv[1], "%i", &x);
    sscanf (argv[2], "%i", &y);
```

```
    z = moltiplica (x, y);

    printf ("%i * %i = %i\n", x, y, z);

    return 0;
}
```

In alternativa si può tradurre il ciclo '**for**' in un ciclo '**while**'.

Listato u3.4. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/9GQLZmXk* , *http://ideone.com/CoQf0* .

```
int moltiplica (int x, int y)
{
    int z = 0;
    int i = 1;

    while (i <= y)
      {
        z = z + x;
        i++;
      }

    return z;
}
```

Divisione intera tra due numeri positivi

Una copia di questo file dovrebbe essere disponibile presso *allegati/dividi.c* .

Listato u3.5. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/h15j1MVo* , *http://ideone.com/EmJ3X* .

```
#include <stdio.h>

int
dividi (int x, int y)
{
    int z = 0;
    int i = x;

    while (i >= y)
      {
        i = i - y;
        z++;
      }

    return z;
}

int
main (int argc, char *argv[])
{
    int x;
    int y;
    int z;

    // Converte le stringhe ottenute dalla riga di comando
    // in numeri interi e li assegna alle variabili x e y.

    sscanf (argv[1], "%i", &x);
    sscanf (argv[2], "%i", &y);

    z = dividi (x, y);
```

```
    printf ("Divisione intera - %i:%i = %i\n", x, y, z);

    return 0;
}
```

Elevamento a potenza

Una copia di questo file dovrebbe essere disponibile presso *allegati/exp.c* .

Listato u3.6. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/mxkVE5yi* , *http://ideone.com/guhJ6* .

```
#include <stdio.h>

int
exp (int x, int y)
{
    int z = 1;
    int i;

    for (i = 1; i <= y; i++)
      {
        z = z * x;
      }

    return z;
}

int
main (int argc, char *argv[])
{
    int x;
    int y;
    int z;
```

```
    // Converte le stringhe ottenute dalla riga di comando
    // in numeri interi e li assegna alle variabili x e y.

    sscanf (argv[1], "%i", &x);
    sscanf (argv[2], "%i", &y);

    z = exp (x, y);

    printf ("%i ** %i = %i\n", x, y, z);

    return 0;
}
```

In alternativa si può tradurre il ciclo '**for**' in un ciclo '**while**'.

Listato u3.7. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/7uP7uRGg* , *http://ideone.com/4X81h* .

```
int
exp (int x, int y)
{
    int z = 1;
    int i = 1;

    while (i <= y)
      {
        z = z * x;
        i++;
      };

    return z;
}
```

È possibile usare anche un algoritmo ricorsivo.

Listato u3.8. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/mwiZu0T7* , *http://ideone.com/x0ei9* .

```
int exp (int x, int y)
{
    if (x == 0)
      {
        return 0;
      }
    else if (y == 0)
      {
        return 1;
      }
    else
      {
        return (x * exp (x, y-1));
      }
}
```

Radice quadrata

Una copia di questo file dovrebbe essere disponibile presso *allegati/radice.c* .

Listato u3.9. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/8FgV7l1v* , *http://ideone.com/rO8KS* .

```
#include <stdio.h>

int
radice (int x)
{
    int z = 0;
    int t = 0;

    while (1)
```

```
      {
        t = z * z;

        if (t > x)
          {
            //  È stato superato il valore massimo.
            z--;
            return z;
          }

        z++;
      }

    // Teoricamente, non dovrebbe mai arrivare qui.
}

int
main (int argc, char *argv[])
{
    int x;
    int z;

    sscanf (argv[1], "%i", &x);

    z = radice (x);

    printf ("radq(%i) = %i\n", x, z);

    return 0;
}
```

Una copia di questo file dovrebbe essere disponibile presso *allegati/fatt.c* .

Listato u3.10. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/vBGnZfEf* , *http://ideone.com/jYVgR* .

```
#include <stdio.h>

int
fatt (int x)
{
    int i = (x - 1);

    while (i > 0)
      {
        x = x * i;
        i--;
      }

    return x;
}

int
main (int argc, char *argv[])
{
    int x;
    int z;

    sscanf (argv[1], "%i", &x);

    z = fatt (x);

    printf ("%i! = %i\n", x, z);
```

```
    return 0;
}
```

In alternativa, l'algoritmo si può tradurre in modo ricorsivo.

Listato u3.11. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/LIC6Vyxp* , *http://ideone.com/6SfUj* .

```
int
fatt (int x)
{
    if (x > 1)
      {
        return (x * fatt (x - 1));
      }
    else
      {
        return 1;
      }
}
```

Massimo comune divisore

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/mcd.c* .

Listato u3.12. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/ETf2XcdR* , *http://ideone.com/8H1og* .

```
#include <stdio.h>

int
mcd (int x, int y)
```

```
{
    while (x != y)
      {
        if (x > y)
          {
            x = x - y;
          }
        else
          {
            y = y - x;
          }
      }

    return x;
}

int
main (int argc, char *argv[])
{
    int x;
    int y;
    int z;

    sscanf (argv[1], "%i", &x);
    sscanf (argv[2], "%i", &y);

    z = mcd (x, y);

    printf ("Il massimo comune divisore di %i e %i è %i\n",
            x, y, z);

    return 0;
}
```

Numero primo

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/primo.c* .

Listato u3.13. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/T6vjaM2y* , *http://ideone.com/X1oos* .

```
#include <stdio.h>

unsigned int
primo (int x)
{
    unsigned int primo = 1;
    int i = 2;
    int j;

    while ((i < x) && primo)
      {
        j = x / i;
        j = x - (j * i);

        if (j == 0)
          {
            primo = 0;
          }
        else
          {
            i++;
          }
      }

    return primo;
}
```

```
int
main (int argc, char *argv[])
{
    int x;

    sscanf (argv[1], "%i", &x);

    if (primo (x))
      {
        printf ("%i è un numero primo\n", x);
      }
    else
      {
        printf ("%i non è un numero primo\n", x);
      }

    return 0;
}
```

Successione di Fibonacci

«

Gli esempi mostrano una funzione che restituisce l'elemento ***n***-esimo nella sequenza di Fibonacci, mentre la chiamata di questa funzione viene fatta in modo da ottenere l'elenco dei primi ***n*** numeri di Fibonacci. La prima soluzione mostra una funzione ricorsiva. Una copia di questo file dovrebbe essere disponibile presso *allegati/fibonacci.c* .

Listato u3.14. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/6CHi9taB* , *http://ideone.com/0Z1dz* .

```
#include <stdio.h>

unsigned int
fibonacci (unsigned int n)
{
    if (n == 0)
      {
        return 0;
      }
    else if (n == 1)
      {
        return 1;
      }
    else
      {
        return (fibonacci (n-1) + fibonacci (n-2));
      }
}

int
main (int argc, char *argv[])
{
    unsigned int n;
    unsigned int i;

    sscanf (argv[1], "%u", &n);

    for (i = 0; i <= n; i++)
      {
        printf ("%u ", fibonacci (i));
      }
```

```
    printf ("\n");

    return 0;
}
```

L'esempio seguente mostra solo la funzione, in forma iterativa:

Listato u3.15. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/1sH2hhIf* , *http://ideone.com/8z9J8* .

```
unsigned int
fibonacci (unsigned int n)
{
    unsigned int f1 = 1;
    unsigned int f0 = 0;
    unsigned int fn = n;
    unsigned int i;

    for (i = 2; i <= n; i++)
      {
        fn = f1 + f0;
        f0 = f1;
        f1 = fn;
      }

    return fn;
}
```

«

Ricerca sequenziale

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/ricercaseq.c* .

Listato u3.16. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/9p0P3GU9* , *http://ideone.com/J8hQb* .

```
#include <stdio.h>

int
ricercaseq (int lista[], int x, int a, int z)
{
    int i;

    // Scandisce l'array alla ricerca dell'elemento.

    for (i = a; i <= z; i++)
      {
        if (x == lista[i])
          {
            return i;
          }
      }

    // La corrispondenza non è stata trovata.

    return -1;
}

int
main (int argc, char *argv[])
```

```
{
    int lista[argc - 2];
    int x;
    int i;

    // Acquisisce il primo argomento come valore da cercare.

    sscanf (argv[1], "%i", &x);

    // Considera gli argomenti successivi come gli elementi
    // dell'array da scandire.

    for (i = 2; i < argc; i++)
      {
        sscanf (argv[i], "%i", &lista[i-2]);
      }

    // Esegue la ricerca.

    i = ricercaseq (lista, x, 0, argc - 2);

    // Emette il risultato.

    printf ("%i si trova nella posizione %i\n", x, i);

    return 0;
}
```

Al posto di dichiarare l'array ***lista*** con una quantità di elementi definita in fase di funzionamento, si può usare la funzione ***malloc()***, avendo cura di incorporare il file '`stdlib.h`':

Listato u3.17. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/5nP79Nf798* , *http://ideone.com/YdMdC* .

```
#include <stdio.h>
#include <stdlib.h>
…
int
main (int argc, char *argv[])
{
    int *lista = (int *) malloc ((argc - 2) * sizeof (int));
…
```

Esiste anche una soluzione ricorsiva che viene mostrata nella subroutine seguente:

Listato u3.18. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/SC2AheV2* , *http://ideone.com/sk64m* .

```
int
ricercaseq (int lista[], int x, int a, int z)
{
    if (a > z)
      {
        // La corrispondenza non è stata trovata.

        return -1;
      }
    else if (x == lista[a])
      {
        return a;
      }
    else
      {
        return ricercaseq (lista, x, a+1, z);
      }
}
```

Ricerca binaria

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/ricercabin.c* .

Listato u3.19. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/vU9RwE9m* , *http://ideone.com/jEyYk* .

```
#include <stdio.h>

int
ricercabin (int lista[], int x, int a, int z)
```

```
{
    int m;

    // Determina l'elemento centrale.

    m = (a + z) / 2;

    if (m < a)
      {
        // Non restano elementi da controllare: l'elemento
        // cercato non c'è.

        return -1;
      }
    else if (x < lista[m])
      {
        // Si ripete la ricerca nella parte inferiore.

        return ricercabin (lista, x, a, m-1);
      }
    else if (x > lista[m])
      {
        // Si ripete la ricerca nella parte superiore.

        return ricercabin (lista, x, m+1, z);
      }
    else
      {
        // La variabile m rappresenta l'indice dell'elemento
        // cercato.

        return m;
      }
}
```

```
int main (int argc, char *argv[])
{
    int lista[argc - 2];
    int x;
    int i;

    // Acquisisce il primo argomento come valore da cercare.

    sscanf (argv[1], "%i", &x);

    // Considera gli argomenti successivi come gli elementi
    // dell'array da scandire.

    for (i = 2; i < argc; i++)
      {
        sscanf (argv[i], "%i", &lista[i-2]);
      }

    // Esegue la ricerca.

    i = ricercabin (lista, x, 0, argc-2);

    // Emette il risultato.

    printf ("%i si trova nella posizione %i\n", x, i);

    return 0;
}
```

Per questo esempio vale la stessa considerazione fatta nella sezione precedente, a proposito dell'uso di ***malloc()*** al posto di un array con una quantità di elementi definita dinamicamente durante il

funzionamento del programma.

Algoritmi tradizionali

«

Bubblesort

«

Viene mostrata prima una soluzione iterativa e successivamente la funzione '**bsort**' in versione ricorsiva.

Una copia di questo file dovrebbe essere disponibile presso *allegati/bsort.c* .

Listato u3.20. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/10ot10rUOE* , *http://ideone.com/hJIie* .

```
#include <stdio.h>

void
bsort (int lista[], int a, int z)
{
    int scambio;
    int j;
    int k;

    // Inizia il ciclo di scansione dell'array.

    for (j = a; j < z; j++)
      {
        // Scansione interna dell'array per collocare nella
        // posizione j l'elemento giusto.

        for (k = j+1; k <= z; k++)
          {
            if (lista[k] < lista[j])
              {
```

```
                    // Scambia i valori.

                    scambio = lista[k];
                    lista[k] = lista[j];
                    lista[j] = scambio;
                  }
              }
        }
}

int
main (int argc, char *argv[])
{
    int lista[argc-1];
    int i;

    // Considera gli argomenti come gli elementi
    // dell'array da ordinare.

    for (i = 1; i < argc; i++)
      {
        sscanf (argv[i], "%i", &lista[i-1]);
      }

    // Esegue il riordino.

    bsort (lista, 0, argc-2);

    // Emette il risultato.

    for (i = 0; i < (argc-1); i++)
      {
        printf ("%i ", lista[i]);
      }
```

```
    printf ("\n");

    return 0;
}
```

Segue la funzione '**bsort**' in versione ricorsiva.

Listato u3.21. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/izHITJ7y* , *http://ideone.com/5mSDT* .

```
void
bsort (int lista[], int a, int z)
{
    int scambio;
    int k;

    if (a < z)
      {
        // Scansione interna dell'array per collocare nella
        // posizione a l'elemento giusto.

        for (k = a+1; k <= z; k++)
          {
            if (lista[k] < lista[a])
              {
                // Scambia i valori.

                scambio = lista[k];
                lista[k] = lista[a];
                lista[a] = scambio;
              }
          }

        bsort (lista, a+1, z);
```

```
        }
}
```

Al posto di dichiarare l'array ***lista*** con una quantità di elementi definita in fase di funzionamento, si può usare la funzione ***malloc()***, avendo cura di incorporare il file '`stdlib.h`':

Listato u3.22. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/M76CQ76gHg* , *http://ideone.com/j7hB5* .

```
#include <stdio.h>
#include <stdlib.h>
…
int
main (int argc, char *argv[])
{
    int *lista = (int *) malloc ((argc - 1) * sizeof (int));
…
```

Torre di Hanoi

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/hanoi.c* .

Listato u3.23. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/FvvVQmru* , *http://ideone.com/kr6DC* .

```
#include <stdio.h>

void hanoi (int n, int p1, int p2)
{
    if (n > 0)
```

```
        {
          hanoi (n-1, p1, 6-p1-p2);
          printf ("Muovi l'anello %i dal piolo %i "
                  "al piolo %i\n",
                  n, p1, p2);
          hanoi (n-1, 6-p1-p2, p2);
        }
}

int main (int argc, char *argv[])
{
    int n;
    int p1;
    int p2;

    sscanf (argv[1], "%i", &n);
    sscanf (argv[2], "%i", &p1);
    sscanf (argv[3], "%i", &p2);

    hanoi (n, p1, p2);

    return 0;
}
```

Quicksort

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/qsort.c* .

Listato u3.24. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/QbDq6aMz* , *http://ideone.com/NJwFO* .

```
#include <stdio.h>

int
part (int lista[], int a, int z)
{
    // Viene preparata una variabile per lo scambio di
    // valori.

    int scambio = 0;

    // Si assume che «a» sia inferiore a «z».

    int i = a + 1;
    int cf = z;

    // Inizia il ciclo di scansione dell'array.

    while (1)
      {
        while (1)
          {
            // Sposta «i» a destra.

            if ((lista[i] > lista[a]) || (i >= cf))
              {
                break;
              }
            else
              {
                i += 1;
              }
```

```
        }
      while (1)
        {
          // Sposta «cf» a sinistra.

          if (lista[cf] <= lista[a])
            {
              break;
            }
          else
           {
              cf -= 1;
           }
        }
      if (cf <= i)
        {
          // È avvenuto l'incontro tra «i» e «cf».

          break;
        }
      else
        {
          // Vengono scambiati i valori.

          scambio = lista[cf];
          lista[cf] = lista[i];
          lista[i] = scambio;

          i += 1;
          cf -= 1;
        }
    }

  // A questo punto lista[a..z] è stata ripartita e «cf» è
```

```
    // la collocazione di «lista[a]».

    scambio = lista[cf];
    lista[cf] = lista[a];
    lista[a] = scambio;

    // A questo punto, lista[cf] è un elemento (un valore)
    // nella giusta posizione.

    return cf;
}

void
quicksort (int lista[], int a, int z)
{
    // Viene preparata la variabile «cf».

    int (cf) = 0;

    if (z > a)
      {
        cf = part (lista, a, z);
        quicksort (lista, a, cf-1);
        quicksort (lista, cf+1, z);
      }
}

int
main (int argc, char *argv[])
{
    int lista[argc - 1];
    int i;

    // Considera gli argomenti come gli elementi
```

```
    // dell'array da ordinare.

    for (i = 1; i < argc; i++)
      {
        sscanf (argv[i], "%i", &lista[i-1]);
      }

    // Esegue il riordino.

    quicksort (lista, 0, argc-2);

    // Emette il risultato.

    for (i = 0; i < (argc-1); i++)
      {
        printf ("%i ", lista[i]);
      }
    printf ("\n");

    return 0;
}
```

Per questo esempio vale la stessa considerazione già fatta a proposito dell'uso di ***malloc()*** al posto di un array con una quantità di elementi definita dinamicamente durante il funzionamento del programma.

Permutazioni

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/permuta.c* .

Listato u3.25. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/ca66C9da* , *http://ideone.com/I5bJV* .

```
#include <stdio.h>

void visualizza (int lista[], int dimensione)
{
    int i;

    for (i = 0; i < dimensione; i++)
      {
        printf ("%i ", lista[i]);
      }
    printf ("\n");
}

void permuta (int lista[], int a, int z, int dimensione)
{
    int scambio;
    int k;

    // Se il segmento di array contiene almeno due elementi,
    // si procede.

    if ((z - a) >= 1)
      {
        // Inizia un ciclo di scambi tra l'ultimo elemento e
        // uno degli altri contenuti nel segmento di array.

        for (k = z; k >= a; k--)
          {
            // Scambia i valori.

            scambio = lista[k];
```

```
            lista[k] = lista[z];
            lista[z] = scambio;

            // Esegue una chiamata ricorsiva per permutare
            // un segmento più piccolo dell'array.

            permuta (lista, a, z - 1, dimensione);

            // Scambia i valori.

            scambio = lista[k];
            lista[k] = lista[z];
            lista[z] = scambio;
          }
      }
    else
      {
        // Visualizza l'array.

        visualizza (lista, dimensione);
      }
}

int
main (int argc, char *argv[])
{
    int lista[argc - 1];
    int i;

    // Considera gli argomenti come gli elementi
    // dell'array da permutare.

    for (i = 1; i < argc; i++)
      {
```

```
        sscanf (argv[i], "%i", &lista[i-1]);
      }

    // Esegue le permutazioni.

    permuta (lista, 0, argc - 2, argc - 1);

    return 0;
}
```

Per questo esempio vale la stessa considerazione già fatta a proposito dell'uso di ***malloc()*** al posto di un array con una quantità di elementi definita dinamicamente durante il funzionamento del programma.

Capitolo 68

Introduzione alle estensioni POSIX

68.1 Dal C a POSIX 912
68.1.1 Byte di otto bit 913
68.1.2 Errori 914
68.1.3 Gestione dei file 915
68.1.4 Altre funzionalità particolari 915
68.2 Espressioni regolari POSIX 916
68.2.1 Compilazione, confronto e rilascio della memoria 918
68.2.2 Compilazione dell'espressione regolare 919
68.2.3 Liberazione della memoria 923
68.2.4 Confronto 923
68.2.5 Estrapolazione di sottostringhe 927
68.2.6 Informazioni diagnostiche 932
68.2.7 Esempio completo 936
68.3 Avvio e conclusione dei processi 940
68.3.1 Biforcazione: «fork» 940
68.3.2 Sostituzione: «exec» 943
68.3.3 Attesa della conclusione di un processo figlio 945
68.3.4 Adozione dei processi 946
68.3.5 Gli zombie 948
68.3.6 Variabili di ambiente 950
68.3.7 Variabili di ambiente e avvio di un programma . . 951

68.4 Nozioni sui thread POSIX 953
68.4.1 Numero identificativo del thread 954
68.4.2 Creazione e conclusione di un thread aggiuntivo . . 955
68.4.3 Caratteristiche della funzione che costituisce un thread aggiuntivo 956
68.4.4 Avvio di thread separati e fusione successiva 957
68.4.5 Conflitto nell'accesso ai dati 962
68.4.6 Accesso alle risorse in modo mutualmente esclusivo 966
68.4.7 Accesso esclusivo, ma condizionato 972
68.4.8 Osservazioni finali 978
68.5 I file secondo i sistemi POSIX 979
68.5.1 Apertura e chiusura di un file 980
68.5.2 Lettura e scrittura 987
68.5.3 Spostamento dell'indicatore interno al file 992
68.5.4 Controllo degli errori 995
68.6 Il file system Unix e la sua gestione tipica 996
68.6.1 Il blocco 996
68.6.2 Il super blocco 997
68.6.3 Gli inode 998
68.6.4 La directory 1001
68.6.5 Tabelle del sistema operativo 1002
68.7 Il file system Minix 1 1008

68.7.1 Blocchi e zone 1009
68.7.2 Struttura generale 1010
68.7.3 Super blocco 1011
68.7.4 Mappa di inode 1013
68.7.5 Mappa di zone 1015
68.7.6 Inode 1016
68.7.7 Directory 1020
68.7.8 Il problema dell'inversione dei byte 1022
68.8 Creazione ed eliminazione di file di qualunque tipo . . 1022
68.8.1 La funzione «mknod()» 1023
68.8.2 La funzione «mkdir()» 1031
68.8.3 La funzione «mkfifo()» 1032
68.8.4 La funzione «unlink()» 1032
68.8.5 La funzione «rmdir()» 1033
68.8.6 La funzione «remove()» 1034
68.9 Condotti 1034
68.9.1 Realizzazione materiale del condotto 1035
68.9.2 Condotti «senza nome» e condotti «con nome» . . 1037
68.9.3 Protocollo di accesso ai condotti 1037
68.9.4 Funzione «pipe()» 1038
68.9.5 Esempio di condotto attraverso un file FIFO 1043
68.10 Lettura delle directory 1047
68.10.1 Tipi derivati 1048
68.10.2 Procedura per accedere a una directory 1048

68.10.3 Attributo «FD_CLOEXEC» 1049
68.10.4 Esempio di utilizzo delle funzioni di accesso alle directory 1050
68.11 Riferimenti 1051

close() 980 closedir() 1048 creat() 980 DIR 1048 dirent.h 1047 environ 950 errno 995 execl() 943 execve() 951 fcntl.h 979 FD_CLOEXEC 1049 fork() 940 init 946 lseek() 992 major() 1028 makedev() 1028 minor() 1028 mkdir() 1031 mkfifo() 1032 1043 mknod() 1023 open() 980 opendir() 1048 O_APPEND 980 O_CREAT 980 O_EXCL 980 O_NOCTTY 980 O_NONBLOCK 980 O_RDONLY 980 O_RDWR 980 O_SYNC 980 O_TRUNC 980 O_WRONLY 980 pipe() 1038 pthread_t 954 read() 987 readdir() 1048 regcomp() 918 919 regerror() 918 932 regex.h 918 regexec() 918 923 927 regex_t 918 919 regfree() 918 923 regmatch_t 918 923 927 remove() 1034 rewinddir() 1048 rmdir() 1033 stat.h 979 struct dirent 1048 S_IRGRP 980 S_IROTH 980 S_IRUSR 980 S_IRWXG 980 S_IRWXO 980 S_IRWXU 980 S_ISGID 980 S_ISUID 980 S_ISVTX 980 S_IWGRP 980 S_IWOTH 980 S_IWUSR 980 S_IXGRP 980 S_IXOTH 980 S_IXUSR 980 unistd.h 979 unlink() 1032 wait() 945 write() 987

68.1 Dal C a POSIX

«

Lo standard del linguaggio C è definito in modo tale da consentirne l'uso in contesti architetturali molto diversi tra loro. Tuttavia, il linguaggio C è nato per i sistemi Unix e quando si vuole considera-

re una libreria di funzioni più ampia, rispetto allo standard del linguaggio, ci si riferisce normalmente allo standard POSIX. Pertanto, lo standard POSIX estende la libreria del linguaggio C, con funzionalità che dipendono da un'organizzazione del sistema operativo conforme, almeno fondamentalmente, a quella di Unix.

Logicamente, i sistemi operativi possono essere più o meno conformi con lo standard POSIX e l'utilizzo delle estensioni introdotte da questo standard va preceduto da una ricerca sulla loro compatibilità.

In generale va osservato che dal C puro a POSIX, la filosofia di programmazione cambia leggermente e occorre considerare alcuni particolari.

68.1.1 Byte di otto bit

Lo standard C prescrive che un byte sia di almeno 8 bit, mentre secondo POSIX, il byte diventa necessariamente di 8 bit. Infatti, nel file di intestazione '`stdint.h`', secondo lo standard C la definizione dei tipi derivati '**[u]int*n*_t**' è facoltativa, ma nello standard POSIX diviene obbligatoria. Così facendo, dato che diviene obbligatorio disporre dei tipi derivati '**int8_t**' e '**uint8_t**', non avendo un altro modo per definirli, è necessario che il tipo '**char**' sia esattamente di 8 bit:

```
typedef signed char             int8_t;
…
typedef unsigned char           uint8_t;
```

Per lo stesso motivo, tutti gli altri tipi interi devono essere multipli (al quadrato), del byte, ma oltre a questo, quando la CPU fosse anche in grado di gestire facilmente interi più grandi di 32 bit, il tipo '**int**' potrebbe essere al massimo di soli 32 bit. Per esempio, dispo-

nendo di una CPU a 128 bit, si è praticamente costretti a dichiarare questi tipi derivati nel modo seguente, riservando, evidentemente, il tipo '**`long long int`**' per gli interi a 128 bit che per l'architettura sarebbero invece normali:

```
typedef signed char                 int8_t;
typedef short int                   int16_t;
typedef int                         int32_t;
typedef long int                    int64_t;
//typedef long long int             int128_t;   // non standard

typedef unsigned char               uint8_t;
typedef unsigned short int          uint16_t;
typedef unsigned int                uint32_t;
typedef unsigned long int           uint64_t;
//typedef unsigned long long int    uint128_t;  // non standard
```

68.1.2 Errori

Molte funzioni, aggiunte dallo standard POSIX, che restituiscono un valore intero, utilizzano il valore −1 per dichiarare la presenza di un errore che ha prodotto un esito non valido. In tal caso, si possono avere funzioni che restituiscono un valore, che, se maggiore o uguale a zero, è valido, mentre se è negativo non lo è, oppure funzioni che semplicemente restituiscono zero o −1.

Questo fatto si scontra con il principio per cui lo zero equivale a *Falso* e un valore diverso da zero equivale a *Vero*, perché in questo caso occorre verificare precisamente il valore, oppure occorre invertirlo logicamente perché abbia il significato atteso.

Questa inversione logica dei risultati si riscontra anche nei programmi di servizio di un sistema operativo POSIX, per cui il linguaggio

della shell POSIX considera lo zero come un valore pari a *Vero* e qualunque altro valore pari a *Falso*.

Oltre a questo fatto, i tipi di errore che si possono annotare nella variabile ***errno*** sono molto più numerosi rispetto a quanto previsto dallo standard C, pertanto aumentano le macro-variabili previste nel file ‘`errno.h`’.

68.1.3 Gestione dei file

La libreria standard del linguaggio C prevede una gestione dei file attraverso dei «flussi», mentre la libreria POSIX introduce il concetto di descrittore del file (*file descriptor*), ovvero un semplice numero intero.[1]

In uno stesso programma si possono usare entrambe le modalità di gestione dei file, ma è evidente che va evitato l’uso simultaneo sullo stesso file.

A parte la differenza nel modo di identificare un file aperto in un programma, la libreria POSIX mette in condizione di accedere alla gestione dei permessi e delle altre caratteristiche dei file, secondo le convenzioni di un sistema Unix; inoltre offre una gestione ordinata dei blocchi di accesso a porzioni degli stessi (*lock*).

68.1.4 Altre funzionalità particolari

La libreria POSIX offre anche altre funzionalità che possono essere importanti. Per esempio definisce delle funzioni per il trattamento delle espressioni regolari e per la gestione dei thread multipli.

68.2 Espressioni regolari POSIX

«

Un'espressione regolare è un modello che descrive la corrispondenza con una porzione di una stringa. Le espressioni regolari sono costruite, in maniera analoga alle espressioni matematiche, combinando espressioni più brevi. Lo standard POSIX distingue due tipi di espressioni regolari: quelle elementari, o BRE, e quelle estese, o ERE (si vedano in particolare le sezioni dedicate alle espressioni regolari in generale: 23.1 e 23.2). Lo standard POSIX prevede una libreria specifica per la gestione delle espressioni regolari, a cui si fa riferimento in questo capitolo.

Va osservato che nel caso del compilatore GCC, con le librerie GNU, per la compilazione sono sufficienti le librerie principali, le quali vengono incluse automaticamente.

Tabella 68.3. Schema sintetico delle espressioni regolari, secondo la definizione POSIX: operatori fondamentali.

Descrizione	BRE POSIX	ERE POSIX
Escape: protezione del carattere successivo o attribuzione a tale carattere di un significato speciale.	`\`	`\`
Ancora all'inizio della riga o della stringa.	`^`	`^`
Ancora alla fine della riga o della stringa.	`$`	`$`
Alternativa.		`|`
Raggruppamento.	`\(  \)`	`(  )`
Elenco (individua un solo carattere).	`[  ]`	`[  ]`

Descrizione	BRE POSIX	ERE POSIX
Riferimento al raggruppamento ***n*** esimo.	***n***	

Tabella 68.4. Schema sintetico delle espressioni regolari, secondo la definizione POSIX: operatori interni alle espressioni tra parentesi quadre.

Descrizione	BRE POSIX	ERE POSIX
Sequenze.	***xy***...	***xy***...
Intervalli.	***x***-***y***	***x***-***y***
Elementi di collazione.	[. .]	[. .]
Caratteri equivalenti.	[= =]	[= =]
Classi di caratteri.	[: :]	[: :]

Tabella 68.5. Schema sintetico delle espressioni regolari, secondo la definizione POSIX: operatori di ripetizione.

Descrizione	BRE POSIX	ERE POSIX
Zero o più ripetizioni del carattere ***x***.	***x****	***x****
Una o nessuna occorrenza del carattere ***x***.		***x***?
Una o più ripetizioni del carattere ***x***.		***x***+
Esattamente ***n*** volte il carattere ***x***.	***x***\{***n***\}	***x***{***n***}
Almeno ***n*** volte ***x***.	***x***\{***n***,\}	***x***{***n***,}

Descrizione	BRE POSIX	ERE POSIX
Da ***n*** a ***m*** volte ***x***.	***x*** \{ ***n*** , ***m*** \}	***x*** { ***n*** , ***m*** }

68.2.1 Compilazione, confronto e rilascio della memoria

«

Per eseguire un confronto con un'espressione regolare, attraverso le funzioni definite dallo standard POSIX, è necessario prima tradurre l'espressione regolare in una variabile strutturata di tipo '**`regex_t`**'. Tale operazione di analisi e traduzione dell'espressione regolare viene definita dallo standard come «compilazione». La compilazione dell'espressione regolare avviene con la funzione ***regcomp()***, con la quale, oltre che fornire la stringa contenente l'espressione regolare stessa, si devono specificare delle opzioni sul modo in cui interpretarla o gestirla. Per esempio, in questa fase va stabilito se l'espressione è di tipo BRE o ERE, se conta la differenza tra lettere maiuscole e minuscole, se si intendono estrapolare delle sottostringhe attraverso la comparazione, se il codice di interruzione di riga ha un qualche valore particolare o meno.

Quando si dispone di un'espressione regolare compilata, si può passare alla comparazione di questa con una stringa, attraverso la funzione ***regexec()***, con la quale si possono dare delle opzioni aggiuntive, nel modo finale di effettuare il confronto. Per la comparazione può essere necessaria la definizione di una variabile strutturata, di tipo '**`regmatch_t`**'.

Quando la variabile strutturata contenente l'espressione regolare non serve più, va rilasciata espressamente la memoria a cui gli elementi della stessa fanno riferimento. In altri termini, non basta liberare

la memoria della variabile che rappresenta la struttura, perché rimarrebbero allocati altri dati raggiunti attraverso dei puntatori. Per liberare un'espressione regolare compilata si utilizza la funzione ***regfree()***.

Le funzioni ***regcomp()*** e ***regexec()***, hanno la caratteristica di restituire un valore intero, pari a zero se l'operazione è stata conclusa con successo, oppure un valore diverso se si è verificato qualche tipo di problema. I valori restituiti, se diversi da zero, sono codificati ordinatamente da macro-variabili simboliche appropriate. Eventualmente, con la funzione ***regerror()***, è possibile ottenere la traduzione dell'errore, associato al riferimento all'espressione regolare compilata, in una stringa più esplicita.

Per utilizzare le espressioni regolari POSIX è necessario includere inizialmente il file di intestazione '`regex.h`'.

68.2.2 Compilazione dell'espressione regolare

Un'espressione regolare, in forma di stringa, viene compilata attraverso la funzione ***regcomp()***, inserendo i dati necessari in una struttura di tipo '**`regex_t`**'.

```
int regcomp (regex_t *restrict re, const char *restrict regex,
             int cflags);
```

Il prototipo della funzione mostra che il primo parametro, ***re***, deve essere un puntatore al tipo '**`regex_t`**': si tratta della struttura che viene modificata attraverso la compilazione. Il secondo parametro, ***regex***, è la stringa che descrive l'espressione regolare (la stringa deve essere terminata regolarmente con un carattere nullo, come di

consueto). L'ultimo parametro, '**cflags**', è un numero intero i cui bit descrivono le opzioni da considerare per l'interpretazione corretta della stringa dell'espressione regolare; tali bit vengono composti assieme attraverso l'uso di macro-variabili simboliche che fanno parte della stessa libreria della funzione ***regcomp()***.

Tabella 68.6. Il tipo '**regex_t**' definisce una variabile strutturata che contiene almeno il membro '**re_sub**'.

Tipo	Nome	Descrizione
`size_t`	`re_sub`	Quantità di sottoespressioni tra parentesi tonde.

Va osservato che l'espressione regolare viene fornita attraverso una stringa «normale», ovvero un array di '**char**'.

L'esempio seguente dovrebbe servire a comprendere l'uso della funzione ***regcomp()***:

```
#include <regex.h>
…
    regex_t re;
    regcomp (&re, "01[a-z]*", REG_EXTENDED|REG_NOSUB);
…
```

Come si può osservare, viene dichiarata una variabile di tipo '**regex_t**', della quale viene fornito il puntatore nella chiamata di ***regcomp()***; inoltre, l'ultimo argomento della funzione è composto utilizzando due macro-variabili simboliche, sommate assieme con l'operatore '`|`', ovvero un OR bit per bit.

Tabella 68.8. Macro-variabili simboliche da usare come opzioni per la compilazione di un'espressione regolare.

Macro-variabile	Significato
`REG_EXTENDED`	L'espressione regolare fornita è di tipo ERE (estesa). Se non si usa questa opzione, si intende che l'espressione sia di tipo BRE.
`REG_ICASE`	L'espressione regolare fornita va valutata senza distinguere tra lettere maiuscole o minuscole.
`REG_NOSUB`	Per la compilazione dell'espressione regolare non si intende tenere conto della corrispondenza eventuale di sottostringhe; in altri termini, non si vogliono considerare le parentesi '**`\(`**' e '**`\)`**', oppure '**`(`**' e '**`)`**' (a seconda che si tratti di ERE o BRE). In tal caso, l'espressione regolare serve per il confronto, ma non per estrapolare porzioni del risultato ottenuto.
`REG_NEWLINE`	In condizioni normali, il codice *new-line* contenuto nella stringa con cui l'espressione regolare deve essere confrontata, viene trattato come gli altri caratteri. Con l'opzione '**`REG_NEWLINE`**', invece, l'operatore '**`^`**' individua l'inizio di un testo che segue un codice *new-line*, mentre l'operatore '**`$`**' individua la fine di un testo che precede un codice *new-line*.

La funzione ***regcomp()*** restituisce zero se il procedimento di compilazione dell'espressione regolare termina regolarmente, senza problemi nell'interpretazione della stringa che la rappresenta; altrimenti

restituisce un valore diverso che rappresenta un errore. Per poter valutare l'errore, occorre fare un confronto con delle macro-variabili simboliche, come descritto nella tabella successiva.

Tabella 68.9. Macro-variabili simboliche che rappresentano il tipo di errore restituito dalla funzione ***regcomp()***.

Macro-variabile	Significato
`REG_BADBR`	Il contenuto di '**\{**…**\}**' (nel caso di BRE) o di '**{**…**}**' (nel caso di ERE), risulta non valido: potrebbe non trattarsi di un numero, oppure potrebbe esserci un numero troppo grande, oppure potrebbero esserci più di due numeri, oppure il primo potrebbe essere più grande del secondo. Infatti, il contenuto di tale raggruppamento deve essere un numero singolo, oppure due numeri separati da una virgola, dove il primo deve essere inferiore al secondo.
`REG_BADPAT`	Espressione regolare non valida (errore di sintassi).
`REG_BADRPT`	Un operatore di ripetizione, del tipo '**?**', '*****' o '**+**', non è preceduto a un'espressione regolare, ovvero si trova in una posizione sbagliata.
`REG_ECOLLATE`	Elemento di collazione (*collating element*) non valido, nell'ambito della configurazione locale attuale.
`REG_ECTYPE`	Riferimento a una classe di caratteri non valida.
`REG_EESCAPE`	L'espressione regolare termina con '****' e ciò non è ammissibile.
`REG_ESUBREG`	Una sequenza '***n***', dove ***n*** è un numero, è errata.

Macro-variabile	Significato
`REG_EBRACE`	Le parentesi graffe che descrivono la ripetizione di qualcosa non bilanciano. Può trattarsi di sequenze del tipo '`\{...\}`' per le espressioni BRE o del tipo '`{...}`' per le espressioni ERE.
`REG_EBRACK`	Parentesi quadre non bilanciate (parentesi aperta e non chiusa, o viceversa).
`REG_EPAREN`	Le parentesi tonde che descrivono delle sottoespressioni non bilanciano. Può trattarsi di sequenze del tipo '`\(...\)`' per le espressioni BRE o del tipo '`(...)`' per le espressioni ERE.
`REG_ERANGE`	Un'estremità di un intervallo di valori non è valido.
`REG_ESPACE`	Il procedimento di interpretazione dell'espressione regolare porta all'esaurimento della memoria disponibile.

68.2.3 Liberazione della memoria

Un'espressione regolare compilata occupa memoria, non solo nella variabile strutturata che la rappresenta, ma anche in altre aree a cui il contenuto di tale variabile può puntare. Quando l'espressione regolare non serve più, la memoria relativa va liberata esplicitamente, attraverso la funzione ***regfree()***, la quale non restituisce alcunché e richiede di indicare solo il puntatore alla variabile strutturata che rappresenta l'espressione regolare stessa.

```
void regfree (regex_t *re);
```

68.2.4 Confronto

«

La comparazione di un'espressione regolare compilata e di una stringa, si svolge con la funzione ***regexec()***. Questa funzione richiede diversi argomenti, perché ci deve essere la possibilità di estrapolare anche delle sottostringhe, corrispondenti a delle sottoespressioni racchiuse tra parentesi tonde.

```
int regexec (const regex_t *restrict re,
             const char *restrict stringa,
             size_t n_match, regmatch_t p_match[restrict],
             int eflags);
```

Il prototipo della funzione ***regexec()*** può apparire inizialmente complicato da interpretare. I primi due parametri sono sostanzialmente l'espressione regolare compilata e la stringa da confrontare. Il terzo parametro rappresenta la quantità di elementi dell'array che viene fornito come quarto parametro (di tipo '**regmatch_t**'). L'ultimo parametro rappresenta delle opzioni da applicare in fase di comparazione.

Per comprendere l'utilizzo della funzione, inizialmente conviene lasciare da parte i parametri ***n_match*** e ***pmatch***. Così facendo è possibile verificare se l'espressione regolare trova una corrispondenza nella stringa fornita. L'esempio seguente (che dovrebbe essere disponibile presso *allegati/c/esempio-posix-regex-1.c*) mostra la dichiarazione di una funzione che svolge tutti i passaggi, dalla compilazione alla liberazione della memoria.

Listato 68.10. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/t4ZYxS5l* , *http://ideone.com/oN1I1* .

```
#include <stdio.h>
#include <regex.h>

int
regex_match (char *pattern, const char *string)
{
    int        status;
    regex_t    re;
    status = regcomp (&re, pattern, REG_EXTENDED|REG_NOSUB);
    if (status != 0)
      {
        return (0);
      }
    status = regexec (&re, string, (size_t) 0, NULL, 0);
    if (status != 0)
      {
        regfree (&re);
        return (0);
      }
    regfree (&re);
    return (1);
}

int
main (void)
{
    char *string       = "Ciao amore mio";
    char *re           = "iao";

    if (regex_match (re, string))
      {
```

```
        printf ("Il modello \"%s\" trova corrispondenza ",
                re);
        printf ("nella stringa \"%s\"\n", string);
      }
    else
      {
        printf ("Il modello \"%s\" ", re);
        printf ("NON trova corrispondenza ");
        printf ("nella stringa \"%s\"\n", string);
      }
    return 0;
}
```

In pratica, la funzione ***regexec()*** viene usata semplicemente fornendo il puntatore alla variabile strutturata contenente l'espressione regolare compilata e la stringa da confrontare:

```
regexec (&re, string, (size_t) 0, NULL, 0);
```

Le opzioni che possono essere indicate alla funzione ***regexec()***, come ultimo argomento, sono solo due e riguardano la facoltà di considerare l'inizio o la fine della stringa come l'inizio o la fine di una riga.

Tabella 68.12. Macro-variabili simboliche da usare come opzioni per la comparazione di una stringa con un'espressione regolare.

Macro-variabile	Significato
`REG_NOTBOL`	In condizioni normali, il carattere '`^`' trova corrispondenza con l'inizio di una stringa. Con questa opzione, si inibisce tale corrispondenza (*not begin of line*).

Macro-variabile	Significato
`REG_NOTEOL`	In condizioni normali, il carattere '`$`' trova corrispondenza con la fine di una stringa. Con questa opzione, si inibisce tale corrispondenza (*not end of line*).

Il valore restituito dalla funzione ***regexec()*** è zero se il confronto avviene con successo, diversamente si ha un valore diverso da zero, per indicare la mancanza di corrispondenza o l'utilizzo eccessivo di memoria. Va osservato che la macro-variabile ***REG_ESPACE*** è la stessa già vista per la funzione ***regcomp()*** e che ***REG_NOTBOL*** rappresenta, opportunamente, un valore differente da tutte le altre macro-variabili ***REG_***….

Tabella 68.13. Macro-variabili simboliche che rappresentano il tipo di errore (o di insuccesso) restituito dalla funzione ***regexec()***.

Macro-variabile	Significato
`REG_ESPACE`	Il procedimento di confronto porta all'esaurimento della memoria disponibile.
`REG_NOMATCH`	Non c'è corrispondenza tra espressione regolare e stringa.

68.2.5 Estrapolazione di sottostringhe

Quando un'espressione regolare contiene una porzione del proprio codice racchiuso tra '`\(`' e '`\)`', nel caso di espressioni BRE, oppure tra '`(`' e '`)`', nel caso di espressioni ERE, è possibile estrapolare la porzione di stringa che corrisponde a tale sottoespressione. Per fare

questo si usa un array di variabili strutturate di tipo '**`regmatch_t`**' che viene fornito come argomento della chiamata di ***regexec()***.

Della variabile strutturata di tipo '**`regmatch_t`**' si sa solo che contiene almeno due campi, denominati '**`rm_so`**' e '**`rm_eo`**' (*regular expression match*: *start offset* e *end offset*). I due campi in questione, sono, a loro volta, di tipo '**`regoff_t`**', corrispondente a un valore intero con segno, di rango appropriato.

Tabella 68.14. Organizzazione di una variabile strutturata di tipo '**`regmatch_t`**'.

Tipo	Nome del membro	Descrizione
`regoff_t`	`rm_so`	Scostamento in byte, dall'inizio della stringa, corrispondente all'inizio della sottostringa individuata.
`regoff_t`	`rm_eo`	Scostamento in byte, dall'inizio della stringa, corrispondente al carattere **successivo** alla sottostringa individuata.

La funzione ***regexec()*** popola il contenuto dell'array di elementi '**`regmatch_t`**', utilizzando il primo elemento (indice 0) per individuare la sottostringa corrispondente all'espressione regolare nel suo complesso, mentre gli elementi successivi riguardano le sottoespressioni eventuali. Pertanto, le sottoespressioni si trovano a partire dall'indice 1 di tale array.

Perché la funzione ***regexec()*** possa estrapolare delle sottostringhe a partire da sottoespressioni, è necessario che l'espressione regolare sia stata compilata senza l'opzione '**REG_NOSUB**'. Infatti, tale opzione viene usata per risparmiare risorse quando si sa che non ci si intende avvalere di tale possibilità.

L'esempio seguente mostra un piccolo programma, completo, in cui la funzione ***regex_match()*** si occupa di verificare la corrispondenza con un'espressione regolare e, se c'è corrispondenza, compila anche un array si stringhe con le sottostringhe estratte. Naturalmente, tale array di stringhe deve essere già stato predisposto prima della chiamata della funzione e deve avere una dimensione adeguata a contenere sia la corrispondenza con l'espressione regolare nel suo complesso, sia la corrispondenza con le altre sottoespressioni eventuali. Il file dovrebbe essere disponibile presso *allegati/c/esempio-posix-regex-2.c* .

Listato 68.15. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/ymVOGkON* , *http://ideone.com/U9Tka* .

```
#include <stdio.h>
#include <regex.h>

int
regex_match (char *pattern, const char *string,
             size_t sub_size, char **sub)
{
    regmatch_t match[sub_size];
    size_t     n;
    size_t     m;
    size_t     d;
```

```
    int         status;
    regex_t     re;
    status = regcomp (&re, pattern, REG_EXTENDED);
    if (status != 0)
      {
        return (status);
      }
    status = regexec (&re, string, sub_size, match, 0);
    if (status == 0)
      {
        for (n=0; n < sub_size; n++)
          {
            for (d = 0, m = match[n].rm_so;
                 m >= 0 && m < match[n].rm_eo;
                 m++, d++)
              {
                sub[n][d] = string[m];
              }
            sub[n][d] = '\0';
          }
      }
    regfree (&re);
    return (status);
}

int
main (void)
{
    int   result;
    char *string       = "Ciao amore mio";
    char *re           = "Ciao (amo)re";
    char  sub0[200];
          sub0[0] = '\0';
    char  sub1[200];
```

```
            sub1[0] = '\0';
    char *sub[] = {sub0, sub1};

    result = regex_match (re, string, 2, sub);

    if (result == 0)
      {
        printf ("Il modello \"%s\" trova corrispondenza ",
                re);
        printf ("nella stringa \"%s\", precisamente ",
                string);
        printf ("nella porzione \"%s\", mentre la ",
                sub[0]);
        printf ("sottostringa estratta è \"%s\".\n",
                sub[1]);
      }
    else
      {
        printf ("Il modello \"%s\" ", re);
        printf ("NON trova corrispondenza ");
        printf ("nella stringa \"%s\"\n", string);
      }
    return 0;
}
```

Compilando il programma ed eseguendolo con i dati che si vedono, si ottiene la visualizzazione del testo seguente:

```
Il modello "Ciao (amo)re" trova corrispondenza nella stringa
"Ciao amore mio", precisamente nella porzione "Ciao amore",
mentre la sottostringa estratta è "amo".
```

68.2.6 Informazioni diagnostiche

«

Le funzioni ***regcomp()*** e ***regexec()*** restituiscono un valore intero che rappresenta l'esito dell'operazione svolta: se è zero l'operazione ha avuto successo, altrimenti c'è un qualche tipo di problema che può essere individuato confrontando tale valore con una serie di macro-variabili prestabilite. Tuttavia, si può ottenere un risultato tradotto in un testo più comprensibile attraverso la funzione ***regerror()***, la quale richiede l'indicazione del numero dell'errore, del puntatore alla variabile strutturata che rappresenta l'espressione regolare a cui si riferisce il problema e le informazioni necessarie a compilare correttamente una stringa con il messaggio appropriato.

```
size_t regerror (int errore, const regex_t *restrict re,
                 char *restrict testo, size_t dimensione);
```

In pratica, il primo parametro è il numero dell'errore o comunque dell'esito dell'operazione svolta, come restituito dalle funzioni ***regcomp()*** e ***regexec()***; il secondo parametro è il puntatore alla variabile strutturata che rappresenta l'espressione regolare a cui si riferisce l'esito in questione; il terzo parametro è un array di tipo '**char**', in cui la funzione deve poter scrivere il testo della spiegazione; l'ultimo parametro è la dimensione massima di tale array (oltre la quale la funzione non deve scrivere).

Il valore restituito dalla funzione ***regerror()*** è la dimensione utilizzata effettivamente nell'array per scrivere il testo dell'esito (inclusa la terminazione con il byte a zero).

L'esempio seguente mostra un piccolo programma, completo, ottenuto dalla modifica di quello apparso nella sezione precedente, dove

in presenza di un esito non soddisfacente per le funzioni ***regcomp()*** e ***regexec()*** viene visualizzato un messaggio esplicito del problema verificatosi. Il programma richiede volutamente un confronto non corretto per produrre un errore. Il file dovrebbe essere disponibile presso *allegati/c/esempio-posix-regex-3.c* .

Listato 68.17. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/onqxRRyX* , *http://ideone.com/Q0w9k* .

```
#include <stdio.h>
#include <regex.h>

int
regex_match (char *pattern, const char *string,
             size_t sub_size, char **sub)
{
    const int msg_size = 200;
    char      msg[msg_size];
              msg[0] = '\0';
    //
    regmatch_t match[sub_size];
    size_t     n;
    size_t     m;
    size_t     d;
    int        status;
    regex_t    re;
    status = regcomp (&re, pattern, REG_EXTENDED);
    if (status != 0)
      {
        regerror (status, &re, msg, msg_size);
        fprintf (stderr, "%s\n", msg);
        return (status);
      }
    status = regexec (&re, string, sub_size, match, 0);
```

```
    if (status != 0)
      {
        regerror (status, &re, msg, msg_size);
        fprintf (stderr, "%s\n", msg);
      }
    else
      {
        for (n=0; n < sub_size; n++)
          {
            for (d = 0, m = match[n].rm_so;
                 m >= 0 && m < match[n].rm_eo;
                 m++, d++)
              {
                sub[n][d] = string[m];
              }
            sub[n][d] = '\0';
          }
      }
    regfree (&re);
    return (status);
}

int
main (void)
{
    int   result;
    char *string       = "Ciao amore mio";
    char *re           = "*Ciao (amo)re";
    char  sub0[200];
          sub0[0] = '\0';
    char  sub1[200];
          sub1[0] = '\0';
    char *sub[] = {sub0, sub1};
```

```
    result = regex_match (re, string, 2, sub);

    if (result == 0)
      {
        printf ("Il modello \"%s\" trova corrispondenza ",
                re);
        printf ("nella stringa \"%s\", precisamente ",
                string);
        printf ("nella porzione \"%s\", mentre la ",
                sub[0]);
        printf ("sottostringa estratta è \"%s\".\n",
                sub[1]);
      }
    else
      {
        printf ("Il modello \"%s\" ", re);
        printf ("NON trova corrispondenza ");
        printf ("nella stringa \"%s\"\n", string);
      }
    return 0;
}
```

L'espressione regolare contiene un errore che consiste nell'uso dell'asterisco all'inizio della stessa. Provando a eseguire il programma si dovrebbe visualizzare il testo seguente:

```
Invalid preceding regular expression
Il modello "*Ciao (amo)re" NON trova corrispondenza nella stringa
"Ciao amore mio"
```

68.2.7 Esempio completo

«

Viene proposta una funzione per semplificare il confronto di una stringa con un'espressione regolare, più completa rispetto a quanto già mostrato negli esempi delle sezioni precedenti. A sua volta la funzione viene mostrata in un programma di prova completo.

```
int regex_match (char *restrict pattern,
                 const char *restrict string,
                 size_t sub_size,
                 char *restrict sub[restrict],
                 int cflags, int eflags, int verbose);
```

Il primo parametro è la stringa che contiene l'espressione regolare; il secondo parametro è la stringa da confrontare con l'espressione; il terzo parametro è la dimensione massima dell'array che costituisce il quarto parametro; il quarto parametro è un array di puntatori a stringhe, di cui però non si conosce l'ampiezza massima; il quinto parametro è un intero che rappresenta le opzioni da usare con la funzione ***regcomp()***; il sesto parametro è un intero che rappresenta le opzioni da usare con la funzione ***regexec()***; l'ultimo parametro, se diverso da zero, richiede la visualizzazione dei messaggi di errore attraverso lo standard error.

La funzione restituisce un valore pari a zero se tutto il procedimento si completa con successo; altrimenti restituisce l'esito prodotto dalla funzione ***regcomp()*** o da ***regexec()***.

Si osservi che all'inizio del programma è possibile definire la macro-variabile '**`restrict`**' come commento. Ciò è necessario se il compilatore non riconosce ancora tale parola chiave nella definizione dei

parametri che sono puntatori. Infatti, si tratta di una caratteristica utile solo nei compilatori ottimizzati, in grado di gestire l'elaborazione degli array in modo diverso da quello tradizionale. Il file dovrebbe essere disponibile presso *allegati/c/esempio-posix-regex-ok.c* .

Listato 68.19. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/c06Z8k6Z* , *http://ideone.com/J0O8h* .

```
#include <stdio.h>
#include <regex.h>
//
// #define restrict /**/
//
//
int
regex_match (char *restrict pattern,
             const char *restrict string,
             size_t sub_size, char *restrict sub[restrict],
             int cflags, int eflags, int verbose)
{
    const int  msg_size = 200;
    char       msg[msg_size];
    //
    regmatch_t pmatch[sub_size];
    size_t     n;
    size_t     m;
    size_t     d;
    //
    int        status;
    regex_t    re;
    //
    status = regcomp (&re, pattern, cflags);
    //
    if (status != 0)
```

```
    {
      if (verbose)
        {
          regerror (status, &re, msg, msg_size);
          fprintf (stderr, "%s\n", msg);
        }
      return (status);
    }
  //
  status = regexec (&re, string, sub_size, pmatch,
                    eflags);
  //
  if (status != 0)
    {
      if (verbose)
        {
          regerror (status, &re, msg, msg_size);
          fprintf (stderr, "%s\n", msg);
        }
    }
  else
    {
      for (n=0; n < sub_size; n++)
        {
          for (d = 0, m = pmatch[n].rm_so;
               m >= 0 && m < pmatch[n].rm_eo;
               m++, d++)
            {
              sub[n][d] = string[m];
            }
          sub[n][d] = '\0';
        }
    }
  //
```

```
    regfree (&re);
    //
    return (status);
}
//
//
//
int
main (void)
{
    int   result;
    char *string       = "Ciao amore mio";
    char *re           = "Ciao (amo)re";
    char  sub0[200];
          sub0[0] = '\0';
    char  sub1[200];
          sub1[0] = '\0';
    char *sub[] = {sub0, sub1};
    //
    result = regex_match (re, string, 2, sub, REG_EXTENDED,
                          0, 1);
    //
    if (result == 0)
      {
        printf ("Il modello \"%s\" trova corrispondenza ",
                re);
        printf ("nella stringa \"%s\", precisamente ",
                string);
        printf ("nella porzione \"%s\", mentre la ",
                sub[0]);
        printf ("sottostringa estratta è \"%s\".\n",
                sub[1]);
      }
    else
```

```
        {
          printf ("Il modello \"%s\" ", re);
          printf ("NON trova corrispondenza ");
          printf ("nella stringa \"%s\"\n", string);
        }
      return 0;
}
```

68.3 Avvio e conclusione dei processi

«

In un sistema Unix, l'avvio di un processo si ottiene attraverso l'uso di due chiamate di sistema: una produce una copia del processo esistente, con la possibilità di distinguere poi tra chi è il genitore e chi è invece il figlio; l'altra carica un processo e lo mette in funzione al posto di quello da cui ha avuto origine. Inizialmente, la questione può sembrare complicata o almeno strana. Se però si ha la possibilità di approfondire il funzionamento basilare di un sistema Unix tradizionale, si scopre che è tutto perfettamente logico e lineare, ovvero, che si tratta della scelta progettuale più semplice che si potesse attuare.

Attorno a questi concetti ci sono poi altre questioni legate ai processi, che è bene introdurre assieme al resto, per avere una visione iniziale relativamente completa.

68.3.1 Biforcazione: «fork»

«

Attraverso la funzione ***fork()***, definita nel file di intestazione '`unistd.h`', si ottiene la duplicazione del processo elaborativo corrente, associandogli un numero PID differente, diventando questo figlio del processo da cui la chiamata ha avuto origine. Ciò che va

chiarito è che il processo ottenuto dalla duplicazione continua a funzionare dal punto in cui si trovava il processo originario, pertanto è dal valore restituito dalla funzione ***fork()*** che si riesce a capire se ci si trova a funzionare come genitore o figlio di quel contesto particolare.

Listato 68.20. Per provare il codice attraverso un servizio *pastebin*: http://ideone.com/XpVBY .

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
//------------------------------------------------------
int
main (void)
{
    pid_t  pid;
    //
    printf ("Io sono il genitore e il mio numero PID "
            "è %i.\n",
            (int) getpid ());
    pid = fork ();
    if (pid == 0)
      {
        printf ("Io sono il figlio della biforcazione "
                "e il mio genitore ha il numero "
                "PID %i.\n", (int) getppid ());
        exit (0);
      }
    else
      {
        printf ("Ho avviato una biforcazione di me "
                "stesso, la quale ha ottenuto il "
                "numero PID %i.\n", pid);
      }
```

```
    return (0);
}
```

Il listato mostra un esempio completo di programma che avvia una biforcazione di se stesso. La funzione ***fork()*** viene usata senza argomenti e restituisce un numero PID: se questo è diverso da zero, significa che si tratta dell'esecuzione del processo genitore; se invece è zero, è in corso l'esecuzione del processo duplicato. Nell'esempio si fa uso delle funzioni ***getpid()*** e di ***getppid()***, per ottenere, rispettivamente il numero del processo in corso e quello del processo genitore.

Come si può osservare, nell'esempio il processo figlio ha vita breve, perché si limita a dichiarare la propria esistenza, quindi chiama la funzione ***exit()*** per concludere esplicitamente la propria attività.

Il risultato dell'esecuzione di questo programma potrebbe essere costituito dai messaggi seguenti:

```
Io sono il genitore e il mio numero PID è 5531.
Io sono il figlio della biforcazione e il mio genitore ha il ↩
↪numero PID 5531.
Ho avviato una biforcazione di me stesso, la quale ha ↩
↪ottenuto il numero PID 5532.
```

Il processo ottenuto dalla biforcazione è sostanzialmente uguale a quello del genitore (a parte la distinzione del numero PID e della gerarchia genitore-figlio); tuttavia, le differenze emergono in base al livello di complessità del programma in questione. Pertanto, è sempre bene accertarsi nel dettaglio di cosa erediti il processo figlio, dalla pagina di manuale *fork(2)*, quando si vuole usare questa funzione.

Le cose essenziali da sapere riguardano principalmente i file aperti

e i *thread*, ovvero i flussi elaborativi. La biforcazione produce la duplicazione dei file aperti nel nuovo processo, condividendo però l'indice che rappresenta la posizione corrente. In tal modo, le operazioni di lettura e scrittura sui file possono svolgersi in modo coordinato: uno legge fino a un certo punto, l'altro legge da lì fino a un'altra posizione, e lo stesso vale per la scrittura. La biforcazione produce un solo *thread* nel processo figlio, costituito precisamente da quello che ha chiamato la funzione ***fork()***.

68.3.2 Sostituzione: «exec»

Un gruppo di funzioni, contraddistinte dal prefisso '**`exec`**', consente di rimpiazzare il processo corrente con un altro, caricando un programma. Rimpiazzare il processo corrente significa che questo si conclude e, da quel punto, dovrebbe iniziare a funzionare un altro programma dall'inizio.

In condizioni normali, un processo che voglia avviare un programma, esegue prima una biforcazione, quindi, nel codice che riguarda il processo figlio esegue una funzione ***exec*...*()***, con cui quel figlio viene rimpiazzato con il nuovo programma.

Nell'esempio successivo viene mostrato l'uso della funzione ***execl()***, con la quale si indica il percorso del programma da avviare, seguito dagli argomenti da dare a questo, tenendo conto che il primo deve corrispondere al nome del programma stesso e che l'ultimo deve essere un puntatore a carattere nullo:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
//-------------------------------------------------------
int
main (void)
{
    pid_t  pid;
    int    status;
    //
    pid = fork ();
    if (pid == 0)
      {
        status = execl ("./prog", "prog", (char *) NULL);
        perror (NULL);
        exit (0);
      }
    return (0);
}
```

Se il programma '**prog**' viene avviato correttamente dalla directory corrente, come indicato nel percorso di avvio, la funzione ***execl()*** «non ritorna», nel senso che il processo che la avvia scompare. Se invece si verifica un errore, la funzione restituisce il valore -1 e l'esecuzione del processo originario prosegue. In questo caso, si usa la funzione ***perror()*** per visualizzare l'errore annotato nella variabile ***errno***, quindi la funzione ***exit()*** conclude comunque il funzionamento del processo.

68.3.3 Attesa della conclusione di un processo figlio

Quando un processo esegue una biforcazione, dalla quale poi si può passare all'esecuzione di un altro programma o meno, ci può essere la necessità di attendere che il processo figlio termini il suo funzionamento. Per fare questo si usa normalmente la funzione ***wait()***, oppure ***waitpid()*** con argomenti appropriati.

Listato 68.23. Per provare il codice attraverso un servizio *pastebin*: *http://ideone.com/vD2hs* .

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
//-----------------------------------------------------
int
main (void)
{
    pid_t  pid_child;
    pid_t  pid_dead;
    int    status;
    //
    pid_child = fork ();
    if (pid_child == 0)
      {
        sleep (1);
        printf ("ciao!\n");
        exit (7);
      }
    printf ("Ho avviato il processo %i.\n", pid_child);
    //
    pid_dead = wait (&status);
    //
    printf ("Il processo %i si è concluso restituendo "
```

```
            "il valore %x.\n",
            pid_dead, WEXITSTATUS (status));
    //
    return (0);
}
```

L'esempio mostra l'avvio di un processo figlio, in cui, dopo una pausa di un secondo si visualizza un messaggio e quindi quel processo termina restituendo il valore 7. Il processo genitore mostra subito un messaggio in cui dichiara il numero PID del figlio, quindi si mette in attesa della sua conclusione. Il valore restituito dal processo figlio confluisce nella variabile ***status***, ma deve essere interpretato attraverso la macroistruzione ***WEXITSTATUS()***. Il risultato prodotto a video dal programma di esempio mostrato è molto simile al testo seguente:

```
Ho avviato il processo 6138.
ciao!
Il processo 6138 si è concluso restituendo il valore 7.
```

68.3.4 Adozione dei processi

«

I sistemi Unix e di conseguenza lo standard POSIX, seguono una convenzione nella numerazione dei processi: il kernel è il processo zero ed è implicito; il processo numero uno è '**`init`**' (o altro in situazioni particolari) e ha il compito di essere quello che genera tutti gli altri.

Quando un processo termina di funzionare, i suoi processi figli vengono affidati a '**`init`**' (o comunque a quel processo che si trova ad avere il numero uno).

Listato 68.25. Per provare il codice attraverso un servizio *pastebin*: http://ideone.com/Wq5je .

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
//-----------------------------------------------------
int
main (void)
{
    pid_t  pid;
    //
    pid = fork ();
    if (pid == 0)
      {
        printf ("Io sono il figlio della biforcazione e "
                "il mio genitore ha il numero "
                "PID %i.\n", (int) getppid ());
        sleep (2);
        printf ("Sono passati due secondi e il mio "
                "genitore ha il numero PID %i.\n",
                (int) getppid ());
        exit (0);
      }
    else
      {
        printf ("Io sono il processo %i e ho avviato una "
                "biforcazione di me stesso, la quale ha "
                "ottenuto il numero PID %i, ma adesso "
                "concludo il mio funzionamento.\n",
                (int) getpid (), pid);
      }
    return (0);
}
```

L'esempio appena mostrato dovrebbe chiarire questo fatto: quando il processo figlio ha superato l'attesa di due secondi, il suo genitore ha già smesso di funzionare, e in quel momento è già stato adottato dal processo numero uno. I messaggi prodotti dal programma sono come quelli seguenti:

```
Io sono il processo 6602 e ho avviato una biforcazione ↩
↪di me stesso, la quale ha ottenuto il numero ↩
↪PID 6603, ma adesso concludo il mio funzionamento.
Io sono il figlio della biforcazione e il mio genitore ↩
↪ha il numero PID 6602.
Sono passati due secondi e il mio genitore ha il ↩
↪numero PID 1.
```

68.3.5 Gli zombie

«

Quando un processo elaborativo conclude il suo funzionamento, per qualunque motivo e in qualunque modo sia, si presume che debba restituire un valore al proprio genitore. Come descritto in precedenza, la funzione ***wait()*** consente a un genitore di recepire la conclusione di un suo processo figlio, ottenendo anche il valore restituito. Tuttavia, non è detto che un genitore sia sempre lì pronto a recepire la conclusione di un proprio figlio, pertanto, i processi conclusi continuano a rimanere annotati nel sistema, fino a quando le loro informazioni devono rimanere disponibili. Un processo concluso, ma in attesa di essere eliminato, è noto come «zombie».

La conclusione di un processo produce automaticamente l'invio di un segnale '**SIGCHLD**' al genitore. Questo segnale, in particolare, se non viene intercettato, produce l'eliminazione dei processi figli defunti. Tuttavia potrebbe essere utilizzato da un genitore per intervenire contestualmente e recepire la conclusione di un processo figlio,

senza rimanere in attesa con la funzione ***wait()*** per questo, come nell'esempio seguente:

Listato 68.27. Per provare il codice attraverso un servizio *pastebin*: *http://ideone.com/JYqHo* .

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <signal.h>
//-------------------------------------------------------
void signal_handler (int signal)
{
    int   status;
    pid_t pid;
    if (signal == SIGCHLD)
      {
        pid = wait (&status);
        printf ("Il processo %i si è concluso restituendo "
                "il valore %x.\n",
                pid, WEXITSTATUS (status));
      }
}
//-------------------------------------------------------
int
main (void)
{
    pid_t  pid;
    //
    signal (SIGCHLD, signal_handler);
    //
    pid = fork ();
    if (pid == 0)
      {
        sleep (1);
```

```
        printf ("ciao!\n");
        exit (7);
      }
    printf ("Ho avviato il processo %i.\n", pid);
    //
    sleep (60);
    //
    return (0);
}
```

Come già chiarito, quando un processo muore, i suoi figli vengono adottati automaticamente da '**init**', o comunque dal processo numero uno. In questa circostanza, però, '**init**' riceve anche il segnale '**SIGCHLD**', perché i processi adottati potrebbero trovarsi già nello stato di «zombie», ovvero in attesa di essere considerati per poter morire definitivamente.

68.3.6 Variabili di ambiente

«

Una caratteristica dei programmi di un sistema Unix, e quindi POSIX, è la disponibilità di quelle che sono note come variabili di ambiente. Va osservato che questo concetto non è presente nel linguaggio C puro e semplice; inoltre, per la stessa ragione, il prototipo della funzione ***main()*** diventa più articolato rispetto a quello di un programma C comune:

```
int main (int argc, char *argv[], char *envp[]);
```

I parametri ***argc*** e ***argv[]*** sono gli stessi, già conosciuti nel linguaggio C, con l'accortezza di avere l'elemento ***argv[argc]*** pari al puntatore nullo ('**NULL**'). Il parametro ***envp[]*** è inteso come un array di

stringhe, il cui contenuto deve avere la forma '***nome=valore***' e l'ultimo elemento, anche in questo caso, deve essere un puntatore nullo (per poter riconoscere la sua conclusione).

In pratica, l'array ***envp[]*** diventa il veicolo per le variabili di ambiente da fornire al programma che si vuole avviare.

Ma la questione non si esaurisce così, perché per motivi storici l'array di stringhe che descrivono le variabili di ambiente è accessibile anche attraverso una variabile globale (esterna), denominata ***environ***. In tal modo, anche se la funzione ***main()*** non fosse provvista del parametro ***envp[]***, sarebbe comunque possibile accedere alle stringhe delle variabili di ambiente.

```
extern char **environ;
```

Per leggere e modificare ciò che rappresenta le variabili di ambiente, si usano poi delle funzioni apposite, i cui prototipi appaiono nel file di intestazione '`stdlib.h`'. Queste funzioni hanno in comune un nome terminante per «env», come ***setenv()***, ***unsetenv()*** e ***putenv()***.

68.3.7 Variabili di ambiente e avvio di un programma

Quando si avvia un nuovo programma, attraverso una delle funzioni ***exec...()***, questo ottiene un insieme di variabili di ambiente, ereditandole dal processo originario (che viene rimpiazzato), oppure attraverso una dichiarazione esplicita. Ciò dipende da quale funzione ***exec...()*** viene usata effettivamente. La funzione da cui poi hanno origine le altre della famiglia ***exec...()*** è ***execve()***:

```
int execve (const char *path, char *const argv[],
            char *const envp[]);
```

Logicamente, il primo parametro (***path***) rappresenta il percorso del programma da avviare, mentre gli altri due corrispondono agli array di stringhe che di norma hanno lo stesso nome nel prototipo della funzione ***main()***.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
//-------------------------------------------------------
int
main (int argc, char *argv[], char *envp[])
{
    pid_t  pid;
    char  *exec_argv[3];
    char  *exec_envp[3];
    //
    exec_envp[0] = "PATH=/bin:/usr/bin:/sbin:/usr/sbin";
    exec_envp[1] = "CONSOLE=/dev/console";
    exec_envp[2] = NULL;
    //
    exec_argv[0] = "./mio_prog";
    exec_argv[1] = "-x";
    exec_argv[2] = NULL;
    //
    pid = fork ();
    if (pid == 0)
      {
        execve (exec_argv[0], exec_argv, exec_envp);
        perror (NULL);
        exit (1);
      }
```

```
    return (0);
}
```

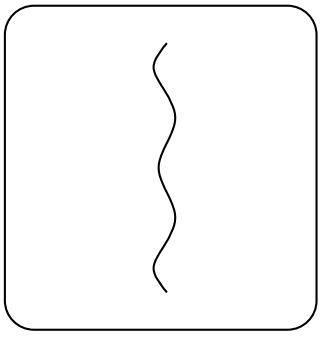
L'esempio mostra la costruzione degli array contenenti le variabili di ambiente e gli argomenti del programma da avviare.

68.4 Nozioni sui thread POSIX

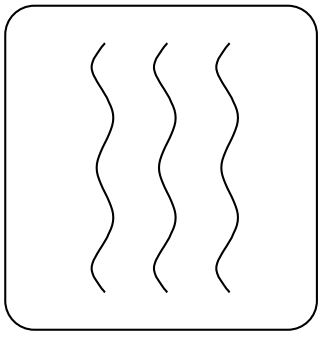
Un programma comune si traduce solitamente in un solo flusso di controllo (o flusso elaborativo), ovvero in un solo ***thread***, nel senso che il procedimento esecutivo è unico, dall'avvio alla conclusione del processo. Un programma più sofisticato, potrebbe gestire gli stessi dati attraverso più flussi di controllo concorrenti e in tal caso si dice che questo utilizza più thread. Pertanto, non va confuso il concetto di processo elaborativo con il flusso di controllo o thread, perché i thread di un processo condividono la stessa memoria, mentre i processi elaborativi, tra di loro, hanno aree di memoria indipendenti.

Il termine inglese thread si traduce letteralmente come «filetto», pertanto viene rappresentato frequentemente in questo modo.

processo elaborativo
a thread singolo



processo elaborativo
con più thread simultanei



La simultaneità di esecuzione dei thread può essere simulata, attraverso la suddivisione del tempo di CPU, oppure può essere anche reale, quando l'elaboratore dispone di più CPU. Tuttavia anche quando si dispone di una sola CPU, l'organizzazione corretta di un programma in più thread può migliorarne le prestazioni.

Lo standard POSIX definisce alcune funzioni per la gestione dei thread, per le quali è necessario includere il file di intestazione '**pthread.h**' (dove la «p» sta per «POSIX»).

In un sistema GNU, o comunque quando si utilizza il compilatore GCC con la libreria dei sistemi GNU, per l'utilizzo delle funzioni che consentono di gestire i thread POSIX, è necessario includere esplicitamente la libreria '**pthread**', con l'opzione '**-lpthread**'. In pratica, per compilare gli esempi di questo capitolo si usano comandi del tipo:

```
$ cc -Wall -lpthread -o file_eseguibile file_sorgente_c [Invio]
```

68.4.1 Numero identificativo del thread

«

Ogni thread ha un proprio numero identificativo, rappresentato attraverso un tipo di dati apposito, denominato '**pthread_t**'. Quando si crea un thread occorre fare riferimento a una variabile di tipo '**pthread_t**', in modo tale che questa sia aggiornata con il numero corretto; successivamente, per ricondurre un thread al flusso principale del processo elaborativo, si utilizza nuovamente quel numero per poterlo individuare.

L'esempio seguente crea la variabile scalare ***mio_thread***, per annotare il numero di un thread:

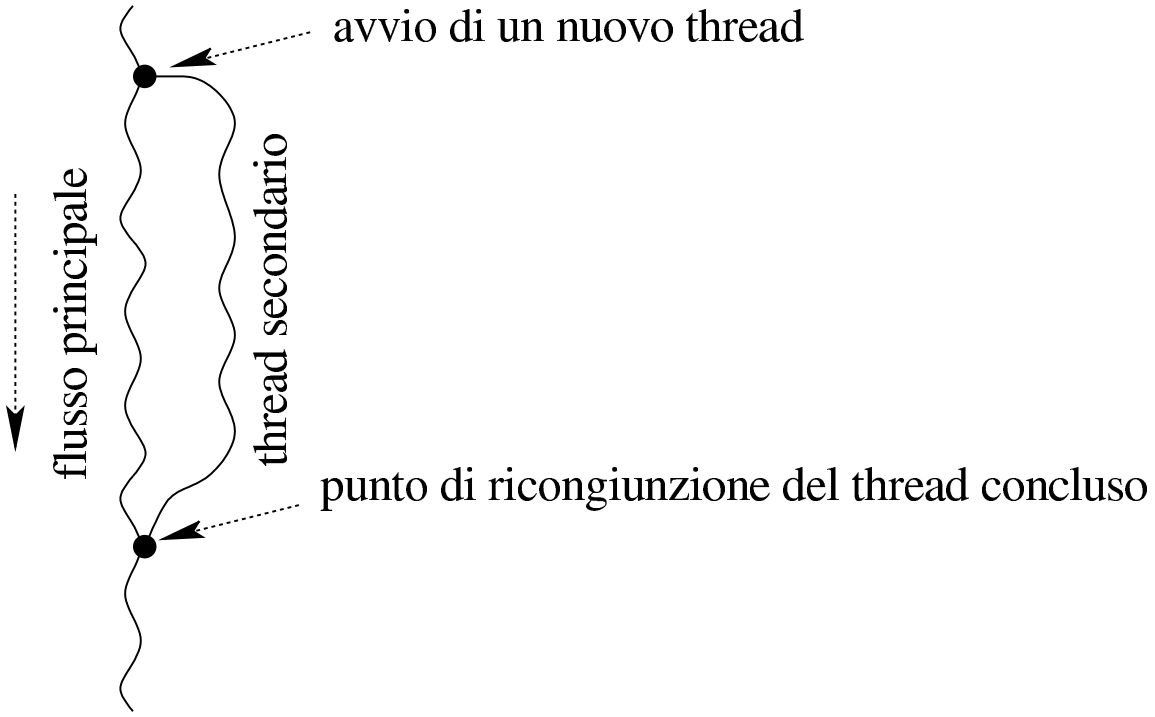
```
pthread_t mio_thread;
```

L'esempio successivo, invece, predispone l'array ***miei_thread[]*** per annotare il numero di identificazione di un massimo di cinque thread:

```
pthread_t miei_thread[5];
```

68.4.2 Creazione e conclusione di un thread aggiuntivo

Un programma ha sempre almeno un thread, ovvero quello principale, la cui creazione è implicita. Tutti gli altri thread che si vogliono gestire vanno creati appositamente: si tratta di fare in modo che una certa funzione sia eseguita senza attenderne la sua conclusione. Ma a un certo punto del flusso principale del programma, è necessario formalizzare la conclusione dei thread aggiuntivi (e se non sono ancora terminati occorre attendere che lo siano effettivamente).



68.4.3 Caratteristiche della funzione che costituisce un thread aggiuntivo

«

La creazione di un nuovo thread coincide con l'avvio di una funzione senza attendere la sua conclusione. Tale funzione deve però avere una forma precisa: riceve esattamente un argomento, costituito da un puntatore indefinito ('**`void *`**'), e restituisce un valore, costituito da un puntatore indefinito.

```
void *funzione (void *arg);
```

In pratica, per passare degli argomenti a una funzione di questo tipo, si predispone una struttura con tutto ciò che serve e se ne passa il puntatore; d'altro canto, la funzione deve essere in grado di estrapolare i dati dalla struttura. Come si comprende, tale funzione ha anche difficoltà a restituire un valore, perché può solo produrre un puntatore a qualcosa che deve risultare già definito prima della sua chiamata.

Per comprendere la cosa viene proposto un programma estremamente banale, in cui la funzione ***function()*** si limita a mostrare ripetutamente un certo carattere, in base ai dati forniti attraverso il riferimento a una struttura.

Listato 68.33. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/2ezjFCNy* , *http://ideone.com/xK3xC* .

```
#include <stdio.h>

struct Arguments {
    char x;
    int  max;
```

```
};

void *
function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    for (i = 0; i < arg->max; i++)
      {
        printf ("%c", arg->x);
      }
    return NULL;
}

int
main (void)
{
    struct Arguments arg = {'x', 10};
    function (&arg);
    printf ("\n");
    return (0);
}
```

Come si vede, la funzione deve sapere come si articola la struttura, per poter accedere ai dati che questa contiene. Generalmente, come nel caso dell'esempio, in una funzione di questo tipo non si restituisce alcunché.

68.4.4 Avvio di thread separati e fusione successiva

Per comprendere il meccanismo di avvio di un thread separato e della sua fusione successiva, viene proposto un esempio molto semplice, con cui si mostrano solo i passaggi indispensabili. Per la preci-

sione, oltre al flusso principale, vengono avviati tre thread ulteriori, attraverso la stessa funzione. Nell'esempio, la funzione usata per avviare i thread, riceve gli argomenti tramite una struttura articolata nello stesso modo già visto nella sezione precedente. Il file dovrebbe essere disponibile presso *allegati/c/esempio-posix-thread-1.c* :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

struct Arguments {
    char x;
    int  max;
};

void *
pthread_function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    long int m = (rand () / 10);

    for (i = 0; i < (arg->max * m); i++)
      {
        if ((i % m) == 0)
          {
            printf ("%c", arg->x);
            fflush (stdout);
          }
      }
    return NULL;
}

int
```

```
main (void)
{
    pthread_t pthread_1;
    pthread_t pthread_2;
    pthread_t pthread_3;

    struct Arguments arg_1 = {'a', 12};
    struct Arguments arg_2 = {'b',  6};
    struct Arguments arg_3 = {'c',  3};

    int status_1;
    int status_2;
    int status_3;

    srand (1234567);

    status_3 = pthread_create (&pthread_3, NULL,
                               pthread_function, &arg_3);
    status_2 = pthread_create (&pthread_2, NULL,
                               pthread_function, &arg_2);
    status_1 = pthread_create (&pthread_1, NULL,
                               pthread_function, &arg_1);

    if ((status_1 + status_2 + status_3) != 0)
      {
        fprintf (stderr, "Errore nella creazione "
                         "dei \"thread\"!\n");
        abort ();
      }

    status_1 = pthread_join (pthread_1, NULL);
    status_2 = pthread_join (pthread_2, NULL);
    status_3 = pthread_join (pthread_3, NULL);
```

```
    if ((status_1 + status_2 + status_3) != 0)
      {
        fprintf (stderr, "Errore nella fusione "
                         "dei \"thread\"!\n");
        abort ();
      }

    return (0);
}
```

All'inizio della funzione ***main()*** si può vedere la dichiarazione di tre variabili di tipo '**pthread_t**', ognuna delle quali viene usata per annotare un numero di identificazione di un thread. Con la chiamata alla funzione ***pthread_create()*** vengono avviati i thread, indicando il riferimento alla variabile da usare per annotare il thread rispettivo, il riferimento alla funzione da avviare e il riferimento alla struttura contenente gli argomenti per tale funzione. Il prototipo seguente è semplificato, per facilitarne la lettura:

```
int pthread_create (pthread_t *tid, pthread_attr_t *attr,
                    void *(*funzione) (void *),
                    void *argomenti);
```

Generalmente, si usa la funzione ***pthread_create()*** senza specificare attributi particolari per il thread che si vuole creare, pertanto si utilizza semplicemente come secondo argomento il valore '**NULL**'.

La funzione ***pthread_create()*** restituisce un valore intero, dove lo zero manifesta il successo dell'operazione, mentre un valore differente indica un problema, decodificabile attraverso il confronto con delle macro-variabili prestabilite. La funzione che avvia il thread for-

nisce il numero dello stesso modificando il contenuto della variabile a cui si riferisce il puntatore fornito come primo argomento.

Una volta accertato che i thread sono stati creati con successo (diversamente il programma termina di funzionare, attraverso la chiamata della funzione ***abort()***), non essendoci altro da fare in questo esempio, viene richiesto di attendere la loro conclusione, attraverso la chiamata della funzione ***pthread_join()***. Tale funzione richiede di indicare il numero del thread di cui si vuole attendere la conclusione, oltre a un puntatore, utile per raggiungere il valore che potrebbe essere restituito dalla funzione che costituiva il thread.

```
int pthread_join (pthread_t tid, void **valore);
```

In pratica, la funzione ***pthread_join()*** sospende l'esecuzione del flusso principale, fino a quando il thread individuato dal numero fornito come primo argomento si conclude (si osservi che in questo caso il numero del thread viene fornito come valore e non più come puntatore). Se il thread non deve produrre alcun risultato utile, il secondo argomento di ***pthread_join()*** può essere il valore nullo ('**NULL**'), altrimenti si deve indicare un puntatore generico, a una variabile che contiene a sua volta un puntatore: tale variabile deve essere quella usata dalla funzione che costituiva il thread per porvi al suo interno il puntatore al risultato dell'elaborazione.

La funzione ***pthread_join()*** restituisce un valore intero, dove lo zero indica il successo dell'operazione, mentre un valore diverso rappresenta un problema, individuabile attraverso il confronto con delle macro-variabili prestabilite.

Il programma di esempio, dopo la fusione dei thread e dopo il con-

trollo dell'esito di tale fusione, si conclude semplicemente. Va osservato che la fusione dei thread è necessaria anche in questo caso, perché il programma non può concludersi (attraverso la fine del flusso principale) prima che tutti i thread accessori siano stati fusi.

La funzione utilizzata per i thread dell'esempio, ovvero ***pthread_function()***, trova un numero casuale abbastanza grande e lo usa per eseguire un ciclo per un numero molto elevato di volte. Al primo ciclo, e poi anche ogni volta che l'indice del ciclo risulta divisibile per il numero casuale trovato, mostra una lettera sullo schermo. A questo proposito, va osservato l'uso della funzione ***fflush()*** per garantire che la lettera emessa attraverso lo standard output venga visualizzata subito, senza rimanere in attesa nella memoria tampone.

I thread dell'esempio vengono avviati con insiemi di dati differenti, in modo che: il thread ***pthread_3*** emetta la lettera «c» per tre volte, il thread ***pthread_2*** emetta la lettera «b» per sei volte e che il thread ***pthread_1*** emetta la lettera «a» per dodici volte.

Il risultato visibile sullo schermo assomiglia a una sequenza come questa:

```
cbaaaaaabcaaaabacabbb
```

68.4.5 Conflitto nell'accesso ai dati

«

Quando un thread opera su dati propri (a cui nessun altro thread, nemmeno quello principale, accede in scrittura), tutto fila liscio senza preoccupazioni. Ma la realtà richiede generalmente che i thread si scambino dei dati, pertanto, quando si aggiorna un'informazione, occorre un modo per escludere gli altri thread dall'interferire.

Nell'esempio successivo si crea volutamente una situazione di conflitto tra alcuni thread che modificano simultaneamente una variabile, denominata ***global***, il cui scopo sarebbe quello di contare i caratteri mostrati sullo schermo. Il file dovrebbe essere disponibile presso *allegati/c/esempio-posix-thread-2.c* .

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

struct Arguments {
    char x;
    int  max;
};

int global;

void *
pthread_function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    long int m = (rand () / 10);
    int g;

    for (i = 0; i < (arg->max * m); i++)
      {
        g = global;
        if ((i % m) == 0)
          {
            printf ("%c", arg->x);
            fflush (stdout);
            g++;
            global = g;
```

```
            }
        }
    return NULL;
}

int
main (void)
{
    pthread_t pthread_1;
    pthread_t pthread_2;
    pthread_t pthread_3;

    struct Arguments arg_1 = {'a', 12};
    struct Arguments arg_2 = {'b',  6};
    struct Arguments arg_3 = {'c',  3};

    int status_1;
    int status_2;
    int status_3;

    global = 0;

    srand (1234567);

    status_3 = pthread_create (&pthread_3, NULL,
                               pthread_function, &arg_3);
    status_2 = pthread_create (&pthread_2, NULL,
                               pthread_function, &arg_2);
    status_1 = pthread_create (&pthread_1, NULL,
                               pthread_function, &arg_1);

    if ((status_1 + status_2 + status_3) != 0)
      {
        fprintf (stderr, "Errore nella creazione "
```

```
                          "dei \"thread\"!\n");
        abort ();
      }

    status_1 = pthread_join (pthread_1, NULL);
    status_2 = pthread_join (pthread_2, NULL);
    status_3 = pthread_join (pthread_3, NULL);

    if ((status_1 + status_2 + status_3) != 0)
      {
        fprintf (stderr, "Errore nella fusione "
                          "dei \"thread\"!\n");
        abort ();
      }

    printf ("\n");
    printf ("La variabile globale ha raggiunto "
            "il valore %i.\n", global);

    return (0);
}
```

Provando a eseguire il programma di esempio, si potrebbero osservare messaggi molto simili a quelli seguenti:

```
cbaaaaaacbaaaaabcabbb
La variabile globale ha raggiunto il valore 19.
```

In questo caso la variabile globale che viene modificata dalla funzione ***pthread_function()*** ha raggiunto solo il valore 19, mentre il valore atteso sarebbe di 21 (essendo visualizzati 21 caratteri sullo schermo). Naturalmente può succedere che il valore ottenuto dalla variabile sia corretto, ma non ci si può contare, perché non è possibile

prevedere la sequenza effettiva delle operazioni.

Naturalmente, si può migliorare la funzione ***pthread_function()*** per ridurre al minimo la possibilità di accavallamenti tra le attività dei vari thread, ma anche così non si può avere la garanzia di evitare i conflitti:[2]

```
void *
pthread_function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    long int m = (rand () / 10);
    int g;
    //
    for (i = 0; i < (arg->max * m); i++)
      {
        if ((i % m) == 0)
          {
            printf ("%c", arg->x);
            fflush (stdout);
            g = global;
            g++;
            global = g;
          }
      }
    return NULL;
}
```

68.4.6 Accesso alle risorse in modo mutualmente esclusivo

«

Per accedere simultaneamente, in modo ordinato, a dati condivisi, occorre definire dei ***mutex***, ovvero delle variabili che hanno il ruolo di «lucchetto» per definire un accesso mutualmente esclusivo a

una certa area di dati. In altri termini, una volta definita una certa attività da svolgere in modo esclusivo, in un certo insieme di dati, gli si associa una variabile speciale con funzione di mutex (lucchetto mutualmente esclusivo) e prima di entrare nella zona critica che richiede un accesso esclusivo a quell'insieme di dati, si cerca di ottenere tale esclusività con una funzione che interroga e modifica la variabile mutex.

```
…
pthread_mutex_t mutex_1 = PTHREAD_MUTEX_INITIALIZER;

void *
pthread_function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    long int m = (rand () / 1000);
    int g;

    for (i = 0; i < (arg->max * m); i++)
      {
        pthread_mutex_lock (&mutex_1);
        g = global;
        if ((i % m) == 0)
          {
            printf ("%c", arg->x);
            fflush (stdout);
            g++;
            global = g;
          }
        pthread_mutex_unlock (&mutex_1);
      }
    return NULL;
}
```

```
...
```

L'estratto di esempio appena mostrato mette in evidenza le modifiche da apportare per gestire il meccanismo di accesso mutualmente esclusivo. In questo caso la porzione di codice da eseguire in modo mutualmente esclusivo va dalla lettura della variabile globale alla sua modifica successiva: nell'esempio le operazioni sono tenute distanti per dimostrare il funzionamento, dato che sarebbe meglio ridurre al minimo il tempo in cui un thread blocca un mutex.

La variabile globale '**`mutex_1`**' viene dichiarata di tipo '**`pthread_mutex_t`**' (presumibilmente si tratta di una struttura) e viene inizializzata attraverso una macro-variabile appropriata alle sue caratteristiche. Successivamente, prima di entrare nella zona critica, il thread deve richiedere l'accesso esclusivo attraverso la funzione ***pthread_mutex_lock()***, specificando il riferimento alla variabile che costituisce il mutex del contesto. Quando il thread ottiene l'accesso esclusivo può riprendere la sua esecuzione e, quando non ha più bisogno di impegnare il mutex, lo libera, con la funzione ***pthread_mutex_unlock()***.

Per completezza viene mostrato il programma di esempio completo. Il file dovrebbe essere disponibile presso *allegati/c/esempio-posix-thread-3.c* .

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

struct Arguments {
    char x;
```

```
    int  max;
};

int global;

pthread_mutex_t mutex_1 = PTHREAD_MUTEX_INITIALIZER;

void *
pthread_function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    long int m = (rand () / 1000);
    int g;
    //
    for (i = 0; i < (arg->max * m); i++)
      {
        pthread_mutex_lock (&mutex_1);
        g = global;
        if ((i % m) == 0)
          {
            printf ("%c", arg->x);
            fflush (stdout);
            g++;
            global = g;
          }
        pthread_mutex_unlock (&mutex_1);
      }
    return NULL;
}

int
main (void)
{
```

```
    pthread_t pthread_1;
    pthread_t pthread_2;
    pthread_t pthread_3;

    struct Arguments arg_1 = {'a', 12};
    struct Arguments arg_2 = {'b',  6};
    struct Arguments arg_3 = {'c',  3};

    int status_1;
    int status_2;
    int status_3;

    global = 0;

    srand (1234567);

    status_3 = pthread_create (&pthread_3, NULL,
                               pthread_function, &arg_3);
    status_2 = pthread_create (&pthread_2, NULL,
                               pthread_function, &arg_2);
    status_1 = pthread_create (&pthread_1, NULL,
                               pthread_function, &arg_1);

    if ((status_1 + status_2 + status_3) != 0)
      {
        fprintf (stderr, "Errore nella creazione "
                         "dei \"thread\"!\n");
        abort ();
      }

    status_1 = pthread_join (pthread_1, NULL);
    status_2 = pthread_join (pthread_2, NULL);
    status_3 = pthread_join (pthread_3, NULL);
```

```
    if ((status_1 + status_2 + status_3) != 0)
      {
        fprintf (stderr, "Errore nella fusione "
                         "dei \"thread\"!\n");
        abort ();
      }

    printf ("\n");
    printf ("La variabile globale ha raggiunto "
            "il valore %i.\n", global);

    return (0);
}
```

Eseguendo il programma si può osservare che non si creano più accavallamenti nella scrittura della variabile globale e il risultato finale è sempre corretto:

```
cbaaaaaabcaaaaaabcbbb
La variabile globale ha raggiunto il valore 21.
```

Come già descritto nella sezione precedente, è sicuramente meglio ridurre al minimo la zona critica, in modo che anche con l'ausilio delle variabili mutex non sia penalizzata la simultaneità di esecuzione dei thread:

```
void *
pthread_function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    long int m = (rand () / 10000);
    int g;
```

```
    for (i = 0; i < (arg->max * m); i++)
      {
        if ((i % m) == 0)
          {
            printf ("%c", arg->x);
            fflush (stdout);
            pthread_mutex_lock (&mutex_1);
            g = global;
            g++;
            global = g;
            pthread_mutex_unlock (&mutex_1);
          }
      }
    return NULL;
}
```

68.4.7 Accesso esclusivo, ma condizionato

«

Può darsi che l'accesso esclusivo a una zona critica debba avvenire solo al verificarsi di una certa condizione. In altri termini, può darsi che prima di intervenire effettivamente in un certo insieme di dati, un thread debba attendere che questi siano pronti. Per ottenere questo risultato, generalmente, a fianco della variabili mutex, si associano delle variabili che rappresentano il verificarsi di una certa condizione, da gestire anche queste attraverso funzioni apposite dei thread POSIX.

L'estratto seguente mostra le modifiche importanti agli esempi già apparsi, per produrre una situazione in cui i thread devono attendere il verificarsi di una condizione per procedere con il loro intervento nell'area critica:

```
struct Arguments {
    char x;
    int  max;
    int  delay;
};

int global;

pthread_mutex_t mutex_1 = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond_1  = PTHREAD_COND_INITIALIZER;

void *
pthread_function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    long int m = (rand () / 10000);
    int g;

    for (i = 0; i < (arg->max * m); i++)
      {
        if ((i % m) == 0)
          {
            pthread_mutex_lock (&mutex_1);
            while (global < arg->delay)
              {
                pthread_cond_wait (&cond_1, &mutex_1);
              }
            printf ("%c", arg->x);
            fflush (stdout);
            g = global;
            g++;
            global = g;
            pthread_cond_broadcast (&cond_1);
```

```
            pthread_mutex_unlock (&mutex_1);
          }
        }
      return NULL;
}
```

Questa volta, la struttura che costituisce gli argomenti della funzione ***pthread_function()*** ha un'informazione in più, che rappresenta un ritardo da inserire prima di iniziare a mostrare i caratteri sullo schermo. In pratica, se la variabile globale ha raggiunto o superato quel tale valore, il thread può procedere con il proprio lavoro, altrimenti deve rimanere in attesa.

Per ottenere questo risultato, la variabile globale '**`cond_1`**' viene dichiarata con il tipo '**`pthread_cond_t`**', allo scopo di poter rappresentare le condizioni dei thread, e viene inizializzata con una macrovariabile appropriata alle sue caratteristiche effettive. Il thread, prima cerca di ottenere un accesso esclusivo, quindi, se lo ottiene, inizia un ciclo in attesa del verificarsi della condizione, richiamando ripetutamente la funzione ***pthread_cond_wait()***, con il riferimento alla variabile della condizione e a quella del mutex.

La chiamata della funzione ***pthread_cond_wait()*** fa sì che il thread che aveva ottenuto l'accesso esclusivo venga messo in pausa, a vantaggio di un altro che può così ottenere l'accesso esclusivo alla zona critica. La pausa in cui si trova il primo thread può terminare nel momento in cui viene usata la funzione ***pthread_cond_broadcast()***, con il riferimento alla condizione che aveva prodotto la sospensione e poi anche la funzione ***pthread_mutex_unlock()***.

Il thread che era stato messo in pausa dalla funzione

pthread_cond_wait(), riprende quando tale funzione ha riottenuto l'accesso esclusivo in base alla propria variabile mutex.

Logicamente, occorre fare attenzione a non creare una situazione in per cui tutti i thread si mettono in pausa per qualcosa che non si verifica.

Segue il programma di esempio, completo di tutte le sue parti. Il file dovrebbe essere disponibile presso *allegati/c/esempio-posix-thread-4.c* .

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

struct Arguments {
    char x;
    int  max;
    int  delay;
};

int global;

pthread_mutex_t mutex_1 = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond_1  = PTHREAD_COND_INITIALIZER;

void *
pthread_function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    long int m = (rand () / 10000);
    int g;

    for (i = 0; i < (arg->max * m); i++)
```

```
      {
        if ((i % m) == 0)
          {
            pthread_mutex_lock (&mutex_1);
            while (global < arg->delay)
              {
                pthread_cond_wait (&cond_1, &mutex_1);
              }
            printf ("%c", arg->x);
            fflush (stdout);
            g = global;
            g++;
            global = g;
            pthread_cond_broadcast (&cond_1);
            pthread_mutex_unlock (&mutex_1);
          }
      }
    return NULL;
}

int
main (void)
{
    pthread_t pthread_1;
    pthread_t pthread_2;
    pthread_t pthread_3;

    struct Arguments arg_1 = {'a', 12,  0};
    struct Arguments arg_2 = {'b',  6,  5};
    struct Arguments arg_3 = {'c',  3, 10};

    int status_1;
    int status_2;
    int status_3;
```

```
    global = 0;

    srand (1234567);

    status_3 = pthread_create (&pthread_3, NULL,
                               pthread_function, &arg_3);
    status_2 = pthread_create (&pthread_2, NULL,
                               pthread_function, &arg_2);
    status_1 = pthread_create (&pthread_1, NULL,
                               pthread_function, &arg_1);

    if ((status_1 + status_2 + status_3) != 0)
      {
        fprintf (stderr, "Errore nella creazione "
                         "dei \"thread\"!\n");
        abort ();
      }

    status_1 = pthread_join (pthread_1, NULL);
    status_2 = pthread_join (pthread_2, NULL);
    status_3 = pthread_join (pthread_3, NULL);

    if ((status_1 + status_2 + status_3) != 0)
      {
        fprintf (stderr, "Errore nella fusione "
                         "dei \"thread\"!\n");
        abort ();
      }

    printf ("\n");
    printf ("La variabile globale ha raggiunto "
            "il valore %i.\n", global);
```

```
    return (0);
}
```

Eseguendo il programma si può ottenere un risultato simile a quello seguente:

```
aaaaabbbbbcbccaaaaaaa
La variabile globale ha raggiunto il valore 21.
```

Nel programma di esempio, il thread associato alla variabile '**pthread_1**' può visualizzare subito i suoi caratteri sullo schermo, mentre quello associato a '**pthread_2**' deve attendere che sia stato visualizzato il quinto, mentre quello associato a '**pthread_3**' deve attendere che sia stato visualizzato il decimo. Naturalmente, se tutti i thread avviati dovessero attendere qualche carattere prima di poter iniziare, questi si bloccherebbero a vicenda, irrimediabilmente; inoltre, lo stesso succederebbe se ci fosse anche un solo thread che deve attendere un valore per la variabile '**global**' che non può essere raggiunto senza il proprio apporto.

68.4.8 Osservazioni finali

Lo standard POSIX prevede una discreta quantità di funzioni per la gestione dei thread; pertanto quanto descritto in questo capitolo è solo il minimo indispensabile per comprenderne il meccanismo. In modo particolare, va tenuto in considerazione che per l'inizializzazione delle variabili mutex e di quelle che rappresentano le condizioni, si possono usare funzioni apposite che non sono state descritte.

Il fatto che ci siano thread distinti rispetto a quello principale, ha delle implicazioni anche sull'invio dei segnali e sulla possibilità di una loro conclusione anticipata. Pertanto sono disponibili le funzioni

pthread_kill() e ***pthread_exit()***, rivolte specificatamente ai thread (escluso sempre il flusso principale):

```
int pthread_kill (pthread_t tid, int segnale);
```

```
void pthread_exit (void *valore_da_restituire);
```

Infine può essere utile la funzione ***pthread_self()***, per ottenere il numero identificativo del thread attuale:

```
pthread_t pthread_self (void);
```

68.5 I file secondo i sistemi POSIX

Il linguaggio C, puro e semplice, prevede una gestione dei file basilare, attraverso il tipo derivato '**FILE**', per cui un file aperto è un «flusso», identificato da un puntatore al tipo '**FILE**'. Lo standard dei sistemi Unix comporta un'infrastruttura più articolata per la gestione dei file, al di sotto di quanto già descrive il C, introducendo il concetto di ***descrittore di file***, corrispondente a un numero intero normale positivo. Le funzioni e le macro-variabili principali per l'apertura e il controllo dei file, secondo la mediazione del concetto di descrittore, sono indicati nel file di intestazione '`fcnt.h`' (*file control*), ma per amministrare le caratteristiche dei file, servono le definizioni e le funzioni del file di intestazione '`sys/stat.h`'; inoltre, altre funzioni importanti al riguardo si trovano nel file '`unistd.h`'.

L'apertura di un file, dal punto di vista dei sistemi Unix (e quindi POSIX), implica non solo l'associazione al numero del descrittore, ma anche l'attribuzione di opzioni di funzionamento ed eventualmente un sistema di blocco di porzioni del file. La creazione di un file implica l'attribuzione di permessi, nel rispetto però della maschera dei permessi esistente.

Va osservato che anche i flussi di file standard del linguaggio C, trovano una corrispondenza nello standard POSIX in altrettanti descrittori già assegnati, costituiti precisamente dai primi tre:

Denominazione	flusso di file C	numero del descrittore POSIX
standard input	`stdin`	0
standard output	`stdout`	1
standard error	`stderr`	2

Lo standard POSIX prescrive che i numeri dei descrittori siano assegnati usando sempre il valore libero più piccolo; pertanto, il primo descrittore a essere utilizzato, dato che i primi tre sono impegnati per i flussi standard, è il numero tre e di seguito vanno i successivi.

68.5.1 Apertura e chiusura di un file

«

L'apertura ed eventuale creazione di un file, secondo le convenzioni POSIX, va eseguita utilizzando la funzione ***open()***. Per motivi storici esiste anche la funzione ***creat()*** che però ha meno possibilità di ***open()***, pertanto il suo utilizzo non è indispensabile.

```
int open (const char *file, int oflag [, mode_t mode]);
```

```
int creat (const char *file, mode_t mode);
```

La funzione ***open()*** apre un file, indicato attraverso una stringa che descrive il suo percorso (relativo o assoluto che sia) secondo le convenzioni POSIX e restituisce il numero del suo descrittore; se però restituisce il valore −1, significa che l'operazione non ha avuto successo e di conseguenza è stato modificato il contenuto della variabile ***errno*** (la quale può essere esaminata per determinarne la causa).

Il valore costituito dal parametro ***oflag*** viene ottenuto combinando assieme, con l'operatore OR binario, una serie di macro-variabili definite nel file '`fcntl.h`', tenendo conto che non tutte le combinazioni sono ammissibili simultaneamente. Se si utilizza l'opzione rappresentata dalla macro-variabile ***O_CREAT***, per richiedere la creazione del file, va usato anche il terzo parametro della funzione, con cui si specifica la modalità di creazione dello stesso.

Tabella 68.47. Macro-variabili da utilizzare per combinare il valore del parametro ***oflag***. Di queste macro-variabili, in particolare, ne va scelta una sola e si è obbligati a usarla per specificare la modalità di accesso al file: in sola lettura, in sola scrittura o in entrambi i modi.

Macro-variabile	Significato
`O_RDONLY`	Si richiede l'apertura del file in sola lettura.

Macro-variabile	Significato
`O_WRONLY`	Si richiede l'apertura del file in sola scrittura.
`O_RDWR`	Si richiede l'apertura del file in lettura e scrittura.

Tabella 68.48. Alcune delle macro-variabili da utilizzare per combinare il valore del parametro ***oflag***. Non tutte le combinazioni di queste opzioni sono ammissibili.

Macro-variabile	Significato
`O_APPEND`	Questa opzione fa in modo che la scrittura avvenga sempre in estensione del contenuto già esistente.
`O_CREAT`	Questa opzione richiede la creazione del file, ammesso che non esista già. Nel caso il file sia effettivamente da creare, vale la modalità di creazione specificata con il parametro ***mode***.
`O_TRUNC`	Questa opzione si può associare solo a una richiesta di accesso in scrittura (in sola scrittura o in lettura e scrittura simultaneamente) relativa a un file normale, con lo scopo di azzerarne il contenuto se questo file esiste già.
`O_EXCL`	Questa opzione può essere usata solo in abbinamento a '**`O_CREAT`**' e richiede la creazione del file o il fallimento dell'operazione se questo esiste già, anche nel caso si tratti solo di un collegamento simbolico che punta a un file inesistente.
`O_NOCTTY`	Questa opzione fa sì che se il file indicato corrisponde a un dispositivo di terminale, questo non possa diventare in alcun caso il terminale di controllo abbinato al processo.

Macro-variabile	Significato
`O_NONBLOCK`	Questa opzione richiede di non attendere per la conclusione delle operazioni, ammesso che ciò sia possibile in base al contesto.
`O_SYNC`	Questa opzione richiede che le operazioni di accesso al file avvengano in modo sincrono rispetto all'hardware. Pertanto, in questo modo, le operazioni di scrittura comportano l'attesa che queste siano realizzate effettivamente.

Il parametro ***mode*** riguarda esclusivamente la creazione del file (specificando l'indicatore '**`O_CREAT`**'). In tal caso, si tratta del numero che esprime i permessi da dare al file. Si tratta degli stessi permessi che si indicano con programmi come '**`chmod`**', quando si usa la forma numerica, e si scrivono preferibilmente in base otto. Naturalmente, i permessi indicati vengono poi filtrati attraverso la maschera dei permessi, come se fosse eseguita questa operazione: '**`(`*mode*` & ~`*umask*`)`**'. Se lo si preferisce, al posto di indicare i permessi richiesti, direttamente in forma numerica, ci si può avvalere di macro-variabili dichiarate nel file di intestazione '`sys/stat.h`', come descritto nella tabella successiva.

Tabella 68.49. Macro-variabili per esprimere i permessi da attribuire a un file che si vuole creare.

Macro-variabile	Valore numerico equivalente	Significato
`S_IRWXU`	0700_8	Rappresenta tutti i permessi di lettura, scrittura ed esecuzione o accesso, per l'utente proprietario.

Macro-variabile	Valore numerico equivalente	Significato
`S_IRUSR`	0400_8	Rappresenta il permesso di lettura per l'utente proprietario.
`S_IWUSR`	0200_8	Rappresenta il permesso di scrittura per l'utente proprietario.
`S_IXUSR`	0100_8	Rappresenta il permesso di esecuzione o attraversamento per l'utente proprietario.
`S_IRWXG`	0070_8	Rappresenta tutti i permessi di lettura, scrittura ed esecuzione o accesso, per il gruppo proprietario.
`S_IRGRP`	0040_8	Rappresenta il permesso di lettura per il gruppo proprietario.
`S_IWGRP`	0020_8	Rappresenta il permesso di scrittura per il gruppo proprietario.
`S_IXGRP`	0010_8	Rappresenta il permesso di esecuzione o attraversamento per il gruppo proprietario.
`S_IRWXO`	0007_8	Rappresenta tutti i permessi di lettura, scrittura ed esecuzione o accesso, per gli altri utenti.
`S_IROTH`	0004_8	Rappresenta il permesso di lettura per gli altri utenti.
`S_IWOTH`	0002_8	Rappresenta il permesso di scrittura per gli altri utenti.
`S_IXOTH`	0001_8	Rappresenta il permesso di esecuzione o attraversamento per gli altri utenti.
`S_ISUID`	4000_8	Rappresenta l'attivazione del bit S-UID.
`S_ISGID`	2000_8	Rappresenta l'attivazione del bit S-GID.

Macro-variabile	Valore numerico equivalente	Significato
`S_ISVTX`	1000_8	Rappresenta l'attivazione del bit Sticky.

Come già accennato, la funzione ***creat()*** non è più indispensabile e può essere sostituita da ***open()***, usata nel modo seguente:

```
open (file, O_WRONLY|O_CREAT|O_TRUNC, mode)
```

Segue un esempio molto semplice in cui si apre un file in scrittura, specificando che se non esiste già, questo va creato, con tutti i permessi che la maschera dei permessi esistente consenta di attribuire. In caso di errore, il contenuto della variabile ***errno*** viene considerato con l'aiuto della funzione ***perror()***. Il file dell'esempio dovrebbe essere disponibile presso *allegati/c/esempio-posix-fcntl-open.c* .

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int
main (void)
{
    const char *file = "/tmp/test";
    int fdn;

    fdn = open (file, O_WRONLY|O_CREAT, 07777);

    if (fdn >= 0)
      {
        printf ("Aperto il file \"%s\", ", file);
```

```
        printf ("associandolo al descrittore %i.\n", fdn);
        close (fdn);
      }
    else
      {
        printf ("Non è possibile aprire il file \"%s\"!\n",
                file);
        perror (NULL);
      }

    return (0);
}
```

La funzione ***open()*** richiede l'inclusione del file '`fcntl.h`', nel quale sono dichiarate anche la macro-variabili ***O_WRONLY*** e ***O_CREAT***; ma per la funzione ***close()*** è necessario includere anche il file '`unistd.h`'.

Come si vede, i permessi da attribuire al file che venisse creato sono tutti quelli disponibili (7777_8). Eventualmente, aggiungendo anche l'inclusione del file '`sys/stat.h`', sarebbe possibile indicare tale richiesta attraverso macro-variabili convenzionali:

```
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

    …
    fdn = open (file, O_WRONLY|O_CREAT,
                S_IRUSR|S_IWUSR|S_IRGRP);
    …
```

In questo esempio, però, i permessi richiesti sono minori, corrispondenti al numero 0640_8.

Va osservato che la funzione ***open()*** può aprire ogni tipo di file, ma può creare solo dei file «normali». Per creare directory e altri tipi di file speciali si usano funzioni apposite. D'altro canto, per attribuire a un file dei permessi, è possibile usare la funzione ***chmod()*** e non è strettamente necessario occuparsene nel momento della creazione.

La funzione ***close()***, già mostrata nell'esempio, ha un prototipo molto semplice: richiede l'indicazione del descrittore del file da chiudere e restituisce zero se tutto va bene, altrimenti produce il valore −1 e aggiorna la variabile ***errno***:

```
int close (int descrittore);
```

68.5.2 Lettura e scrittura

Come per le funzioni dello standard C, anche per quelle a livello POSIX si accede al contenuto del file attraverso un indicatore della posizione espresso in byte (con la differenza che non si pone il problema di distinguere tra file di testo e file binari). A ogni descrittore di file sono associate delle informazioni, amministrate in modo trasparente dal sistema operativo, e a queste si accede solo attraverso delle funzioni. Tra queste informazioni si trova anche l'indicatore che consente di determinare la posizione iniziale per la lettura o la scrittura.

A seconda di come viene aperto il file, l'indicatore della posizione che lo riguarda viene inizializzato nel modo più logico, come descritto a proposito della funzione ***open()***. Questo indicatore vie-

ne spostato automaticamente a seconda delle operazioni di lettura e scrittura che si compiono, tuttavia, quando si passa da una modalità di accesso all'altra, è necessario spostare l'indicatore attraverso le istruzioni opportune, in modo da non creare ambiguità.

Per la lettura di un file aperto e qualificato con un descrittore, si può usare la funzione ***read()*** che legge una quantità di byte trattandoli come un array. Si osservi l'esempio seguente:

```
…
char buf[100];
int fdn;
ssize_t dim;
…
fdn = open (…);
…
dim = read (fdn, buf, 100);
…
```

In questo modo si intende leggere 100 byte, collocandoli nell'array ***buf***, con la stessa capacità massima. Naturalmente, non è detto che la lettura abbia successo, o quantomeno non è detto che si riesca a leggere la quantità di elementi richiesta. Il valore restituito dalla funzione rappresenta la quantità di byte letti effettivamente. Se si verifica un qualsiasi tipo di errore che impedisce la lettura, la funzione si limita a restituire -1, mentre lo zero è un risultato valido e indica che la lettura è giunta alla fine del file.

Quando il file viene aperto in lettura, in condizioni normali l'indicatore interno viene posizionato all'inizio del file; quindi, ogni operazione di lettura sposta in avanti il puntatore, in modo che la prossima lettura avvenga a partire dalla posizione immediatamente successiva:

```
…
char buf[100];
int fdn;
ssize_t dim;
…
fdn = open (…);
…
while (1)                  // Ciclo senza fine.
  {
    dim = read (fdn, buf, 100);
    if (dim == 0)
      {
        break;             // Termina il ciclo.
      }
    …
  }
…
```

In questo modo, come mostra l'esempio, viene letto tutto il file a colpi di 100 byte alla volta, tranne l'ultima in cui si ottiene solo quello che resta da leggere.

Analogamente, la scrittura può essere eseguita con la funzione ***write()*** che scrive una quantità di byte trattandoli come un array, nello stesso modo già visto con la funzione ***read()***. Anche in questo caso, la scrittura procede a partire dalla posizione corrente riferita al file.

```
…
    char buf[100];
    int fdn;
    ssize_t dim;
    …
    fdn = open (…);
    …
    dim = write (fdn, buf, 100);
    …
```

L'esempio, come nel caso di ***read()***, mostra la scrittura di 100 byte, prelevati da un array. Il valore restituito dalla funzione è la quantità di elementi che sono stati scritti con successo. Se si verifica un errore la funzione restituisce il valore −1, mentre lo zero è un valore valido.

Anche in scrittura è importante l'indicatore della posizione interna del file. Di solito, quando si crea un file o lo si estende, l'indicatore si trova sempre alla fine. L'esempio seguente mostra lo scheletro di un programma che crea un file, copiando il contenuto di un altro (non viene utilizzato alcun tipo di controllo degli errori).

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
…
int
main (void)
{
    char          buf[1024];
    int           fdn_in;
    int           fdn_out;
    ssize_t       dim;
    …
    fdn_in = open (file, O_RDONLY);
```

```
    …
    fdn_out = open (file, O_WRONLY);
    …
    while (1)                       // Ciclo senza fine.
      {
        dim = read (fdn, buf, 1024);
        if (dim == 0)
          {
            break;                  // Termina il ciclo.
          }
        …
        write (fdn_out, buf, dim);
        …
      }
    …
    close (fdn_in);
    close (fdn_out);
    return 0;
}
```

Seguono i modelli sintattici di ***read()*** e ***write()***, espressi in forma di prototipi di funzione:

```
ssize_t read (int fdn, void *buf, size_t n);
```

```
ssize_t write (int fdn, const void *buf, size_t n);
```

Il tipo di dati '**ssize_t**' rappresenta l'equivalente di '**size_t**', ma con segno, allo scopo di poter rappresentare il valore −1, che indica un esito errato; il tipo '**void**' per l'array in cui vanno scritti o da cui vanno letti i dati, permette l'utilizzo di qualunque tipo per i suoi

elementi, anche se le operazioni di lettura e scrittura operano solo al livello di byte.

68.5.3 Spostamento dell'indicatore interno al file

«

Lo spostamento diretto dell'indicatore interno della posizione di un file aperto è un'operazione necessaria quando il file è stato aperto simultaneamente in lettura e in scrittura, e da un tipo di operazione si vuole passare all'altro. Per questo si utilizza la funzione ***lseek()***, con la quale è possibile leggere e modificare tale posizione attuale. La posizione e gli spostamenti sono espressi in byte; la variabile usata per rappresentare questi spostamenti è di tipo '**`off_t`**' (*offset*).

La funzione ***lseek()*** esegue lo spostamento a partire dall'inizio del file, oppure dalla posizione attuale, oppure dalla posizione finale. Per questo utilizza un parametro che può avere tre valori identificati rispettivamente da tre macro-variabili, definite all'interno del file '`stdio.h`': ***SEEK_SET***, ***SEEK_CUR*** e ***SEEK_END***. l'esempio seguente mostra lo spostamento del puntatore, riferito al descrittore di file ***fdn***, in avanti di 10 byte, a partire dalla posizione attuale.

```
…
i = lseek (fdn, 10, SEEK_CUR);
…
```

La funzione ***lseek()*** restituisce la posizione raggiunta all'interno del file, partendo dall'inizio dello stesso, se lo spostamento avviene con successo, altrimenti produce il valore -1.

L'esempio seguente mostra lo scheletro di un programma, senza controlli sugli errori, che, dopo aver aperto un file in lettura e scrittura, lo legge a blocchi di dimensioni uguali, modifica questi blocchi

e li riscrive nel file. Il file dell'esempio dovrebbe essere disponibile presso *allegati/c/esempio-posix-read-write.c* .

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

static const size_t max = 100;      // Dimensione massima del
                                    // record logico.
int
main (void)
{
  const char *file = "/tmp/test"; // File da elaborare.
  char        buf[max];       // Allocazione del record.
  int         fdn;            // Descrittore del file.
  off_t       pos_1;          // Posizione interna al file.
  off_t       pos_2;          // Posizione interna al file.
  ssize_t     dim;            // Dimensione del record letto.

  fdn = open (file, O_RDWR);

  while (1)                   // Ciclo senza fine.
    {
      //
      // Salva la posizione del puntatore interno al file
      // prima di eseguire la lettura.
      //
      pos_1 = lseek (fdn, 0, SEEK_CUR);

      dim = read (fdn, buf, max);

      if (dim == 0)
        {
          break;              // Termina il ciclo.
        }
```

```
        //
        // Salva la posizione del puntatore interno al file
        // dopo la lettura.
        //
        pos_2 = lseek (fdn, 0, SEEK_CUR);
        //
        // Sposta il puntatore alla posizione precedente alla
        // lettura.
        //
        lseek (fdn, pos_1, SEEK_SET);
        //
        // Esegue qualche modifica nei dati, per esempio
        // mette un punto esclamativo all'inizio.
        //
        buf[0] = '!';
        //
        // Riscrive il record modificato.
        //
        write (fdn, buf, dim);
        //
        // Riporta il puntatore interno al file alla posizione
        // corretta per eseguire la lettura successiva
        //
        lseek (fdn, pos_2, SEEK_SET);
    }

    close (fdn);
    return 0;
}
```

Segue il modello sintattico per l'uso della funzione ***lseek()***, espresso attraverso il suo prototipo:

```
off_t lseek (int fdn, off_t spostamento,
             int punto_di_partenza);
```

Il valore dello spostamento, costituito dal secondo parametro, rappresenta una quantità di byte che può essere anche negativa, indicando in tal caso un arretramento dal punto di partenza specificato dal terzo parametro. Il valore restituito da ***lseek()*** è la nuova posizione all'interno del file, espressa a partire dall'inizio dello stesso (pertanto deve trattarsi di un valore maggiore o uguale a zero); se invece si presenta un errore, si ottiene −1 (precisamente si definisce come '**`(off_t) -1`**').

68.5.4 Controllo degli errori

Le funzioni descritte, quando si verifica un errore, annotano il numero dell'errore nella variabile globale ***errno*** (il nome ***errno*** dovrebbe essere precisamente un'espressione che si traduce nell'accesso a un'area di memoria condiviso dal programma, distinto in base al thread). Il significato del valore attribuito alla variabile ***errno*** è descritto da macro-variabili definite nel file '`errno.h`', il quale fa già parte dello standard C, ma viene esteso da POSIX.

La lettura della variabile ***errno*** porta alla conoscenza dell'ultimo errore che si è presentato e non è previsto il suo azzeramento automatico; pertanto, occorre accertarsi del verificarsi di un problema, prima di interrogare la variabile, oppure la si deve azzerare prima di chiamare una funzione di cui si vuole verificare l'esito.

Per interpretare l'errore annotato nella variabile ***errno*** e visualizzare direttamente un messaggio attraverso lo standard error, si può usare

la funzione ***perror()***, già descritta nei capitoli sul C:

```
void perror (const char *s);
```

68.6 Il file system Unix e la sua gestione tipica

«

Per comprendere il senso dell'organizzazione della libreria C e di quella POSIX, per quanto riguarda la gestione dei file, è necessario conoscere l'impostazione originale della gestione di un file system in un sistema Unix. A tale riguardo ci sono due livelli: quello del file system, così come viene strutturato nell'unità di memorizzazione e la gestione dei file aperti, a livelli diversi, partendo dall'inode, fino al flusso di file del C, passando per il concetto di descrittore del file.

68.6.1 Il blocco

«

In un file system Unix tradizionale, lo spazio di un'unità di memorizzazione è suddiviso in blocchi di byte, di dimensione pari a un multiplo del settore fisico, ma si tratta comunque di un valore che si ottiene come potenza di 2. Considerato che le unità di memorizzazione comuni hanno settori fisici da 512 byte, il blocco di un tale file system può essere da 1024, 2048, 4096 byte,... La dimensione effettiva di tale blocco dipende però dalle caratteristiche specifiche di quel tipo di file system, tenendo conto che spesso è possibile scegliere la sua dimensione in fase di inizializzazione.

Nel caso del file system Minix, si distingue tra blocchi e zone. La zona è un concetto specifico dei sistemi Minix e, in realtà, la zona di Minix è l'entità del file system che più si avvicina al blocco dei sistemi Unix tradizionali.

68.6.2 Il super blocco

Le unità di memorizzazione possono essere organizzate in due modi: con partizioni o senza. Una partizione è a sua volta come un'unità singola, non divisa in partizioni.

L'inizio di un'unità di memorizzazione viene riservato generalmente per il codice di avvio del sistema operativo, tenendo conto che questo vale sia per le unità suddivise in partizioni, sia per quelle non suddivise, sia per le partizioni stesse. Pertanto, nessun file system sovrascrive il primo settore di un'unità, anche se può considerarlo parte della propria gestione.

Dopo lo spazio che viene lasciato per il codice di avvio del sistema operativo (di uno o più settori), si colloca generalmente quello che è noto come ***super blocco***, il quale si può considerare come una tabella riassuntiva delle caratteristiche generali del file system e della sua situazione.

Le informazioni contenute nel super blocco devono consentire di sapere: qual è la dimensione del blocco (ammesso che questa non sia fissa); qual è la dimensione dell'unità in blocchi; dove sono le tabelle che rappresentano gli inode; quanti sono gli inode, quali sono quelli liberi e quali invece sono impegnati; quali sono i blocchi che possono essere utilizzati per i dati (file, directory e altre tabelle per i riferimenti indiretti) e quali invece sono impegnati.

Per conoscere quali sono gli inode e le zone libere o impegnate, si possono usare sistemi diversi. In generale è probabile che si usino mappe di bit (come nel caso di Minix), oppure delle liste. Ma in generale, ciò che consente di sapere e di annotare gli inode e le zone impegnate o libere, fa parte concettualmente del super blocco, anche

se materialmente può trattarsi di strutture di dati separate.

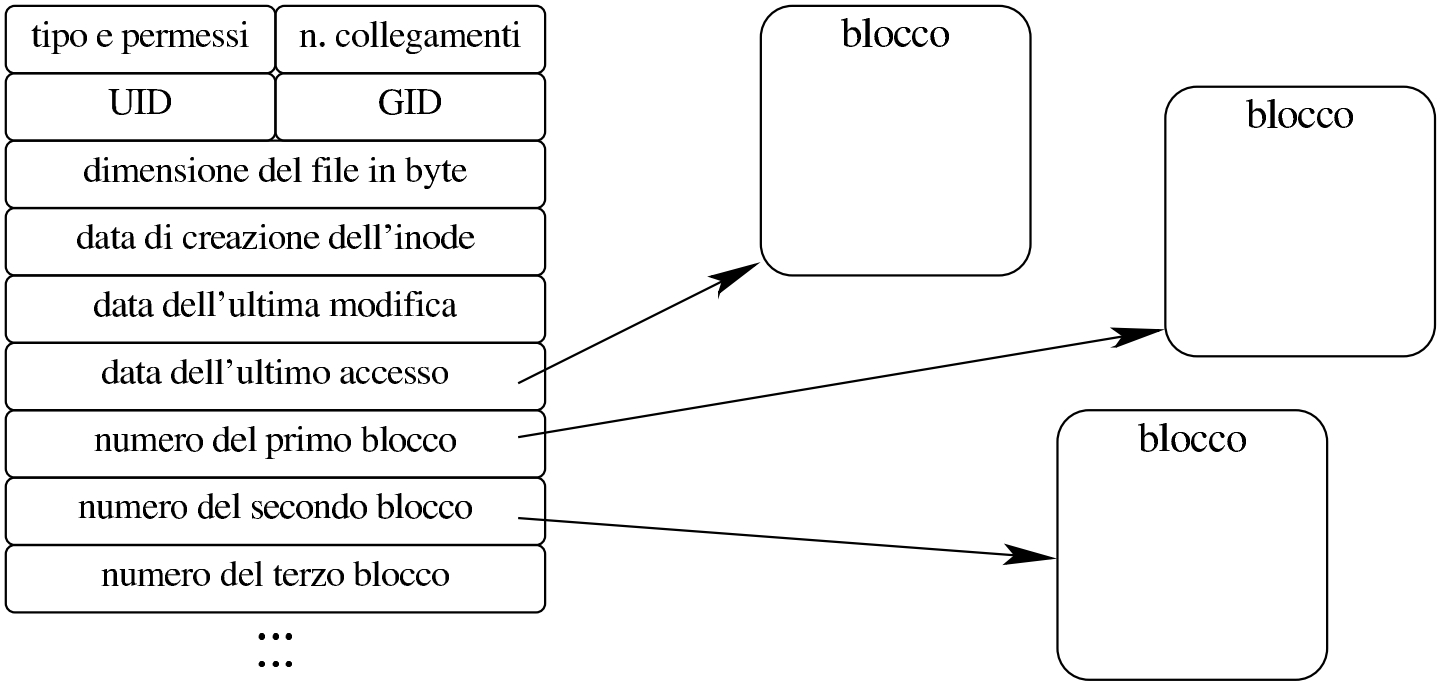
68.6.3 Gli inode

«

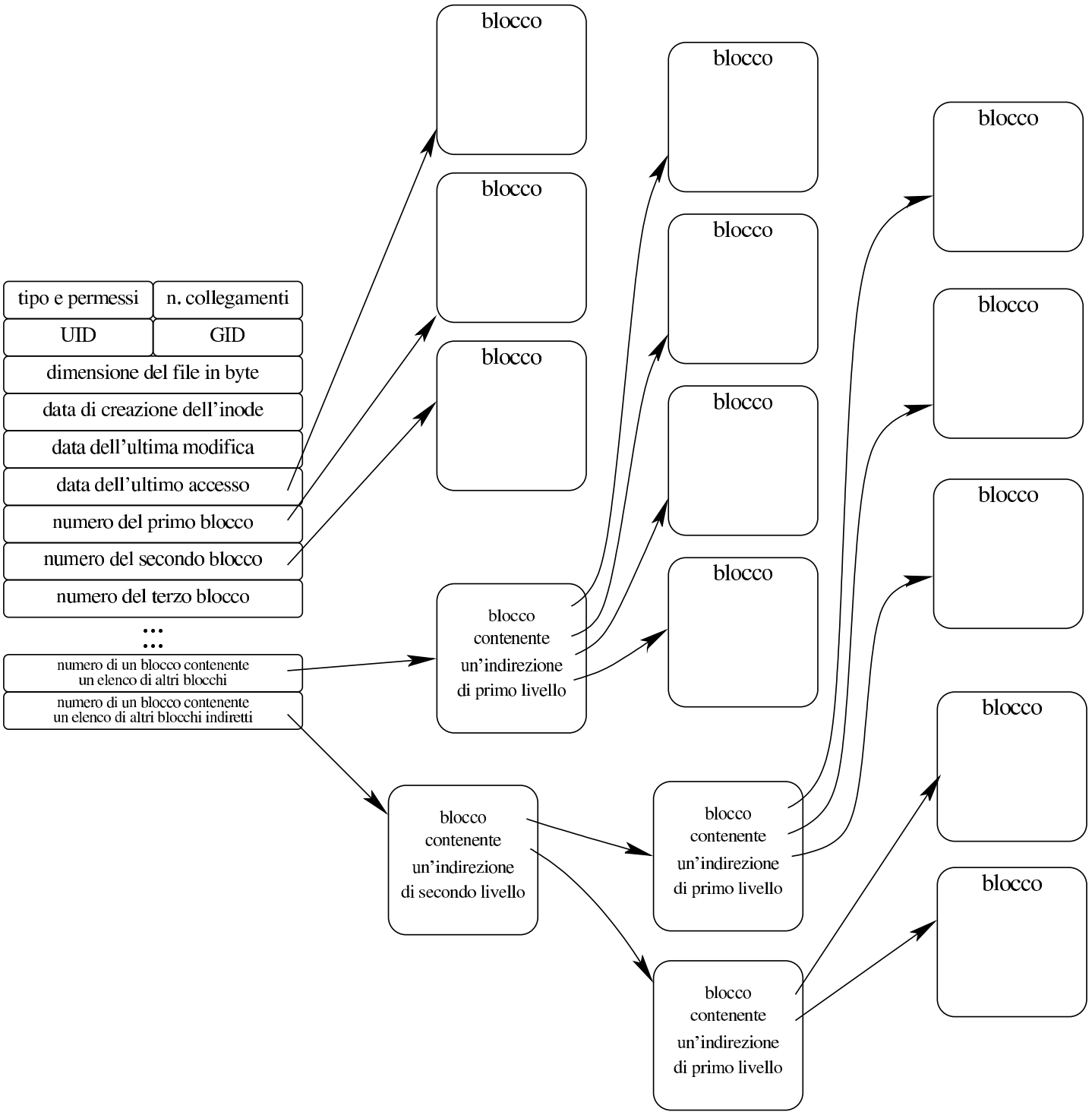
Dopo il super blocco, includendo in questo anche ciò che consente di sapere quali inode e quali blocchi sono impegnati o liberi, si collocano generalmente delle tabelle che rappresentano, ognuna, un inode. Di solito, un inode occupa un sottomultiplo dello spazio di un blocco, in modo da facilitare i calcoli per individuarne la collocazione.

Un inode contiene almeno queste informazioni: il tipo di file e i permessi di accesso, in un valore a 16 bit; il numero UID del proprietario del file; il numero GID del gruppo proprietario; la dimensione del file (o della directory) espressa in byte; le date di accesso, modifica del file e creazione dell'inode; la quantità di riferimenti provenienti dalle directory; una serie di numeri di blocchi occupati dal file o dalla directory.

Il contenuto del file rappresentato dall'inode si articola in blocchi, i quali non sono necessariamente contigui e nemmeno ordinati: è l'insieme dei riferimenti contenuti nell'inode che determina la posizione e l'ordine in cui questi vanno considerati. Inoltre, dato che non è possibile allocare nel file system uno spazio più piccolo di un blocco, è indispensabile l'informazione sulla dimensione del file per sapere quando questo termina nel suo ultimo blocco utilizzato.



Dal momento che deve essere possibile rappresentare file di grandi dimensioni, i primi riferimenti ai blocchi utilizzati sono diretti, come si vede nel disegno appena mostrato, mentre si prevedono generalmente dei riferimenti indiretti, a blocchi che contengono a loro volta i riferimenti di altri blocchi. In tal caso si parla di «indirezione» di primo, secondo ed eventualmente anche di terzo livello.



In questa struttura di collegamenti ai blocchi, va osservato che un file potrebbe non avere allocato tutti blocchi che risulterebbero dalla dimensione riportata. Per esempio, utilizzando blocchi da 1024 byte, un file che risulta essere grande 10240 byte, non è detto che occupi effettivamente 10 blocchi come sembrerebbe. Infatti, la scrittura nel

file potrebbe essere avvenuta specificando una piccola parte verso la fine e in altre posizioni. In pratica, almeno in linea teorica, questo tipo di organizzazione a inode, consente di scrivere il file dove si vuole, sapendo che lo spazio intermedio, se non viene allocato, risulta contenere dei dati con bit a zero.

Nella tabella che rappresenta l'inode, i blocchi non allocati risultano indicati con il numero zero, pertanto, il vero blocco zero non può essere accessibile attraverso gli inode (ma d'altra parte è normale che il blocco zero sia impegnato dal codice di avvio, dal super blocco ed eventualmente dalle tabelle degli inode stessi).

È importante osservare che in un file system Unix non esiste mai l'inode con il numero zero, perché questo valore viene utilizzato per fare riferimento a un errore o comunque a situazioni speciali. Di solito, l'inode numero uno corrisponde alla directory principale, dell'unità presa in considerazione (ovvero di una sua partizione).

68.6.4 La directory

La directory è un file come gli altri, riconoscibile perché, nell'inode, il campo che definisce il tipo e i permessi, riporta l'indicazione relativa. Nei file system tradizionali, il file che rappresenta la directory è formato normalmente da *record* di lunghezza uniforme, in cui si distingue un campo contenente il numero di un inode e un altro contenente il nome di un file.

n. inode	nome
	.
	..
	cat
	cp
	dd

In pratica, si associa il tale inode a un certo nome. Va ricordato che l'inode zero non esiste e che, di norma, l'inode uno è quello della directory radice dell'unità di memorizzazione (o della partizione relativa).

Nel file di intestazione '`limits.h`', la macro-variabile ***NAME_MAX*** rappresenta la quantità minima di caratteri che possono essere usati per i nomi dei file (nelle directory). Questa quantità non include il carattere nullo di terminazione delle stringhe; pertanto, se corrispondesse al valore 14, vorrebbe dire che i nomi possono avere effettivamente 14 caratteri. Ciò avviene a differenza della macro-variabile ***PATH_MAX***, per i percorsi, la quale deve invece includere anche il carattere nullo di terminazione.

La directory si costruisce come descritto, ma rimane il fatto che le prime due voci debbano essere «.» e «..»: la prima corrispondente al riferimento dell'inode della directory stessa; la seconda corrispondente al riferimento dell'inode della directory genitrice (ovvero quella precedente in senso gerarchico), con la variante che la directory radice può solo puntare a se stessa, in ogni caso.

68.6.5 Tabelle del sistema operativo

Il file system dei sistemi Unix, oltre che avere una certa forma nella fisicità dell'unità di memorizzazione, ha anche una rappresentazione astratta tradizionale nel sistema operativo, nel modo in cui si prendono in considerazione i file aperti e ciò da cui questi dipendono.

Nella semplificazione dei sistemi tradizionali, si utilizzano delle tabelle per: i super blocchi delle unità innestate; gli inode in corso di utilizzazione; i file aperti; i descrittori dei file aperti; i flussi di file abbinati ai descrittori.

68.6.5.1 Tabella dei super blocchi

Per accedere a un file, è necessario poter raggiungere il file system di un certo dispositivo, il quale deve essere stato innestato, ovvero reso disponibile nel file system generale del sistema operativo. Per raccogliere la situazione delle unità innestate serve una tabella, la quale può essere vista come quella dei dispositivi o dei super blocchi.

Le voci della tabella dei super blocchi riproducono i super blocchi delle unità innestate, incluse le mappe o le tabelle di utilizzo degli inode e dei blocchi, oltre ad altre informazioni accessorie. Quando di crea o si elimina un file, quando lo si estende e lo si riduce, la voce relativa di tale tabella dei super blocchi va aggiornata anche nell'unità di memorizzazione, per quanto riguarda la situazione di utilizzo degli inode e dei blocchi di dati.

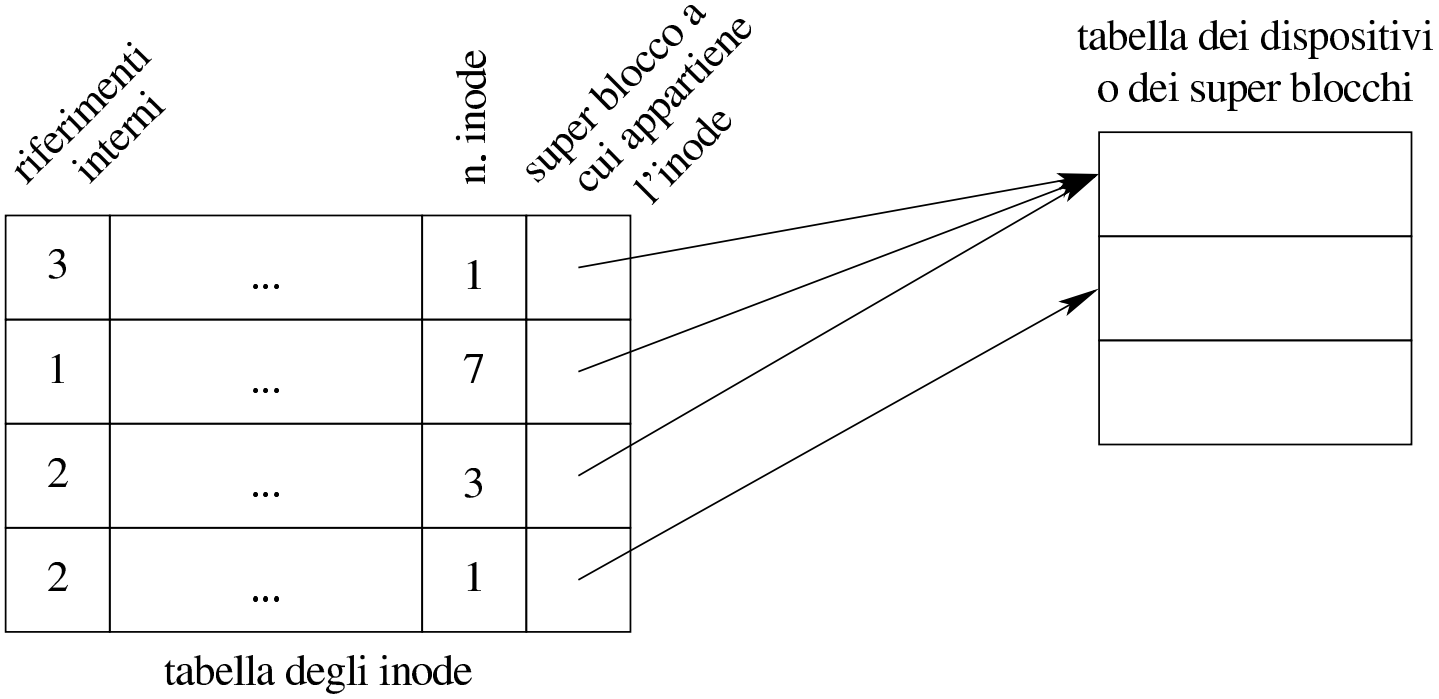
Va osservato che il sistema operativo deve innestare almeno una unità, contenente il file system principale, pertanto, almeno un super blocco deve essere sempre presente nella tabella.

68.6.5.2 Tabella degli inode

«

Quando si accede a un file per la prima volta, le informazioni relative al suo inode vengono caricate in una voce della tabella degli inode. Il contenuto minimo di questa voce è costituito di norma da tutti i dati dell'inode contenuti nel file system, incluso il numero di questo, aggiungendo il riferimento alla voce che rappresenta il super blocco da cui proviene e la quantità di riferimenti interni (i riferimenti interni non vanno confusi con i collegamenti nel file system, provenienti dalle directory).

Quando si apre più volte lo stesso file, o comunque ciò che fa capo allo stesso inode, nella tabella di inode si ha sempre solo una voce, dove il contatore dei riferimenti interni serve a sapere quante volte risulta aperto. Quando poi tale contatore arriva a zero, perché i file vengono chiusi mano a mano, la voce della tabella è libera e può essere riutilizzata per un altro inode, oppure può essere semplicemente ripresa così come si trova, se il file viene riaperto (incrementando nuovamente il contatore).



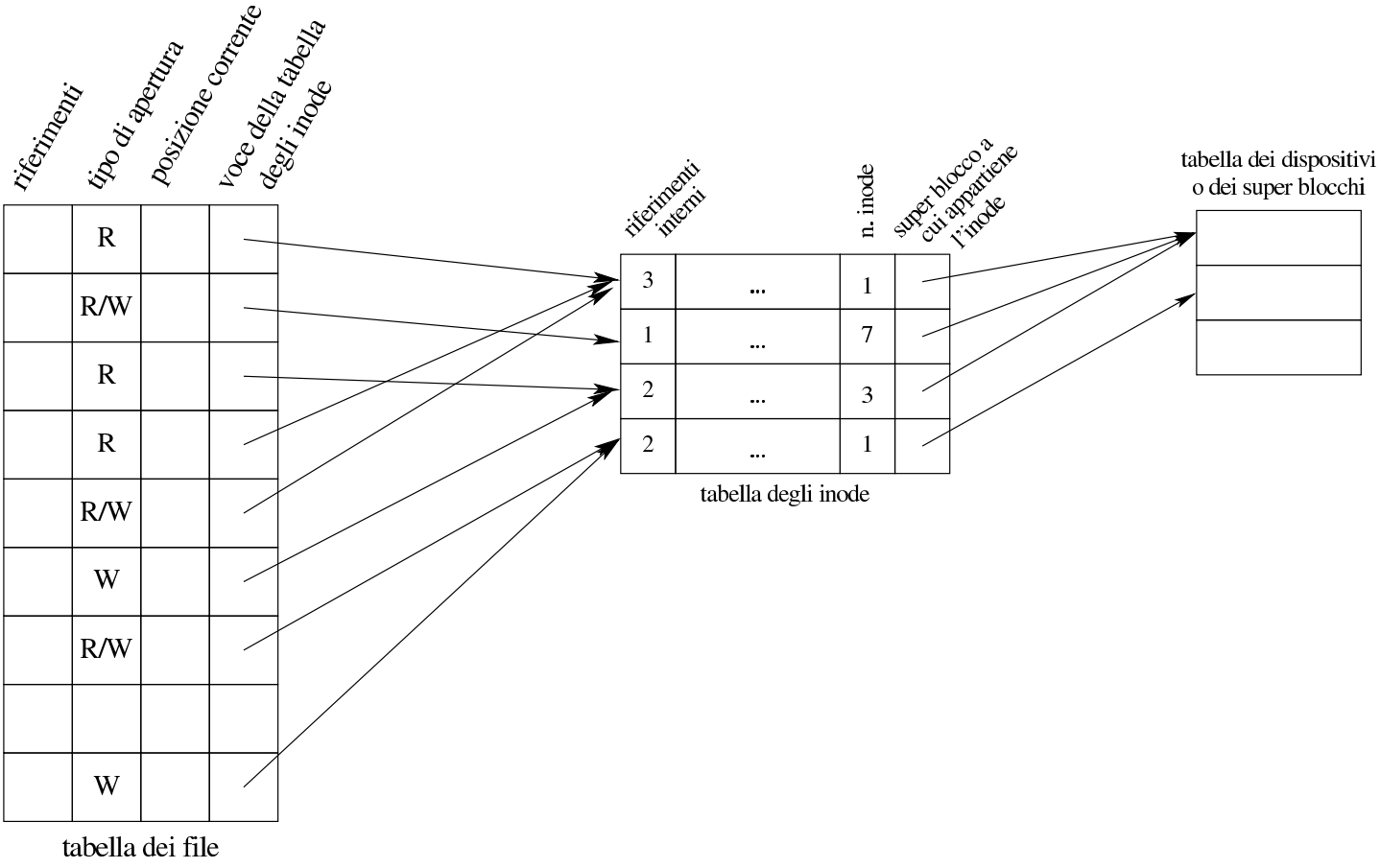
Va però osservato che, durante il funzionamento del sistema operativo, potrebbe farsi riferimento a inode astratti, privi del collegamento a un dispositivo o super blocco. Questo caso riguarda in particolare i condotti «privi di nome», ovvero quelli che non derivano dall'apertura di un file speciale di tipo FIFO.

68.6.5.3 Tabella dei file

L'apertura di un file, oltre che coinvolgere la tabella degli inode, implica l'aggiunta di una voce nella tabella dei file di sistema (ovvero dei file aperti complessivamente nel sistema operativo). Le voci di questa tabella devono avere un riferimento all'inode, la modalità di apertura (lettura, scrittura o entrambe), la posizione corrente nel file per le letture o le scritture successive, un contatore di riferimenti interni.

L'apertura di un file implica sempre l'aggiunta di una nuova voce nella tabella dei file, ma ci sono delle situazioni in cui uno stesso

processo o più processi differenti possono condividere la stessa voce della tabella dei file di sistema. Come nel caso degli inode, quando il contatore dei riferimenti raggiunge lo zero, significa che la voce corrispondente è chiusa (o libera) e può essere riutilizzata per il prossimo file da aprire.

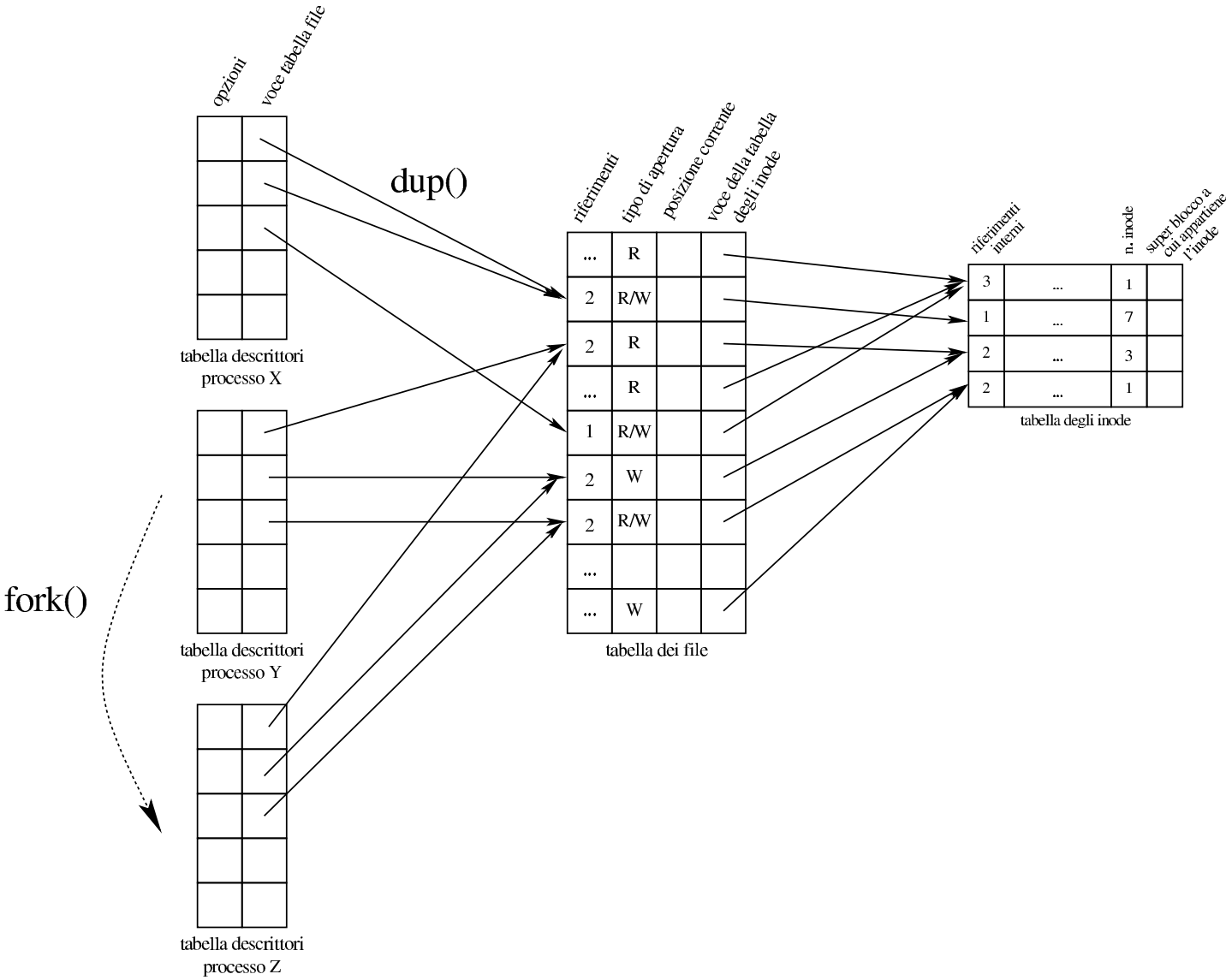


68.6.5.4 Tabella dei descrittori dei file

«

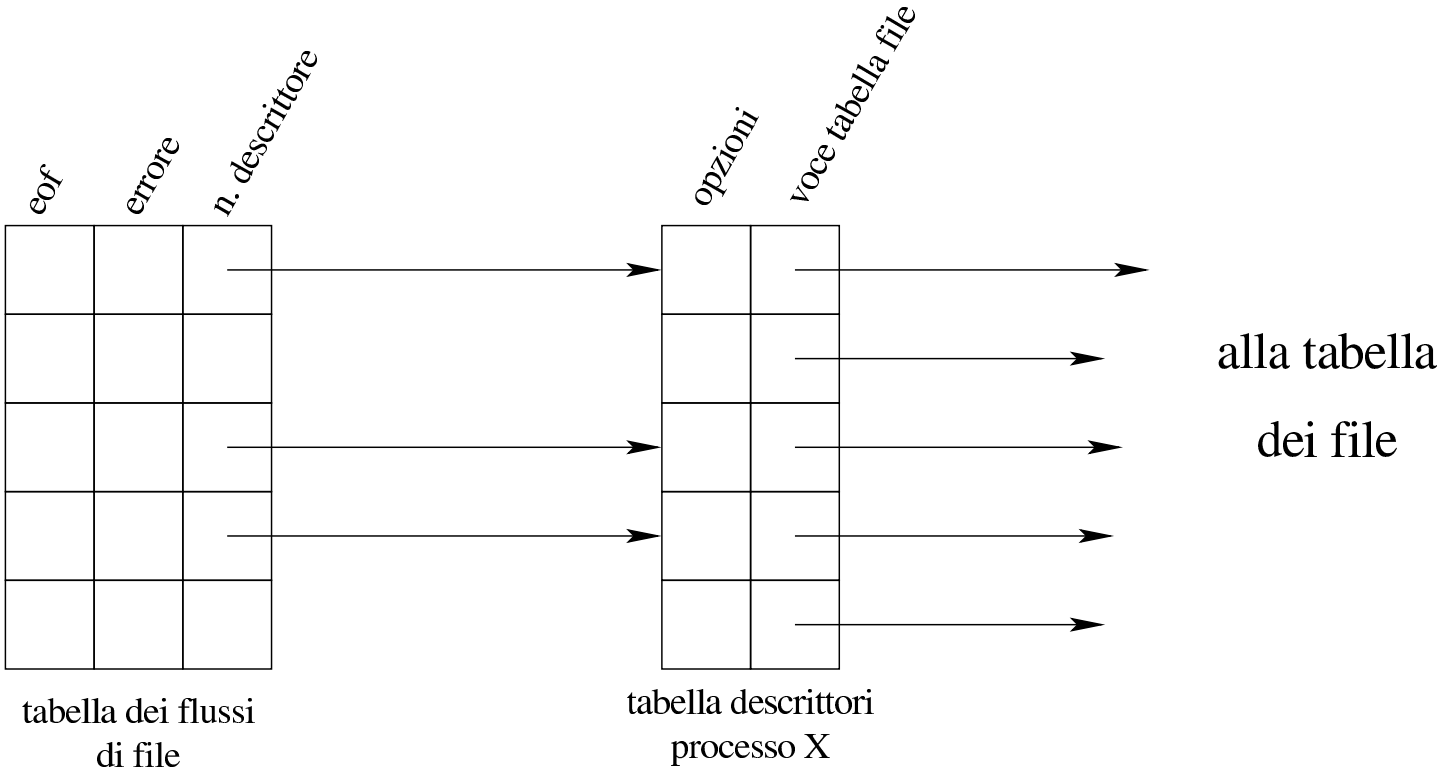
Ogni processo elaborativo ha una propria tabella dei descrittori dei file, nella quale, le voci rappresentano i file aperti dal processo stesso. Le voci della tabella includono il riferimento alla tabella dei file di sistema e le opzioni date in fase di apertura, riguardanti aspetti più precisi rispetto alla semplice distinzione di un accesso in lettura o in scrittura.

Quando un processo elaborativo si sdoppia, attraverso la chiamata di sistema che fa capo alla funzione ***fork()***, i descrittori dei file vengono duplicati e i riferimenti corrispondenti nella tabella dei file di sistema si incrementano. Nello stesso modo, se un processo elaborativo utilizza la funzione della libreria standard ***dup()***, ottiene la duplicazione di un descrittore, incrementando il contatore dei riferimenti nella tabella dei file.



68.6.5.5 Flussi di file

Dal punto di vista del processo elaborativo, la gestione dei file in forma di descrittori o di flussi, potrebbe sembrare indipendente, ma in pratica ciò non può essere. La gestione dei flussi di file implica la presenza di una tabella aggiuntiva (oltre a quella dei descrittori), contenente, per ogni voce, il riferimento al descrittore, un indicatore di errore e un altro indicatore di fine file.



Come suggerisce intuitivamente il disegno, in un sistema operativo POSIX, un flusso di file aperto ha un proprio descrittore di file corrispondente, anche se nell'ambito del programma può rimanere sconosciuto il numero del descrittore abbinato. Tuttavia, per converso, è possibile aprire un file attraverso un descrittore, senza che sia coinvolto necessariamente il flusso che gli corrisponderebbe. A tale proposito, lo standard prescrive la presenza di funzioni che consentono di ristabilire il collegamento esplicito tra flussi e descrittori.

68.7 Il file system Minix 1

Come esempio di come può essere strutturato effettivamente un file system, conforme alle richieste dello standard POSIX, viene proposta la spiegazione dettagliata del tipo usato dal sistema operativo Minix, nelle sue primissime edizioni.

Il kernel Linux consente di accedere a file system Minix 1, eventualmente con l'estensione dei nomi a 30 byte, mentre manca una gestione efficace del file system Minix 2 e manca del tutto la possibilità di accedere alla versione Minix 3.

68.7.1 Blocchi e zone

Il file system Minix 1 suddivide lo spazio disponibile in ***blocchi*** da 1024 byte; così, qualunque oggetto sia memorizzato occupa un multiplo di tale dimensione. Di solito, le unità di memorizzazione di massa sono organizzate in settori da 512 byte, pertanto tale organizzazione in blocchi si adatta perfettamente alle unità comuni.

Per l'indirizzamento dei dati, all'interno del file system, si utilizza il concetto di ***zona***, corrispondente a un multiplo del blocco, ottenuto però come potenza di due. Pertanto possono esserci zone della stessa dimensione dei blocchi, oppure doppie, quadruple,... In pratica, deve essere possibile rappresentare con un numero intero, il logaritmo in base due del rapporto tra la dimensione della zona e la dimensione del blocco.

$$\log_2 \frac{dim\_zona}{dim\_blocco}$$

Per esempio, una zona da 8192 byte, porta a un rapporto tra zona e blocco di 8 e $\boldsymbol{log}_2 8$ è pari a 3 (in quanto $2^3 = 8$).

Il valore del logaritmo in base due, del rapporto tra zona e blocco, fa parte delle informazioni contenute nel file system Minix 1, perché serve a ottenere la dimensione della zona, attraverso lo scorrimento a sinistra del valore 1024. Per esempio così:

```
y = 1024 << 3;
```

In tal caso, la variabile ***y*** va a contenere il valore 8192. Ma in alternativa, basta calcolare i multipli di blocco, ottenendo così il valore 8:

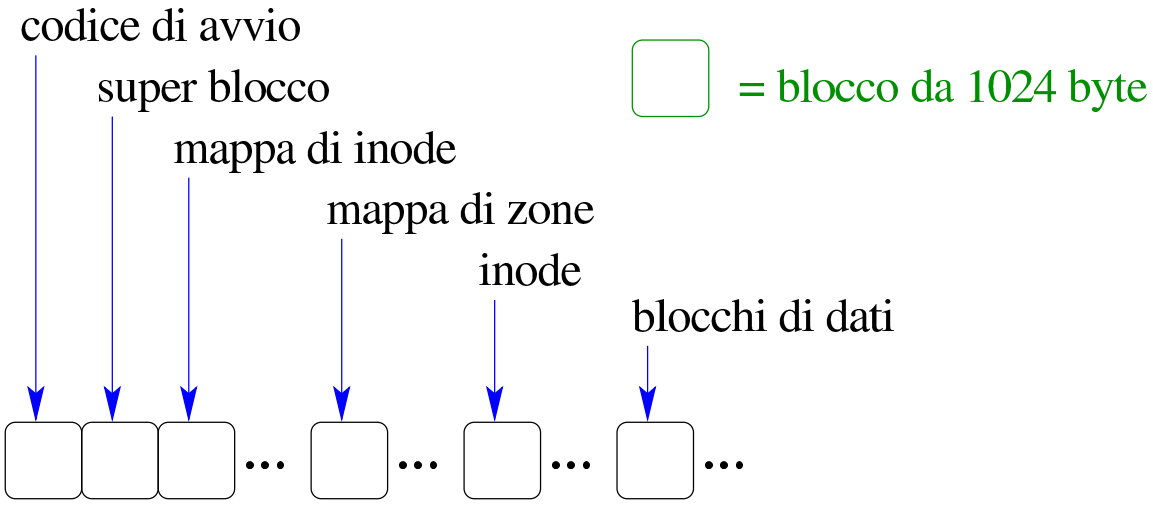
```
y = 1 << 3;
```

Nella tradizione Unix non esiste la «zona», la quale riguarda specificatamente il file system Minix. La zona di Minix rappresenta in pratica il concetto di «blocco» dei sistemi Unix.

68.7.2 Struttura generale

«

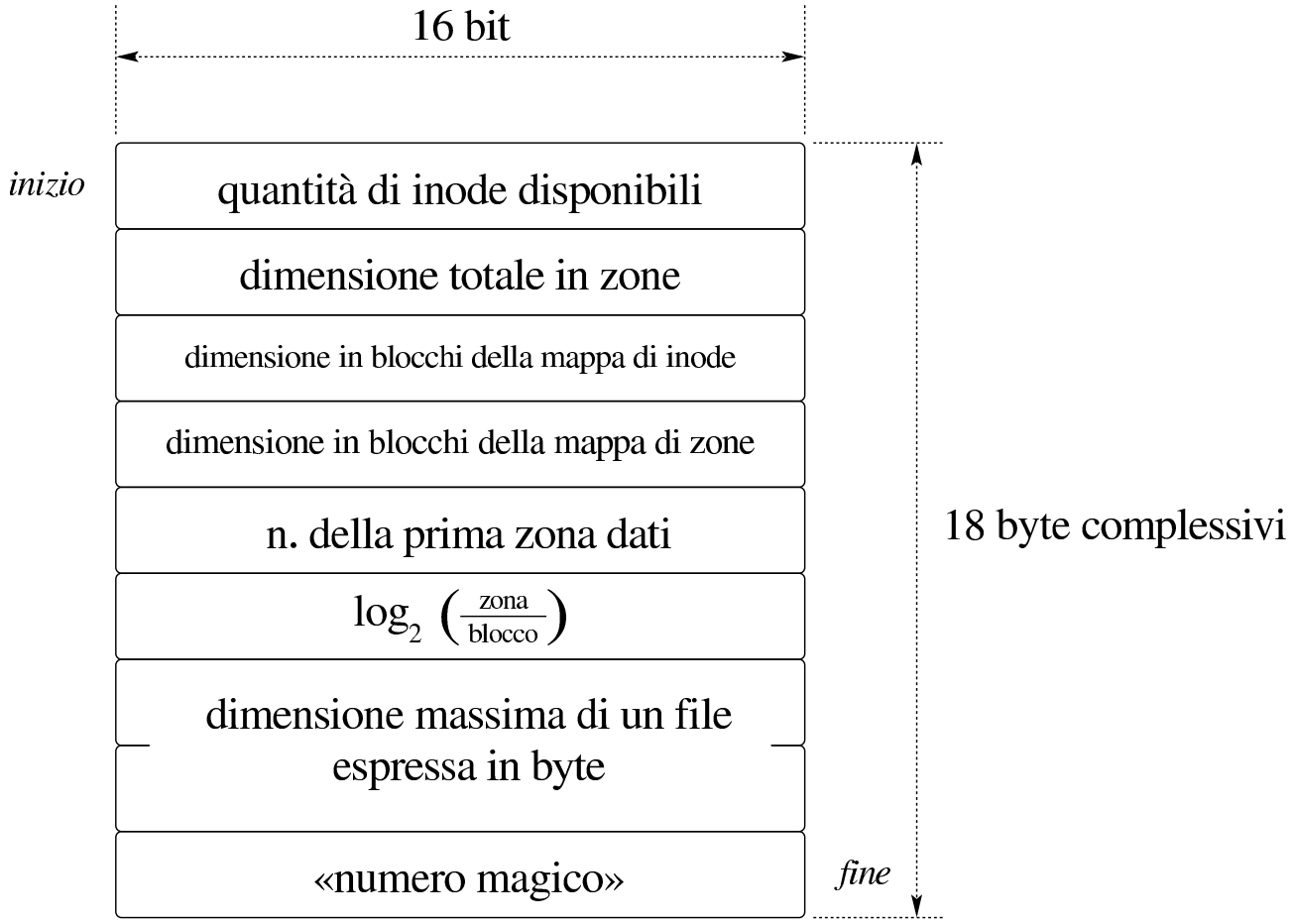
La struttura generale del file system Minix 1 è schematizzata dalla figura successiva. Il primo blocco (da 1024 byte) è riservato al codice di avvio, anche se di solito questo occupa soltanto un settore da 512 byte. Il secondo blocco contiene il «super blocco», ovvero l'intestazione del file system, con le informazioni generali sullo stesso. Il terzo blocco, ed eventuali altri blocchi successivi, sono utilizzati per una mappa degli inode, la quale ha lo scopo di annotare quali sono utilizzati e quali sono liberi. A partire dal blocco successivo inizia la mappa delle zone utilizzate (zone, intese come multipli dei blocchi, come spiegato nella sezione precedente). Dopo la mappa delle zone appaiono i blocchi contenenti gli inode (tanti quanti sono previsti nella mappa di inode). Successivamente appaiono i blocchi usati dalle zone di dati che utilizzano lo spazio rimanente.



Va osservato che se le zone hanno una dimensione maggiore dei blocchi, il primo blocco utile per la memorizzazione dei dati (dopo gli inode) deve iniziare all'inizio di una zona, contando le zone a partire dal primo blocco (quello riservato dal codice di avvio). Pertanto, potrebbero rimanere anche blocchi non utilizzabili, dopo quelli delle mappe e prima di quelli dei dati.

68.7.3 Super blocco

Il super blocco raccoglie le informazioni più importanti del file system e dalla sua integrità dipende l'accessibilità di tutto il resto del contenuto presente. Anche se gli viene riservato un blocco intero, in pratica, il super blocco di Minix 1 occupa molto meno spazio.



Il secondo campo del super blocco, come si vede dalla figura, rappresenta la dimensione dell'unità di memorizzazione (o della partizione considerata), espressa in zone. Per esempio, se si utilizzano zone uguali ai blocchi, un dischetto da 1440 Kibyte è composto esattamente da 1440 zone.

Il quinto campo indica la prima zona dati, contando a partire da zero. La prima zona dati è la prima zona che possa essere usata, dopo gli inode. Nella mappa delle zone utilizzate, il bit che rappresenta la zona dati numero uno, si riferisce a questa prima zona dati (nella mappa delle zone, la zona dati zero risulta sempre utilizzata ma in realtà non esiste).

Il sesto campo indica il logaritmo in base due, del rapporto tra di-

mensione della zona e del blocco. Per esempio, un valore pari a zero indica che la zona è uguale al blocco; uno indica che la zona è costituita da due blocchi; tre indica che la zona è composta da quattro blocchi e così di seguito.

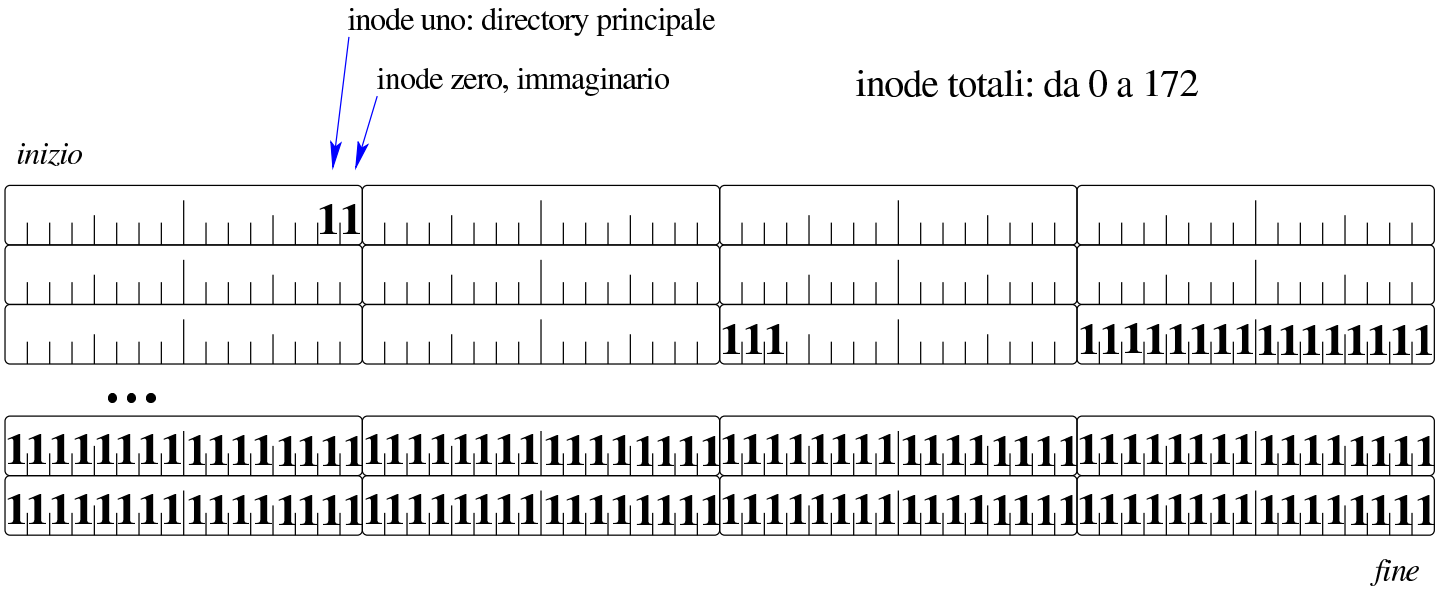
Il «numero magico» è il codice di riconoscimento, usato per verificare che si tratti effettivamente di un file system Minix 1. Tale numero deve essere $137F_{16}$.

68.7.4 Mappa di inode

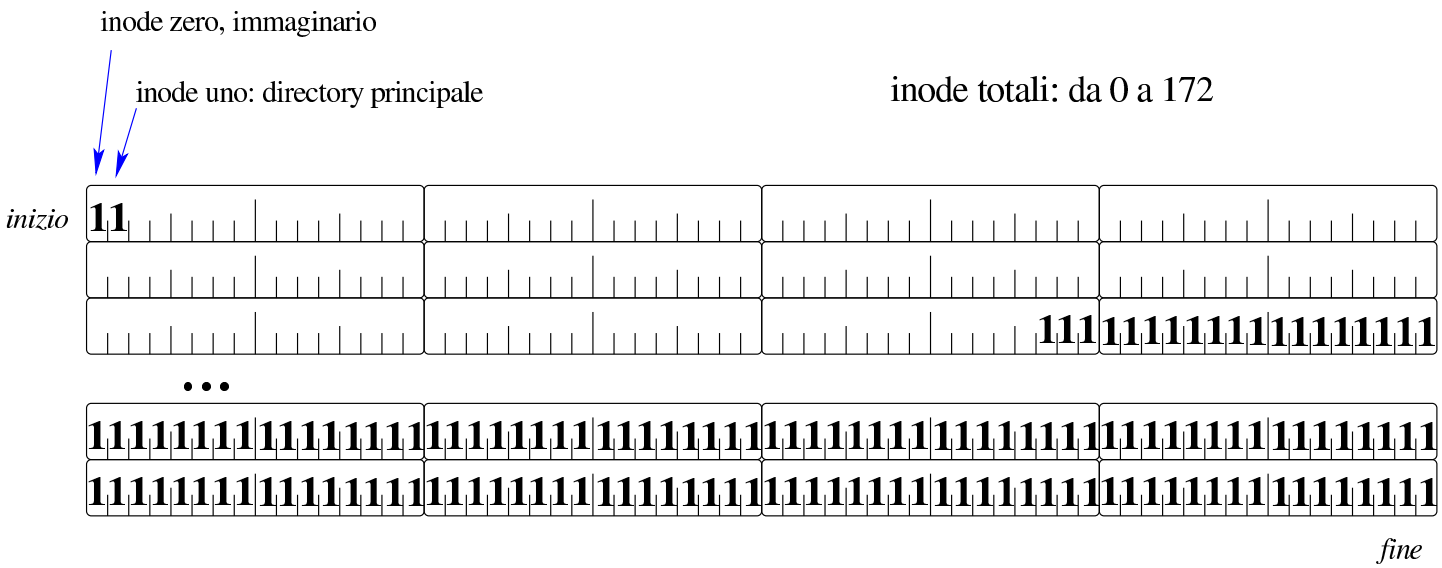
La mappa di inode è costituita da un insieme di bit, ognuno dei quali rappresenta lo stato di utilizzazione di un certo inode: 1 indica un inode utilizzato; 0 indica un inode libero. In questa mappa, il primo bit, riferito all'inode zero, è sempre attivo. Ma l'inode zero, in pratica, non viene rappresentato e il primo vero inode, ovvero quello riferito alla directory radice, ha sempre il numero uno.

In base al fatto che l'inode zero, in pratica, non esiste, anche se risulta sempre utilizzato, va considerato che il valore presente nel primo campo del super blocco indica la quantità reale di inode, ovvero l'indice massimo (partendo da zero) che si può utilizzare nella loro scansione.

La mappa di bit va però scandita, suddividendola a blocchi da 16 bit. L'esempio seguente rappresenta una mappa per 172 inode, dove si vede il primo (zero) impegnato e il secondo che già è predisposto per la directory principale:



Nel disegno sono rappresentati solo i bit a uno, lasciando gli altri come spazi vuoti. Va osservato che l'ordine in cui si dispongono i bit non è quello che ci si aspetterebbe: il primo, quello dell'inode zero, appare a destra del suo insieme di 16 bit (si tratta quindi del bit meno significativo), mentre lo si attenderebbe a sinistra secondo il senso di lettura latino. Considerato che i bit inutilizzati vanno posti a uno, come se esistessero altrettanti inode impegnati, l'ultimo insieme di 16 bit va interpretato con attenzione. Per avere una visione più umana della mappa, occorrerebbe invertire la sequenza di bit in ogni gruppetto:



Al problema dell'inversione della sequenza di bit, si aggiunge il fatto che il file system è nato per un'architettura *little endian*, ovvero a byte invertiti, ma la questione viene trattata alla fine del capitolo.

Per sapere dove si trova un certo inode ***n***, occorre considerare che questi si collocano dopo i blocchi della mappa di zone, che il primo vero inode è quello con indice uno, ovvero il secondo, in base alla numerazione della mappa. Come viene descritto successivamente, ogni inode occupa 32 byte, pertanto, in ogni blocco ci stanno esattamente 32 inode.

68.7.5 Mappa di zone

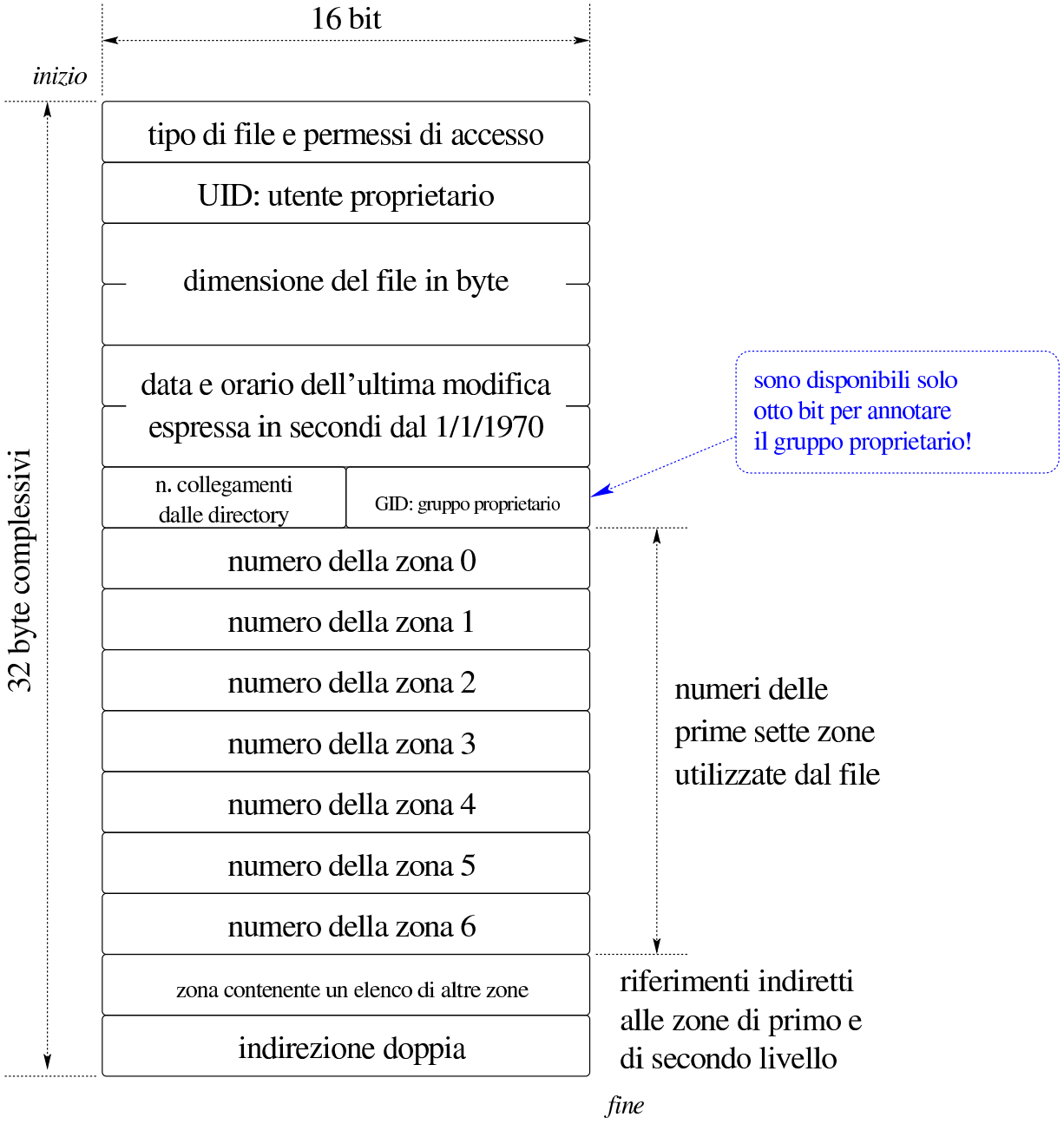
La mappa delle zone dei dati, funziona come quella di inode, dove i bit a uno indicano una zona utilizzata e il primo bit, riferito alla zona zero, è sempre a uno, ma in realtà la prima vera zona dati è quella a cui corrisponde l'indice uno. Come per la mappa di inode, anche in questo caso valgono le stesse considerazioni relative al fatto che la scansione deve essere fatta a gruppi di 16 bit e che il conteggio inizia dalla parte numericamente meno significativa di tali gruppi.

È bene precisare che la mappa si riferisce alle zone dei dati, pertanto riguarda quelle zone che iniziano dopo tutte le informazioni già descritte, compresa la stessa mappa e la tabella di inode successiva. Per fare un esempio, se nel super blocco è scritto che la prima zona dati è quella con il numero 19, significa che il bit con indice uno della mappa (il secondo) individua la zona 19 e le zone precedenti non possono essere utilizzate per i dati.

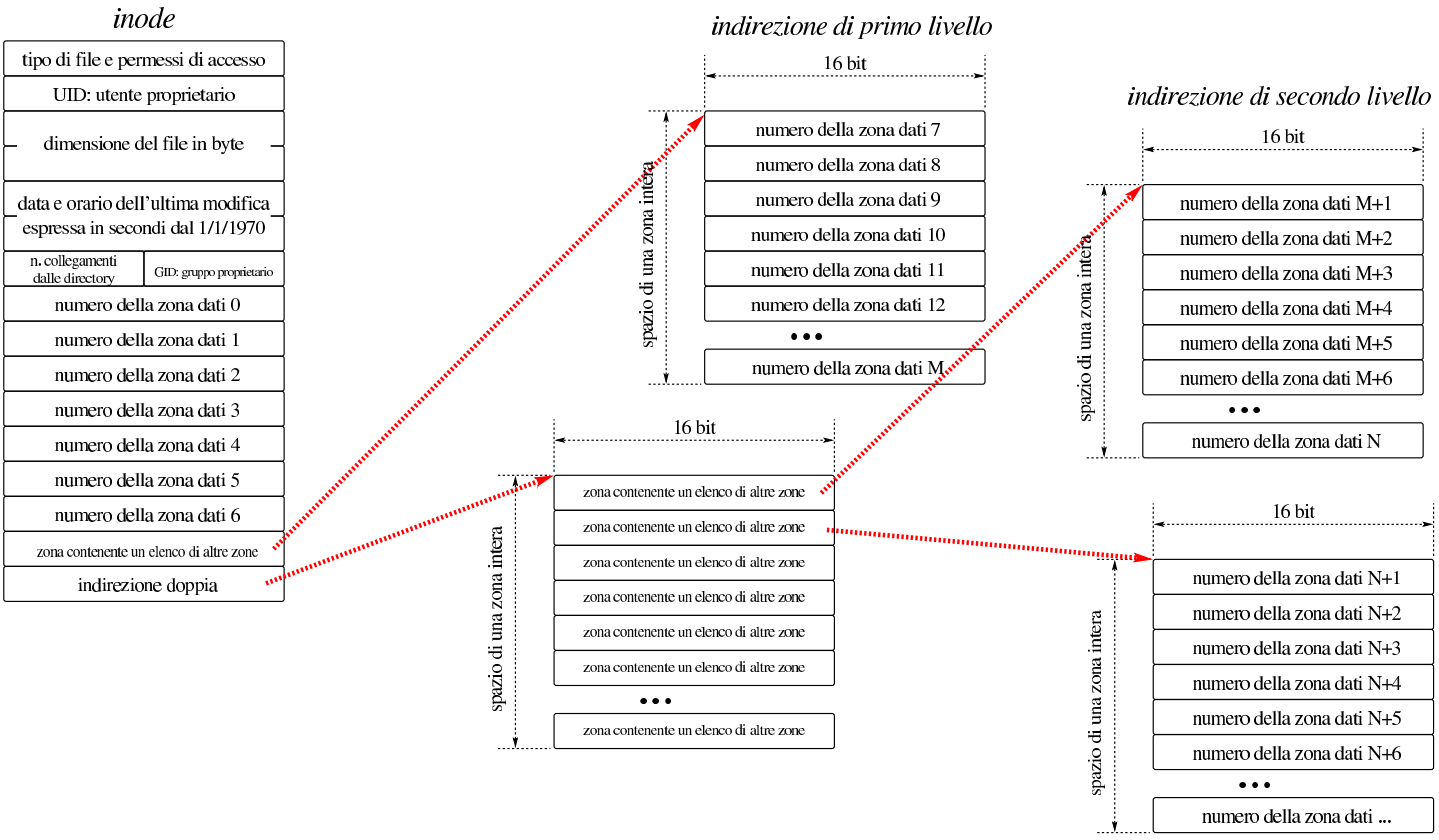
68.7.6 Inode

«

I blocchi successivi alla mappa delle zone dei dati, sono utilizzati per gli inode, di cui si conosce la quantità, perché questa è annotata nel super blocco. Nel file system Minix 1, ogni inode occupa esattamente 32 byte.



La figura successiva mostra il meccanismo usato per indirizzare file che occupano più di sette zone, attraverso elenchi aggiuntivi, ognuno dei quali occupa a sua volta una zona intera. I riferimenti indiretti alle zone possono essere quindi di primo livello, o di secondo livello, come suggerito dalla figura stessa.

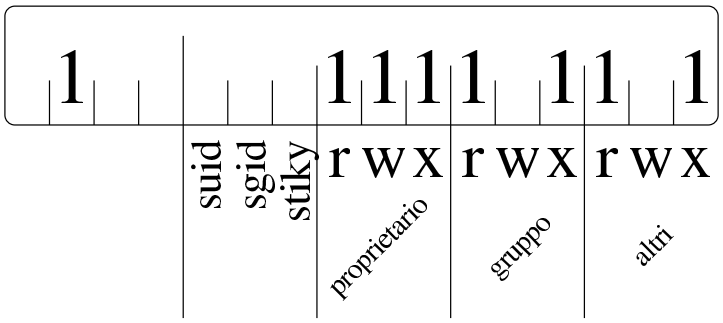


Ipotizzando di avere zone della stessa dimensione dei blocchi (1024 byte), dal momento che gli elenchi indiretti possono contenere a loro volta 512 numeri di zona, sarebbe possibile gestire file con una dimensione massima di 7+512+512×512 Kibyte, ovvero 262663 Kibyte. Disponendo di zone della dimensione di quattro blocchi, si potrebbero gestire file da 7+2048+2048×2048 Kibyte, ovvero 4196359 Kibyte. Con lo stesso criterio, con zone da otto blocchi, si potrebbero gestire file da poco più di 16 Gibyte; con zone da 16 blocchi si arriverebbe a poco più di 64 Gibyte. A questi limiti si aggiunge però il fatto che le zone sono individuate da numeri a 16 bit; pertanto, con zone da un solo blocco, si possono indirizzare al massimo 65536 Kibyte, ovvero 64 Mibyte; con zone da due blocchi si arriva a 128 Mibyte; con zone da 16 blocchi si arriva al massimo a 1 Gibyte. Pertanto, la doppia «indirezione» può essere usata solo parzialmente e non avrebbe senso un'indirezione tripla.

A parte la limitazione nella dimensione dei file, va annotato un fatto che può risultare più spiacevole: il numero del gruppo proprietario del file (GID) viene rappresentato con soli 8 bit. Ciò significa che si possono indicare gruppi fino al numero 255 e in pratica, quando vi si copia un file, il numero del gruppo viene troncato nella parte più significativa. Un altro limite importante riguarda il fatto che l'inode riporti solo la data di modifica del file, mancando così la data di accesso e la data di creazione dell'inode stesso.

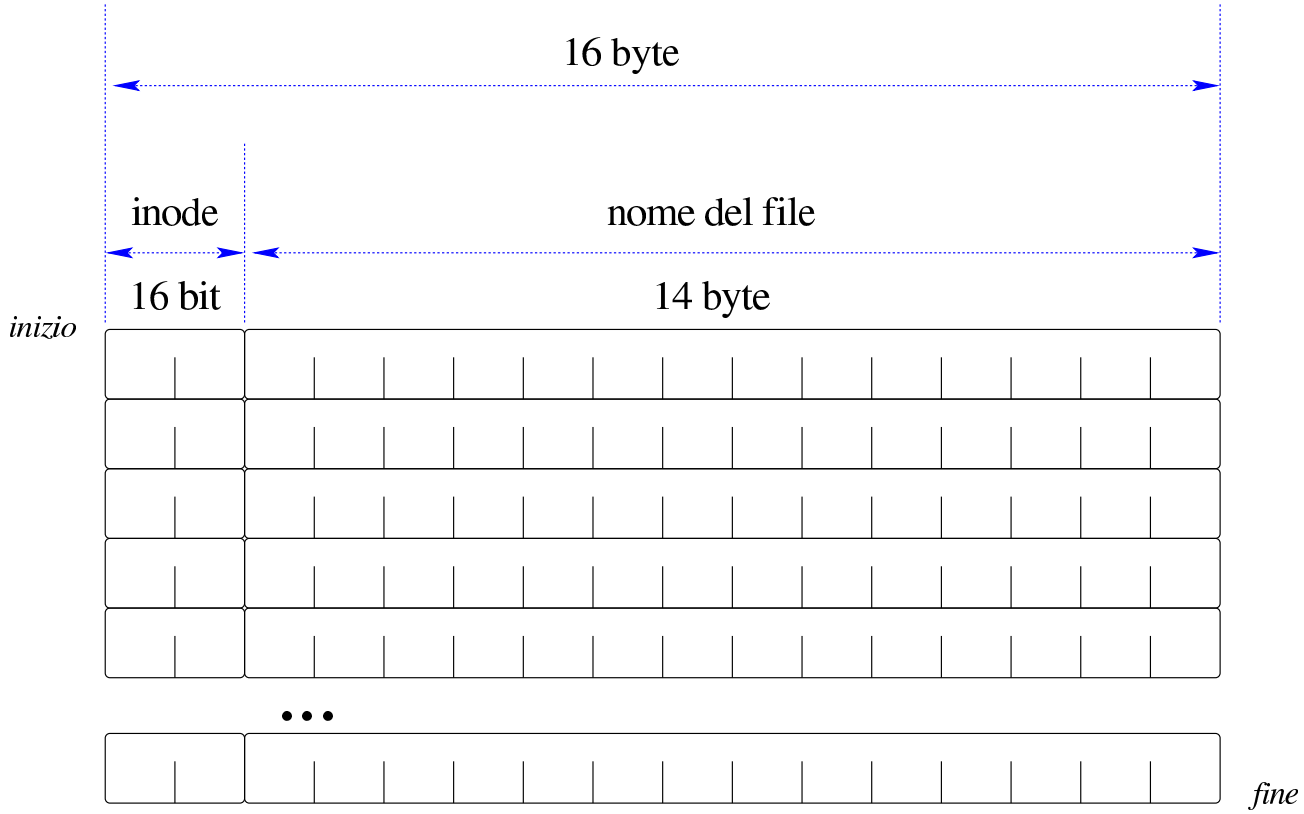
Il primo campo da 16 bit di un inode, rappresenta il tipo e i permessi del file a cui si riferisce l'inode (si veda anche la sezione 70.2.1 a proposito della «modalità» POSIX). L'interpretazione di questo valore deve avvenire secondo gli standard dei sistemi POSIX, ovvero secondo lo schema seguente, dove si ipotizza una directory con permessi di accesso e di lettura per tutti gli utenti:

0 0 0 1	file FIFO
0 0 1 0	dispositivo a caratteri
0 1 0 0	directory
0 1 1 0	dispositivo a blocchi
1 0 0 0	file normale
1 0 1 0	collegamento simbolico
1 1 0 0	socket di dominio Unix



68.7.7 Directory

Le directory si collocano nelle zone dei dati, come gli altri file, e a loro si accede a partire da un inode (come per tutti gli altri file). La struttura di una directory è molto semplice, come si può vedere nella figura successiva:



In pratica, si tratta di un file suddiviso in *record* a dimensione fissa da 16 byte, dove i primi due byte rappresentano il numero inode del nome che occupa i restanti 14 byte. Il file system Minix 1 consente così di rappresentare nomi di file fino a un massimo di 14 caratteri, ma nei sistemi GNU/Linux si utilizza spesso un'estensione con directory aventi *record* da 32 byte, in modo da poter rappresentare nomi fino a 30 caratteri.

La directory è un file come gli altri, pertanto, per sapere quante sono le voci che la compongono, occorre conoscere la dimensione del file, come annotato nel suo inode. Per esempio, una directory con quattro voci (inclusi i nomi '.' e '..'), occupa 64 byte.

68.7.8 Il problema dell'inversione dei byte

«

Il sistema operativo Minix nasce negli anni 1980 per elaboratori a 16 bit con l'inversione dei byte (*little endian*). Per questa ragione, i byte che costituiscono l'organizzazione del file system Minix 1 sono invertiti. Ciò diventa un problema quando si legge il contenuto del file system in modo diretto, in esadecimale, perché tutte le voci che prevedono una rappresentazione a 16 o a 32 bit, vanno rovesciate in modo appropriato, per poterle interpretare correttamente. Per esempio, in un'altra sezione è stato descritto il modo in cui viene popolata la mappa degli inode e delle zone dei dati; ai problemi lì descritti si aggiungerebbe anche l'inversione dei byte.

Se si considera che un file system serve per scrivere dati anche su unità di memorizzazione rimovibili, utilizzabili presumibilmente su altri sistemi e altre architetture, sarebbe più appropriata una progettazione che preveda sempre la scrittura «ordinata» dei byte, come si fa per i dati trasmessi in rete. Nel caso particolare di Minix, con l'evolvere del sistema e con l'adattamento anche ad altre architetture, si è reso necessario considerare se il file system a cui si accede giunge ordinato secondo la propria architettura, oppure se per questa è inverso. In pratica, un numero magico $137F_{16}$ indica che il file system va bene così; altrimenti, il numero $7F13_{16}$ richiede che i valori a 16 e a 32 siano invertiti (byte per byte), per poter essere interpretati correttamente.

68.8 Creazione ed eliminazione di file di qualunque tipo

«

Un file «normale», definito in inglese come *regular file*, è il contenitore di una sequenza di byte, rappresentato in qualche modo nel

file system. Nei sistemi Unix, anche le directory sono dei file, benché si tratti evidentemente di un tipo speciale, per il quale si richiede un trattamento particolareggiato; inoltre, altri tipi di entità rientrano nella gestione complessiva del concetto di file per i sistemi Unix. Originariamente è stato usato il termine «nodo», da cui deriva il nome della funzione ***mknod()***, con cui si poteva creare qualunque tipo di file.

68.8.1 La funzione «mknod()»

La funzione ***mknod()***, dichiarata nel file di intestazione '`sys/stat.h`', potenzialmente, è in grado di creare qualunque tipo di file, ma completamente vuoto, ammesso che si tratti di un tipo di file che ha un contenuto rappresentato nel file system. Teoricamente questa funzione potrebbe creare anche delle directory, ammesso che il sistema operativo lo consenta, ma si tratterebbe comunque di directory prive delle voci obbligatorie '`.`' e '`..`', quindi si tratterebbe di directory incomplete ed errate per il file system.

```
int mknod (const char *path, mode_t mode, dev_t dev);
```

Il primo parametro della funzione è una stringa che rappresenta il percorso del file da creare nel file system; il secondo parametro, ***mode***, individua il tipo di file ed eventualmente i permessi di accesso; l'ultimo parametro, ***dev***, il numero del file di dispositivo, ammesso che si tratti della creazione di questo tipo di file. La tabella successiva elenca le macro-variabili da usare per comporre il valore del parametro ***mode***, usando l'operatore OR binario, avendo la cura di specificare una sola macro-variabile per il tipo.

Tabella 68.76. Macro-variabili per esprimere, complessivamente il tipo e i permessi di un file.

Macro-variabile	Valore numerico equivalente	Significato
`S_IFBLK`	vedere ‘`sys/stat.h`’	File di dispositivo a blocchi.
`S_IFCHR`	vedere ‘`sys/stat.h`’	File di dispositivo a caratteri.
`S_IFIFO`	vedere ‘`sys/stat.h`’	File FIFO.
`S_IFREG`	vedere ‘`sys/stat.h`’	File normale (*regular file*).
`S_IFDIR`	vedere ‘`sys/stat.h`’	Directory.
`S_IFLNK`	vedere ‘`sys/stat.h`’	Collegamento simbolico.
`S_IFSOCK`	vedere ‘`sys/stat.h`’	Socket di dominio Unix.
`S_ISUID`	4000_8	Rappresenta l’attivazione del bit S-UID.
`S_ISGID`	2000_8	Rappresenta l’attivazione del bit S-GID.
`S_ISVTX`	1000_8	Rappresenta l’attivazione del bit Sticky.
`S_IRWXU`	0700_8	Rappresenta tutti i permessi di lettura, scrittura ed esecuzione o accesso, per l’utente proprietario.
`S_IRUSR`	0400_8	Rappresenta il permesso di lettura per l’utente proprietario.
`S_IWUSR`	0200_8	Rappresenta il permesso di scrittura per l’utente proprietario.

Macro-variabile	Valore numerico equivalente	Significato
S_IXUSR	0100_8	Rappresenta il permesso di esecuzione o attraversamento per l'utente proprietario.
S_IRWXG	0070_8	Rappresenta tutti i permessi di lettura, scrittura ed esecuzione o accesso, per il gruppo proprietario.
S_IRGRP	0040_8	Rappresenta il permesso di lettura per il gruppo proprietario.
S_IWGRP	0020_8	Rappresenta il permesso di scrittura per il gruppo proprietario.
S_IXGRP	0010_8	Rappresenta il permesso di esecuzione o attraversamento per il gruppo proprietario.
S_IRWXO	0007_8	Rappresenta tutti i permessi di lettura, scrittura ed esecuzione o accesso, per gli altri utenti.
S_IROTH	0004_8	Rappresenta il permesso di lettura per gli altri utenti.
S_IWOTH	0002_8	Rappresenta il permesso di scrittura per gli altri utenti.
S_IXOTH	0001_8	Rappresenta il permesso di esecuzione o attraversamento per gli altri utenti.

68.8.1.1 Creazione di un file «normale»

L'esempio seguente mostra l'uso della funzione ***mknod()*** per la creazione di un file comune:

```
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
  int status;
  status = mknod ("/tmp/miofile",
                  (mode_t) (S_IFREG | S_IRUSR | S_IWUSR
                            | S_IXUSR),
                  (dev_t) 0);
  if (status != 0) perror (NULL);
  return (0);
}
```

68.8.1.2 Creazione di una directory

«

L'esempio seguente mostra un programma elementare che ha lo scopo di creare una directory vuota, priva anche delle voci obbligatorie '.' e '..'. In condizioni normali, il sistema operativo dovrebbe impedire tale azione, producendo un messaggio di errore.

```
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
  int status;
  status = mknod ("/tmp/miadir",
                  (mode_t) (S_IFDIR | S_IRUSR | S_IWUSR
                            | S_IXUSR),
                  (dev_t) 0);
  if (status != 0) perror (NULL);
  return (0);
}
```

68.8.1.3 Creazione di un file FIFO

Un file FIFO è un «condotto» (*pipe*) rappresentato da un file e si distingue dai condotti creati internamente, senza tale associazione simbolica. Pertanto si distingue anche tra «condotti con nome» (*pipe* con nome) o file FIFO e «condotti senza nome» (*pipe* senza nome) o solo *pipe*.

L'esempio seguente mostra un programma elementare che ha lo scopo di creare un file FIFO.

```
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
  int status;
  status = mknod ("/tmp/miofifo",
                  (mode_t) (S_IFIFO | S_IRUSR | S_IWUSR
                            | S_IXUSR),
                  (dev_t) 0);
  if (status != 0) perror (NULL);
  return (0);
}
```

68.8.1.4 Creazione di file di dispositivo

«

La creazione di un file di dispositivo richiede l'indicazione del numero del dispositivo. Ciò comporta una complicazione, perché quel numero contiene simultaneamente le informazioni sul numero primario e sul numero secondario.

Originariamente, il numero del dispositivo era formato da 16 bit, di cui l'ottetto (il byte) più significativo rappresentava il numero primario, mentre quello meno significativo il numero secondario. Dal momento che questa organizzazione è sì quella tradizionale, ma non è richiesta dallo standard, diventa necessario disporre di funzioni o macroistruzioni che aiutino a comporre correttamente il numero di dispositivo complessivo, o a estrapolare le sue componenti.

Vari sistemi che si rifanno al modello di Unix introducono tre funzioni o macroistruzioni, utili per manipolare i numeri di dispositivo. Purtroppo queste funzioni non sono standard, benché abbastanza

diffuse:

```
dev_t makedev (int major, int minor);
```

```
int   major   (dev_t device);
```

```
int   minor   (dev_t device);
```

La funzione ***makedev()*** assembla il numero primario e il numero secondario ottenuti come argomenti, restituendo un numero di dispositivo complessivo; per converso, le funzioni ***major()*** e ***minor()*** estrapolano rispettivamente il numero primario e il numero secondario, a partire da un numero di dispositivo complessivo. Tali funzioni dovrebbero essere dichiarate nel file di intestazione ‘`sys/types.h`’.

I due esempi seguenti mostrano la creazione di due file di dispositivo, uno a caratteri e uno a blocchi. Ci si avvale della funzione ***makedev()*** per assemblare il numero di dispositivo complessivo, a partire dal numero primario e dal numero secondario.

```
#include <sys/stat.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
  int status;
  status = mknod ("/tmp/miodev1",
                  (mode_t) (S_IFCHR | S_IRUSR | S_IWUSR
                            | S_IXUSR),
                  makedev (1, 2));
  if (status != 0) perror (NULL);
  return (0);
}
```

```
#include <sys/stat.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
  int status;
  status = mknod ("/tmp/miodev2",
                  (mode_t) (S_IFBLK | S_IRUSR | S_IWUSR
                            | S_IXUSR),
                  makedev (3, 4));
  if (status != 0) perror (NULL);
  return (0);
}
```

Va osservato che la creazione di un file di dispositivo dovrebbe risultare concessa solo a un processo in funzione con i privilegi dell'utente con numero UID pari a zero ('**root**').

68.8.2 La funzione «mkdir()»

La funzione ***mkdir()*** costituisce il modo corretto per creare una directory vuota (ma provvista delle voci '`.`' e '`..`' obbligatorie).

```
int mkdir (const char *path, mode_t mode);
```

A differenza di ***mknod()***, il parametro ***mode*** va usato esclusivamente per indicare i permessi di accesso richiesti, tenendo conto, naturalmente, che questi vengono filtrati ulteriormente in base alla maschera dei permessi (*user mask*). In altri termini, nel parametro ***mode*** non si può specificare il tipo di file, cosa che comunque sarebbe ignorata, dato che si tratta della creazione di una directory e di nulla altro.

```
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
  int status;
  status = mkdir ("/tmp/miadir",
                  (mode_t) (S_IRUSR | S_IWUSR | S_IXUSR));
  if (status != 0) perror (NULL);
  return (0);
}
```

L'esempio mostra la creazione della directory '`/tmp/miadir/`' in un programma completo e molto semplice.

68.8.3 La funzione «mkfifo()»

La funzione ***mkfifo()*** consente di creare un file FIFO, specificando il percorso e i permessi, in modo analogo a quanto si farebbe con ***mkdir()*** per la creazione delle directory, con la differenza che in questo caso l'uso della funzione ***mknod()*** sarebbe comunque corretto.

```
int mkfifo (const char *path, mode_t mode);
```

Segue un esempio molto semplice, al pari di quelli già apparsi nel capitolo.

```
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
  int status;
  status = mkfifo ("/tmp/miofifo",
                   (mode_t) (S_IRUSR | S_IWUSR | S_IXUSR));
  if (status != 0) perror (NULL);
  return (0);
}
```

68.8.4 La funzione «unlink()»

La funzione ***unlink()*** consente di eliminare un file, possibilmente di qualunque tipo, «scollegandolo» dalla directory a cui si riferisce il percorso indicato per l'operazione. Dal momento che un file è rappresentato in un file system Unix da un inode, tale inode viene eliminato effettivamente se non ci più altri riferimenti allo stesso.

In linea di principio, con ***unlink()*** non dovrebbe essere possibile la cancellazione di una directory; inoltre, se nel frattempo il file in questione risulta utilizzato da un processo, l'operazione di cancellazione (scollegamento) dovrebbe completarsi soltanto nel momento in cui il file risulta chiuso a tutti gli effetti.

```
int unlink (const char *path);
```

L'esempio seguente mostra la cancellazione del file '`/tmp/cancellami`':

```
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
  int status;
  status = unlink ("/tmp/cancellami");
  if (status != 0) perror (NULL);
  return (0);
}
```

68.8.5 La funzione «rmdir()»

La funzione ***rmdir()*** consente di eliminare una directory, purché vuota (contenente soltanto le voci '`.`' e '`..`'), specificandone il percorso.

```
int rmdir (const char *path);
```

L'esempio seguente mostra la cancellazione della directory '`/tmp/cancellami/`':

```
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
  int status;
  status = rmdir ("/tmp/cancellami");
  if (status != 0) perror (NULL);
  return (0);
}
```

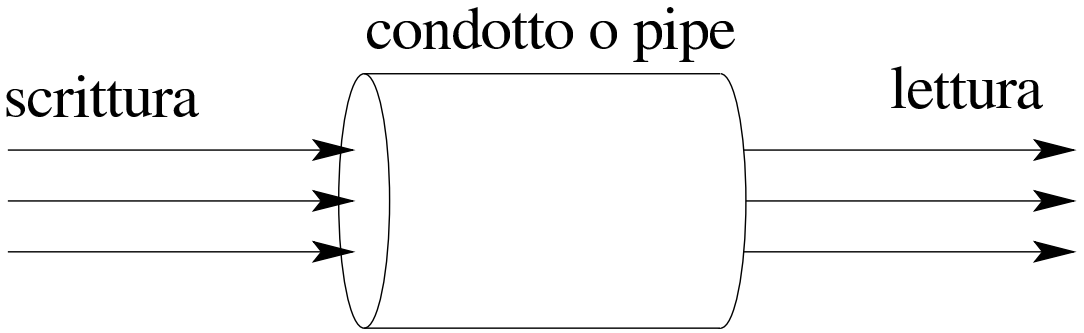
68.8.6 La funzione «remove()»

La funzione ***remove()*** cancella un file, utilizzando ***unlink()*** oppure ***rmdir()***, in base al tipo di file specificato per la rimozione stessa.

```
int remove (const char *path);
```

68.9 Condotti

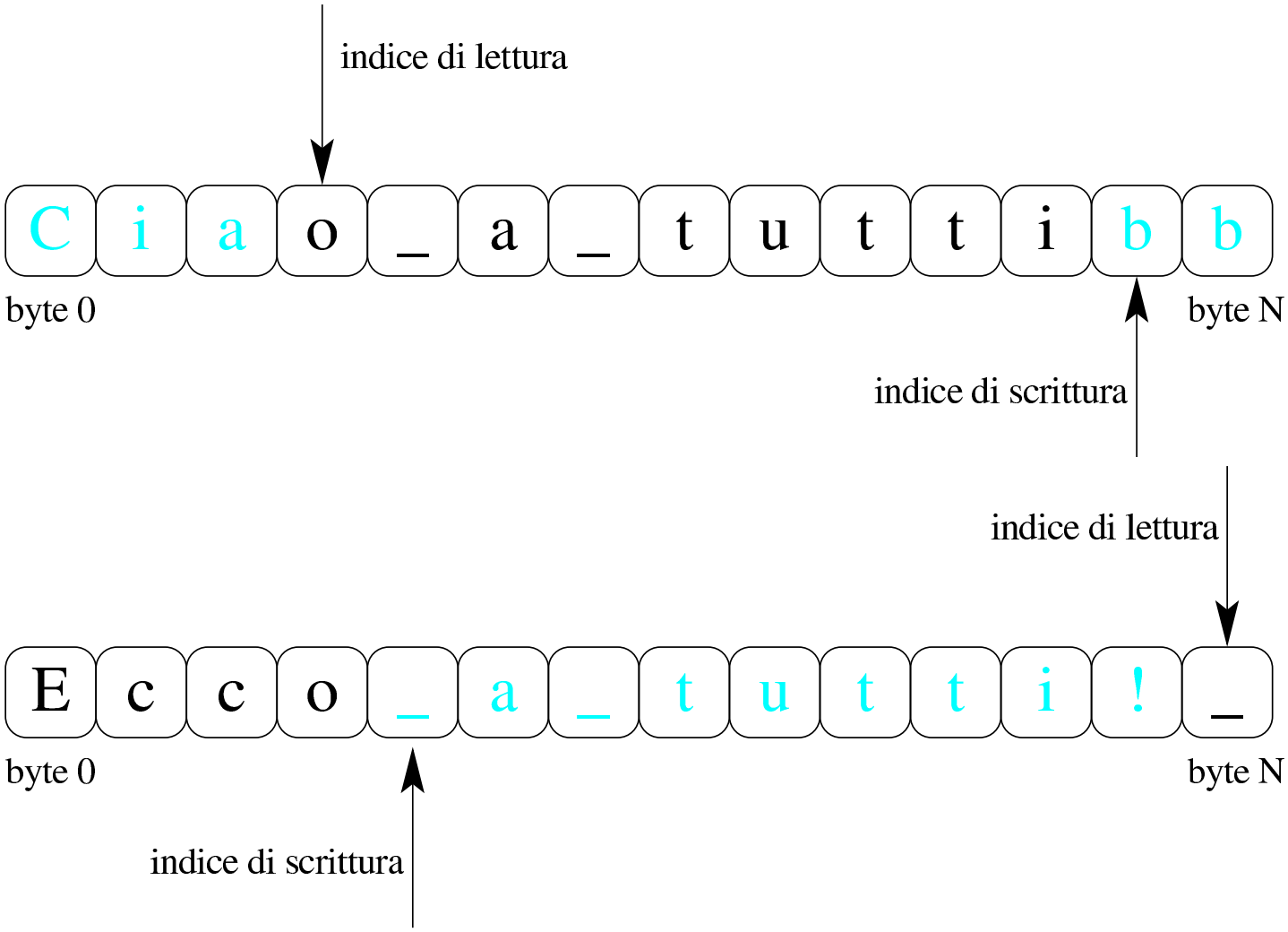
I ***condotti***, o *pipe*, sono dei file virtuali, ad accesso FIFO (*First in, first out*). In altri termini sono delle ***code***, in forma di file. Un condotto richiede che ci siano sia processi che vi scrivono, sia processi che vi leggono le informazioni. La lettura comporta il prelievo di dati e la liberazione di spazio disponibile per ulteriori operazioni di scrittura.



Se più di un processo apre uno stesso condotto in scrittura, non è possibile prevedere l'ordine in cui le operazioni di scrittura procedono; nello stesso modo, se più processi aprono uno stesso condotto in lettura, non è possibile prevedere con quale ordine vengano raccolti i dati dal condotto stesso. In altri termini, è compito dei processi di definire un protocollo tra di loro, se i dati devono confluire secondo un certo criterio.

68.9.1 Realizzazione materiale del condotto

In memoria centrale, il condotto si realizza come un array di byte, a cui si accede attraverso due indici: uno per la scrittura e l'altro per la lettura, tenendo conto che raggiunta la fine, si riprende dall'inizio.



Lo schema mostra un array scandito con due indici, dove i caratteri di colore nero rappresentano i byte scritti e ancora da leggere, mentre quelli in azzurro-ciano (ovvero quelli di colore più chiaro), rappresentano i byte già letti che possono essere sovrascritti. Lo schema mostra due momenti differenti, dove nel secondo caso l'indice di scrittura, una volta raggiunta la fine dell'array, riprende dall'inizio.

Dal momento che un condotto viene rappresentato in memoria come un inode, con tanto di elenco di riferimenti ai blocchi utilizzati nel file system, dato che tali annotazioni non servono perché nulla viene memorizzato in un file system, originariamente si utilizzava proprio quella porzione di memoria (quella dei blocchi diretti) per

l'array che consente di conservare temporaneamente i dati. Tuttavia, dal momento che lo standard di oggi richiede che lo spazio nella coda di un condotto sia abbastanza grande, è improbabile che si utilizzi ancora questo metodo.

68.9.2 Condotti «senza nome» e condotti «con nome»

Il condotto, come concetto, è un file virtuale già «aperto» e utilizzato da qualche processo elaborativo. In questi termini, un condotto potrebbe essere creato al volo, da un processo che successivamente ne avvia un altro con il quale deve comunicare. Un condotto realizzato al volo non ha alcun riferimento nel file system, pertanto gli si attribuisce la caratteristica di essere «senza nome». D'altro canto, un condotto può essere rappresentato nel file system da un file speciale, di tipo FIFO, da trattare come se fosse un file normale, benché non lo sia. Nel secondo caso si tratta di un condotto «con nome», perché c'è un nome nel file system, ovvero si tratta di un file FIFO.

68.9.3 Protocollo di accesso ai condotti

Una volta creato un condotto (che questo sia senza nome o che derivi dall'apertura di un file FIFO, ciò non fa differenza), solo dopo che questo risulta utilizzato sia in scrittura, sia in lettura, si può procedere con le operazioni di scrittura e lettura.

Quando un processo tenta di leggere da un condotto nel quale non sono disponibili dati nuovi, questo viene sospeso, in attesa di dati; nello stesso modo, un processo che tenta di scrivere in un condotto che non ha spazio disponibile (perché i dati già inseriti non sono ancora stati letti), viene sospeso in attesa di tale disponibilità.

Quando un processo tenta di leggere da un condotto che non viene più utilizzato in scrittura (perché non ci sono più descrittori di file associati al condotto in scrittura), si trova di fronte a un file concluso (nel senso che si avvera la condizione di fine del file); quando invece un processo tenta di scrivere in un condotto a cui non corrisponde più alcun descrittore in lettura, questo processo riceve il segnale SIGPIPE e, se il processo lo ignora, l'operazione di scrittura termina con un errore '**EPIPE**'.

Dal momento che la natura di un condotto è quella di essere ad accesso sequenziale, non è possibile posizionare l'indice di lettura o scrittura, con l'ausilio della funzione ***lseek()***.

68.9.4 Funzione «pipe()»

«

La funzione ***pipe()*** crea un condotto al volo, restituendo attraverso un array di due elementi i numeri dei descrittori per l'accesso in lettura e in scrittura. Tale condotto non ha un file FIFO corrispondente, ma risulta ugualmente aperto e disponibile al processo che lo crea.

```
int pipe (int fd[2]);
```

La funzione restituisce l'esito dell'operazione: zero in caso di successo, oppure -1 in caso di problemi (aggiornando il valore della variabile ***errno***). I descrittori per l'accesso vengono ottenuti dal contenuto dell'array ***fd[]***, dove ***fd[0]*** è il descrittore per l'accesso in lettura, mentre ***fd[1]*** è quello per l'accesso in scrittura.

Con questa funzione, un processo crea un condotto e ottiene i due descrittori di accesso; tuttavia, lo scopo di un condotto è quello di mettere in comunicazione due o più processi. Pertanto, dopo un'o-

perazione di questo tipo, si passa quasi certamente a una biforcazione ed eventualmente all'esecuzione di un altro programma. Dopo la biforcazione è normale che il processo originario chiuda il descrittore che non gli serve (dal momento che utilizza probabilmente solo quello di scrittura o solo quello di lettura) e che così faccia anche il processo sdoppiato: il programma che eventualmente venisse caricato al posto del processo sdoppiato troverebbe già tutto pronto per iniziare a lavorare correttamente. Il listato successivo mostra un esempio, disponibile eventualmente presso *allegati/c/esempio-posix-pipe.c* , ottenuto modificando un esempio analogo che appare nella pagina di manuale *pipe(2)* di un sistema GNU/Linux.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//-----------------------------------------------------
int
main (void)
{
  int             pipefd[2];
  pid_t           child;
  char            buffer;
  char           *message = "ciao a tutti voi amici "
                            "vicini e lontani\n";
  int             i;
  size_t          size;
  ssize_t         written;
  //
```

```
  //
  //
  if (pipe (pipefd) == -1)
    {
      perror ("pipe");
      exit (EXIT_FAILURE);
    }
  //
  //
  //
  child = fork ();
  if (child == -1)
    {
      perror ("fork");
      exit (EXIT_FAILURE);
    }
  //
  //
  //
  if (child == 0)
    {
      //
      // Questo è il figlio e deve leggere dal condotto;
      // pertanto, chiude il descrittore di scrittura nel
      // condotto.
      //
      close (pipefd[1]);
      //
      // Legge un byte alla volta, finché c'è qualcosa da
      // poter leggere.
      //
      while (read (pipefd[0], &buffer, 1) > 0)
        {
          //
```

```
            // Emette il byte letto attraverso lo standard
            // output.
            //
            write (STDOUT_FILENO, &buffer, 1);
          }
        //
        // Chiude il condotto e conclude il suo funzionamento.
        //
        close (pipefd[0]);
        //
        exit (EXIT_SUCCESS);
      }
    else
      {
        //
        // Questo è il genitore, il quale chiude il
        // descrittore di lettura nel condotto.
        //
        close (pipefd[0]);
        //
        // Inizia un ciclo senza fine.
        //
        while (1)
          {
            //
            // Scrive il messaggio contenuto nella stringa
            // 'message' attraverso il condotto.
            //
            for (i = 0, written = 0, size = strlen (message);
                 i < strlen (message);
                 i += written, size -= written)
              {
                written = write (pipefd[1], &message[i],
                                 size);
```

```
                if (written < 0)
                  {
                    //
                    // Essendosi verificato un errore, chiude
                    // il descrittore del condotto e si mette
                    // in attesa della morte del proprio
                    // processo figlio. Al termine conclude il
                    // proprio funzionamento.
                    //
                    perror ("pipe");
                    close (pipefd[1]);
                    wait (NULL);  // Wait for child.
                    exit (EXIT_FAILURE);
                  }
              }
          }
        //
        // Dal momento che il codice precedente è racchiuso in
        // un ciclo infinito, ciò che segue non può essere mai
        // eseguito.
        // Comunque, nel caso si volesse gestire il ciclo
        // precedente, a questo punto verrebbe chiuso il
        // condotto da parte del processo genitore, attendendo
        // la morte del proprio processo figlio, prima di
        // concludere regolarmente il proprio funzionamento.
        //
        close (pipefd[1]);
        wait (NULL);
        exit (EXIT_SUCCESS);
    }
}
```

Il programma dell'esempio, sdoppiandosi in due processi, da un lato (quello del genitore) scrive nel condotto la stringa ***message***

una quantità di volte indefinita, mentre dall'altro legge dal condotto il messaggio riproducendolo attraverso lo standard output sullo schermo, fino a quando uno dei due processi viene interrotto.

68.9.5 Esempio di condotto attraverso un file FIFO

Nella sezione precedente appare un esempio completo di programma che crea e utilizza un condotto. Nel listato successivo, disponibile eventualmente presso *allegati/c/esempio-posix-fifo.c* , si vede un altro esempio, pressocché equivalente, in cui i due processi prodotti comunicano attraverso un condotto derivante da un file FIFO. Il file FIFO viene creato preventivamente dal processo genitore, prima di sdoppiarsi, poi i due processi aprono il file e riproducono lo stesso comportamento già descritto nella sezione precedente.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/wait.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
//-------------------------------------------------------
int
main (void)
{
  int             fd;
  pid_t           child;
  char            buffer;
  char           *message = "ciao a tutti voi amici "
                            "vicini e lontani\n";
  int             i;
  size_t          size;
```

```
  ssize_t          written;
  int              status;
  //
  // Prima di creare il file FIFO, cancella quello che
  // potrebbe esserci già.
  //
  unlink ("/tmp/fifo");
  //
  // Usa 'mknod()', ma potrebbe usare 'mkfifo()' per creare
  // il file FIFO.
  //
  status = mknod ("/tmp/fifo",
                  S_IFIFO | S_IRUSR | S_IWUSR, 0);
  if (status != 0)
    {
      perror ("mknod fifo");
      exit (EXIT_FAILURE);
    }
  //
  // Il processo si sdoppia.
  //
  child = fork ();
  if (child == -1)
    {
      perror ("fork");
      exit (EXIT_FAILURE);
    }
  //
  //
  //
  if (child == 0)
    {
      //
      // Questo è il processo figlio che apre il file FIFO
```

```
      // in lettura.
      //
      fd = open ("/tmp/fifo", O_RDONLY);
      if (fd < 0)
        {
          perror ("fifo read open");
          exit (EXIT_FAILURE);
        }
      //
      // Legge un byte alla volta riproducendolo attraverso
      // lo standard output.
      //
      while (read (fd, &buffer, 1) > 0)
        {
          write (STDOUT_FILENO, &buffer, 1);
        }
      //
      // Chiude il file FIFO e si conclude il funzionamento
      // del processo figlio.
      //
      close (fd);
      //
      exit (EXIT_SUCCESS);
    }
  else
    {
      //
      // Questo è il processo genitore che apre il file
      // FIFO in scrittura.
      //
      fd = open ("/tmp/fifo", O_WRONLY);
      if (fd < 0)
        {
          perror ("fifo write open");
```

```
        exit (EXIT_FAILURE);
      }
    //
    // Inizia un ciclo infinito.
    //
    while (1)
      {
        //
        // Scrive il messaggio contenuto nella stringa
        // 'message' attraverso il condotto.
        //
        for (i = 0, written = 0, size = strlen (message);
             i < strlen (message);
             i += written, size -= written)
          {
            written = write (fd, &message[i], size);
            if (written < 0)
              {
                //
                // Essendosi verificato un errore, chiude
                // il file FIFO e si mette in attesa della
                // morte del proprio processo figlio.
                // Al termine conclude il proprio
                // funzionamento.
                //
                perror ("pipe");
                close (fd);
                wait (NULL);
                exit (EXIT_FAILURE);
              }
          }
      }
    //
    // Dal momento che il codice precedente è racchiuso
```

```
        // in un ciclo infinito, ciò che segue non può essere
        // mai eseguito.
        // Comunque, nel caso si volesse gestire il ciclo
        // precedente, a questo punto verrebbe chiuso il file
        // FIFO da parte del processo genitore, attendendo la
        // morte del proprio processo figlio, prima di
        // concludere regolarmente il proprio funzionamento.
        //
        close (fd);
        wait (NULL);
        exit (EXIT_SUCCESS);
    }
}
```

68.10 Lettura delle directory

Le directory si creano con la funzione ***mkdir()***, in modo da garantire che le voci obbligatorie '`.`' e '`..`' siano presenti. Successivamente, l'aggiornamento delle directory avviene in modo trasparente da parte del sistema operativo, in base alle operazioni di creazione ed eliminazione dei file. Rimane il problema della lettura delle directory, dalla quale ottenere nomi e riferimenti a inode, per poter conoscere il loro «contenuto».

Di norma è possibile leggere le directory come se fossero dei file puri e semplici, ma così facendo occorre interpretarne il contenuto secondo le regole di quel file system particolare. In generale ciò è sconsigliabile, pertanto vengono in aiuto alcune funzioni descritte nel file di intestazione '`dirent.h`' (si veda anche la sezione 70.6 sul file di intestazione '`dirent.h`').

68.10.1 Tipi derivati

La gestione delle directory, secondo il file di intestazione ‘`dirent.h`’, prevede due tipi derivati: ‘**`DIR`**’ e ‘**`struct dirent`**’. Il tipo ‘**`DIR`**’ serve a rappresentare una variabile strutturata con tutte le informazioni relative a un flusso di file riferito a una directory. In altri termini, è l’equivalente del tipo ‘**`FILE`**’, ma utile solo per l’accesso alle directory. Il tipo ‘**`struct dirent`**’ serve a poter rappresentare i due componenti indispensabili di ogni voce di directory dei sistemi Unix: numero di inode e nome. Pertanto, il tipo ‘**`struct dirent`**’ contiene almeno i membri ‘**`d_ino`**’ e ‘**`d_name`**’, per contenere rispettivamente il numero di inode e il nome relativo:

```
struct dirent {
    ino_t  d_ino;
    char   d_name[];
    …
};
```

68.10.2 Procedura per accedere a una directory

Per accedere a una directory, occorre prima aprirla, con la funzione ***opendir()***, la quale restituisce un puntatore a una variabile di tipo ‘**`DIR`**’, dove tale puntatore rappresenta così la directory in forma di flusso.

```
DIR *opendir (const char *path);
```

La lettura di una directory avviene a blocchi di una voce per volta, utilizzando la funzione ***readdir()***: ogni lettura produce la voce

successiva della directory, in forma di variabile strutturata di tipo '**`struct dirent`**', della quale si ottiene il puntatore.

```
struct dirent *readdir (DIR *dp);
```

Evidentemente, da come è strutturato il prototipo della funzione, si intuisce che la variabile strutturata a cui punta ciò che restituisce la funzione stessa, deve trovarsi in una zona di memoria statica, la quale viene riutilizzata ogni volta che si chiama la funzione ***readdir()***.

Dato che le letture si susseguono in modo sequenziale, quando si vuole che la prossima lettura ricominci dalla prima voce, si utilizza la funzione ***rewinddir()***:

```
void rewinddir (DIR *dp);
```

Al termine, come per i file normali, il flusso aperto di directory va chiuso con la funzione ***closedir()***, dove il valore restituito rappresenta il successo o meno dell'operazione:

```
int closedir (DIR *dp);
```

68.10.3 Attributo «FD_CLOEXEC»

Un sistema Unix dispone di un metodo di accesso ai file basato sui descrittori, sopra il quale si inserisce la gestione dei flussi di file. Per quanto riguarda le directory, lo standard non specifica se i flussi relativi debbano avvalersi dei descrittori o meno. Tuttavia, se si usano i descrittori, si presenta una situazione particolare: se si esegue un

altro processo, in sostituzione di quello in corso (con una delle funzioni ***exec*...*()***), i descrittori aperti vengono ereditati tali e quali dal nuovo processo. Nel caso delle directory, ciò va evitato.

Nei sistemi in cui il tipo '**DIR**' viene gestito tramite riferimenti a descrittori, l'apertura di una directory comporta l'attivazione dell'indicatore rappresentato dalla macro-variabile ***FD_CLOEXEC*** (file di intestazione '`fcntl.h`'), con il quale si assicura che il descrittore venga chiuso nel caso di utilizzo di una funzione ***exec*...*()***.

68.10.4 Esempio di utilizzo delle funzioni di accesso alle directory

«

Il listato successivo mostra un esempio molto semplice di programma che legge la directory corrente, mostrando l'elenco dei nomi che contiene, senza indicare però altre informazioni. Eventualmente si può ottenere il file dell'esempio dall'indirizzo *allegati/c/esempio-posix-dirent.c*

```
#include <errno.h>
#include <stdio.h>
#include <dirent.h>
//-------------------------------------------------------
int
main (int argc, char *argv[], char *envp[])
{
  DIR            *dp;
  struct dirent  *dir;
  //
  dp = opendir (".");
  if (dp == NULL)
    {
      perror (NULL);
      return (1);
    }
  //
  while ((dir = readdir (dp)) != NULL)
    {
      printf ("%s\n", dir->d_name);
    }
  //
  closedir (dp);
  //
  return (0);
}
```

Eventualmente si veda una realizzazione molto semplice del programma '**ls**' nei sorgenti di os32 (listato 96.1.23).

68.11 Riferimenti

- Maurice J. Bach, *The design of the UNIX operating system*, Prentice Hall, 1990, ISBN 0132017997

- The Open Group, *The UNIX System*, http://www.unix.org/
- Free Software Foundation, *The GNU C Library*, http://www.gnu.org/software/libc/manual/
- The Open Group, *The Single UNIX® Specification, Version 2, Regular Expressions*, http://pubs.opengroup.org/onlinepubs/000095399/basedefs/xbd_chap09.html
- Daniel Robbins, *POSIX threads explained*, http://www.ibm.com/developerworks/library/l-posix1/ , http://www.ibm.com/developerworks/library/l-posix2/ , http://www.ibm.com/developerworks/library/l-posix3/
- pagina di manuale *pthreads(7)* di un sistema GNU/Linux
- Mark Hayes, *POSIX threads tutorial*, http://math.arizona.edu/~swig/documentation/pthreads/
- The Open Group, *The Single UNIX® Specification, Version 2, pthread.h*, http://pubs.opengroup.org/onlinepubs/000095399/basedefs/pthread.h.html
- The Open Group, *The Single UNIX® Specification, Version 2, dirent.h*, http://pubs.opengroup.org/onlinepubs/000095399/basedefs/dirent.h.html
- Pagine di manuale: *mknod(2)*, *mkdir(2)*, *mkfifo(3)*, *unlink(2)*, *pipe(2)*, *mkfifo(3)*, *opendir(3)*, *readdir(3)*, *rewinddir(3)*, *closedir(3)*

[1] Per la libreria POSIX, la gestione dei flussi del linguaggio C è costruita avvalendosi del sistema dei descrittori, con l'aggiunta però di una memoria tampone.

[2] Nel programma di esempio si può fare sicuramente di meglio, incrementando direttamente la variabile globale, senza tanti travasi come invece viene fatto. Ma lo scopo di questi esempi è simulare una situazione più complessa, senza complicazioni che esulano dal problema specifico che si vuole descrivere.

Capitolo 69 Libreria C, con qualche estensione POSIX

69.1 Funzionalità di libreria non dichiarate 1069
69.2 File «assert.h» 1069
69.2.1 Utilizzo 1070
69.3 File «limits.h» 1071
69.3.1 Confronto tra architetture 1075
69.3.2 Estensioni POSIX 1075
69.4 File «stdint.h» 1077
69.4.1 Tipi interi ad ampiezza esatta 1078
69.4.2 Tipi interi di rango minimo 1079
69.4.3 Tipi interi «veloci» 1082
69.4.4 Tipi interi per rappresentare dei puntatori . . 1084
69.4.5 Tipi interi di rango massimo 1085
69.4.6 Limiti per altri tipi interi 1086
69.5 File «errno.h» 1087
69.6 File «locale.h» 1092
69.6.1 Impostazione della configurazione locale 1093
69.6.2 Composizione dei valori numerici 1096
69.7 File «ctype.h» 1101
69.7.1 Funzioni «is...()» 1102

69.7.2 macroistruzioni «is...()» 1116
69.7.3 Funzioni di conversione 1117
69.7.4 macroistruzioni di conversione 1120
69.7.5 Esempio di utilizzo delle funzioni 1120
69.8 File «stdarg.h» 1123
69.8.1 Realizzazione 1124
69.8.2 Esempio di utilizzo delle macro 1125
69.8.3 Promozione 1127
69.9 File «stdlib.h» 1127
69.9.1 Tipi di dati speciali 1128
69.9.2 Macro-variabili 1128
69.9.3 Conversioni numeriche 1130
69.9.4 Funzioni per la generazione di numeri in modo pseudo-casuale 1136
69.9.5 Funzioni standard per la generazione di numeri pseudo-casuali 1139
69.9.6 Amministrazione della memoria 1140
69.9.7 Conclusione forzata del programma 1141
69.9.8 Funzioni di comunicazione con l'ambiente 1142
69.9.9 Funzioni di ricerca e riordino 1143
69.9.10 Funzioni per l'aritmetica con i numeri interi 1146
69.9.11 Funzioni per la gestione di caratteri estesi e sequenze multibyte 1148
69.9.12 Funzione «mblen()» 1149

69.9.13 Funzioni «mbtowc()» e «wctomb()» 1151
69.9.14 Funzioni «mbstowcs()» e «wcstombs()» 1153
69.10 File «inttypes.h» 1156
69.10.1 Divisione intera con interi di rango massimo . 1157
69.10.2 Macro-variabili in qualità di specificatori di conversione 1158
69.10.3 Valore assoluto 1165
69.10.4 Conversione da stringa a numero intero ... 1166
69.11 File «iso646.h» 1167
69.12 File «stdbool.h» 1168
69.13 File «stddef.h» 1169
69.14 File «string.h» 1170
69.14.1 Copia 1170
69.14.2 Concatenamento 1180
69.14.3 Comparazione 1184
69.14.4 Ricerca 1191
69.14.5 Funzioni varie 1217
69.15 File «signal.h» 1223
69.15.1 Dichiarazione contorta 1223
69.15.2 Tipo speciale 1225
69.15.3 Denominazione dei segnali 1226
69.15.4 Segnali secondo POSIX 1227

69.15.5 Gestori fittizi di segnali 1231
69.15.6 Funzioni 1233
69.15.7 Esempio 1235
69.16 File «time.h» 1239
69.16.1 Il tempo di CPU 1239
69.16.2 Rappresentazione interna del tempo 1241
69.16.3 Rappresentazione strutturata del tempo 1242
69.16.4 Funzioni per l'elaborazione di valori legati al tempo 1243
69.16.5 Conversione in stringa 1247
69.17 File «stdio.h» 1254
69.17.1 Tipi 1255
69.17.2 Macro-variabili varie 1256
69.17.3 Ipotesi di gestione del tipo «FILE» 1259
69.17.4 Flussi standard 1260
69.17.5 Funzioni per la rimozione e la ridenominazione dei file 1261
69.17.6 Funzioni per la gestione dei file temporanei . 1261
69.17.7 Funzioni per l'apertura e la chiusura dei flussi di file 1264
69.17.8 Funzioni per la gestione della memoria tampone 1268
69.17.9 Funzioni per la composizione dell'output 1270
69.17.10 Funzioni per l'interpretazione dell'input 1280

69.17.11 Funzioni per la lettura e la scrittura di un carattere alla volta 1289
69.17.12 Funzioni per l'input e l'output di file di testo 1292
69.17.13 Funzioni per l'input e output diretto 1294
69.17.14 Funzioni per il posizionamento 1295
69.17.15 Accesso esclusivo ai flussi di file 1297
69.17.16 Condotti 1299
69.17.17 Informazioni sul terminale 1299
69.17.18 Gestione degli errori 1300
69.17.19 Realizzazione di «vsnprintf()» e altre collegate 1301
69.18 Riferimenti 1302

abort() 1141 abs() 1146 and 1167 and_eq 1167 asctime() 1247 assert() 1069 assert.h 1069 atexit() 1141 atof() 1130 atoi() 1130 atol() 1130 atoll() 1130 bitand 1167 bool 1168 bsearch() 1143 BUFSIZ 1256 calloc() 1140 CHAR_BIT 1071 CHAR_MAX 1071 CHAR_MIN 1071 clearerr() 1300 clock() 1239 CLOCKS_PER_SEC 1239 clock_t 1239 compl 1167 ctermid() 1299 ctime() 1248 ctype.h 1101 difftime() 1244 div() 1146 div_t 1128 1128 EDOM 1087 EILSEQ 1087 EOF 1256 ERANGE 1087 errno 1087 errno.h 1087 exit() 1141 EXIT_FAILURE 1128 EXIT_SUCCESS 1128 false 1168 fclose() 1264 fdopen() 1264 feof() 1300 ferror() 1300 fflush() 1268 fgetc() 1289 fgetpos() 1295 fgets() 1292 FILE 1255 FILENAME_MAX 1256 flockfile() 1297 fopen() 1264 FOPEN_MAX 1256

fpos_t 1255 fprintf() 1278 fputc() 1289 fputs() 1292
fread() 1294 free() 1140 freopen() 1264 fscanf()
1287 fseek() 1295 fseeko() 1295 fsetpos() 1295
ftell() 1295 ftello() 1295 ftrylockfile() 1297
funlockfile() 1297 fwrite() 1294 getc() 1289
getchar() 1289 getchar_unlocked() 1297
getc_unlocked() 1297 getenv() 1142 gets() 1292
gmtime() 1246 imaxabs() 1165 imaxdiv() 1157
imaxdiv_t 1157 INT16_C() 1079 INT16_MAX 1078
INT16_MIN 1078 int16_t 1078 INT32_C() 1079
INT32_MAX 1078 INT32_MIN 1078 int32_t 1078
INT64_C() 1079 INT64_MAX 1078 INT64_MIN 1078
int64_t 1078 INT8_C() 1079 INT8_MAX 1078 INT8_MIN
1078 int8_t 1078 INTMAX_C() 1085 INTMAX_MAX 1085
INTMAX_MIN 1085 intmax_t 1085 INTPTR_MAX 1084
INTPTR_MIN 1084 intptr_t 1084 inttypes.h 1156
INT_FAST16_MAX 1082 INT_FAST16_MIN 1082
int_fast16_t 1082 INT_FAST32_MAX 1082
INT_FAST32_MIN 1082 int_fast32_t 1082
INT_FAST64_MAX 1082 INT_FAST64_MIN 1082
int_fast64_t 1082 INT_FAST8_MAX 1082
INT_FAST8_MIN 1082 int_fast8_t 1082
INT_LEAST16_MAX 1079 INT_LEAST16_MIN 1079
int_least16_t 1079 INT_LEAST32_MAX 1079
INT_LEAST32_MIN 1079 int_least32_t 1079
INT_LEAST64_MAX 1079 INT_LEAST64_MIN 1079
int_least64_t 1079 INT_LEAST8_MAX 1079
INT_LEAST8_MIN 1079 int_least8_t 1079 INT_MAX 1071
INT_MIN 1071 isalnum() 1102 isalpha() 1103

isascii() 1114 isblank() 1104 iscntrl() 1105
isdigit() 1106 isgraph() 1107 islower() 1108
iso646.h 1167 isprint() 1109 ispunct() 1110
isspace() 1111 isupper() 1112 isxdigit() 1113
labs() 1146 LC_TIME 1249 ldiv() 1146 ldiv_t 1128
limits.h 1071 llabs() 1146 lldiv() 1146 lldiv_t 1128
LLONG_MAX 1071 LLONG_MIN 1071 locale.h 1092
localtime() 1246 LONG_BIT 1075 LONG_MAX 1071
LONG_MIN 1071 L_ctermid 1256 L_tmpnam 1256
malloc() 1140 mblen() 1149 mbstowcs() 1153
mbtowc() 1151 MB_CUR_MAX 1128 MB_LEN_MAX 1071
memccpy() 1172 memchr() 1192 memcmp() 1184
memcpy() 1171 memmove() 1174 memset() 1217
mktime() 1244 NDEBUG 1069 not 1167 not_eq 1167 NULL
1169 offsetof 1169 or 1167 or_eq 1167 pclose() 1299
perror() 1300 popen() 1299 PRId16 1158 PRId32 1158
PRId64 1158 PRId8 1158 PRIdFAST16 1158 PRIdFAST32
1158 PRIdFAST64 1158 PRIdFAST8 1158 PRIdLEAST16
1158 PRIdLEAST32 1158 PRIdLEAST64 1158 PRIdLEAST8
1158 PRIdMAX 1158 PRIdPTR 1158 PRIi16 1158 PRIi32
1158 PRIi64 1158 PRIi8 1158 PRIiFAST16 1158
PRIiFAST32 1158 PRIiFAST64 1158 PRIiFAST8 1158
PRIiLEAST16 1158 PRIiLEAST32 1158 PRIiLEAST64 1158
PRIiLEAST8 1158 PRIiMAX 1158 PRIiPTR 1158 printf()
1278 PRIo16 1158 PRIo32 1158 PRIo64 1158 PRIo8 1158
PRIoFAST16 1158 PRIoFAST32 1158 PRIoFAST64 1158
PRIoFAST8 1158 PRIoLEAST16 1158 PRIoLEAST32 1158
PRIoLEAST64 1158 PRIoLEAST8 1158 PRIoMAX 1158
PRIoPTR 1158 PRIu16 1158 PRIu32 1158 PRIu64 1158

PRIu8 1158 PRIuFAST16 1158 PRIuFAST32 1158 PRIuFAST64 1158 PRIuFAST8 1158 PRIuLEAST16 1158 PRIuLEAST32 1158 PRIuLEAST64 1158 PRIuLEAST8 1158 PRIuMAX 1158 PRIuPTR 1158 PRIx16 1158 PRIX16 1158 PRIx32 1158 PRIX32 1158 PRIx64 1158 PRIX64 1158 PRIX8 1158 PRIx8 1158 PRIxFAST16 1158 PRIXFAST16 1158 PRIxFAST32 1158 PRIXFAST32 1158 PRIxFAST64 1158 PRIXFAST64 1158 PRIXFAST8 1158 PRIxFAST8 1158 PRIxLEAST16 1158 PRIXLEAST16 1158 PRIxLEAST32 1158 PRIXLEAST32 1158 PRIxLEAST64 1158 PRIXLEAST64 1158 PRIXLEAST8 1158 PRIxLEAST8 1158 PRIxMAX 1158 PRIXMAX 1158 PRIxPTR 1158 PRIXPTR 1158 PTRDIFF_MAX 1086 PTRDIFF_MIN 1086 ptrdiff_t 1086 1169 putc() 1289 putchar() 1289 putchar_unlocked() 1297 putc_unlocked() 1297 puts() 1292 P_tmpdir 1256 qsort() 1143 raise() 1233 rand() 1136 RAND_MAX 1128 realloc() 1140 remove() 1261 rename() 1261 rewind() 1295 scanf() 1287 SCHAR_MAX 1071 SCHAR_MIN 1071 SCNd16 1158 SCNd32 1158 SCNd64 1158 SCNd8 1158 SCNdFAST16 1158 SCNdFAST32 1158 SCNdFAST64 1158 SCNdFAST8 1158 SCNdLEAST16 1158 SCNdLEAST32 1158 SCNdLEAST64 1158 SCNdLEAST8 1158 SCNdMAX 1158 SCNdPTR 1158 SCNi16 1158 SCNi32 1158 SCNi64 1158 SCNi8 1158 SCNiFAST16 1158 SCNiFAST32 1158 SCNiFAST64 1158 SCNiFAST8 1158 SCNiLEAST16 1158 SCNiLEAST32 1158 SCNiLEAST64 1158 SCNiLEAST8 1158 SCNiMAX 1158 SCNiPTR 1158 SCNo16 1158 SCNo32 1158 SCNo64 1158 SCNo8 1158 SCNoFAST16 1158 SCNoFAST32 1158 SCNoFAST64 1158 SCNoFAST8 1158

SCNoLEAST16 1158 SCNoLEAST32 1158 SCNoLEAST64 1158 SCNoLEAST8 1158 SCNoMAX 1158 SCNoPTR 1158 SCNu16 1158 SCNu32 1158 SCNu64 1158 SCNu8 1158 SCNuFAST16 1158 SCNuFAST32 1158 SCNuFAST64 1158 SCNuFAST8 1158 SCNuLEAST16 1158 SCNuLEAST32 1158 SCNuLEAST64 1158 SCNuLEAST8 1158 SCNuMAX 1158 SCNuPTR 1158 SCNx16 1158 SCNx32 1158 SCNx64 1158 SCNx8 1158 SCNxFAST16 1158 SCNxFAST32 1158 SCNxFAST64 1158 SCNxFAST8 1158 SCNxLEAST16 1158 SCNxLEAST32 1158 SCNxLEAST64 1158 SCNxLEAST8 1158 SCNxMAX 1158 SCNxPTR 1158 SEEK_CUR 1256 SEEK_END 1256 SEEK_SET 1256 setbuf() 1268 setvbuf() 1268 SHRT_MAX 1071 SHRT_MIN 1071 SIGABRT 1226 1227 SIGALRM 1227 SIGBUS 1227 SIGCHLD 1227 SIGCONT 1227 SIGFPE 1226 1227 SIGHUP 1227 SIGILL 1226 1227 SIGINT 1226 1227 SIGKILL 1227 signal() 1233 signal.h 1223 SIGPIPE 1227 SIGPOLL 1227 SIGPROF 1227 SIGQUIT 1227 SIGSEGV 1226 1227 SIGSTOP 1227 SIGSYS 1227 SIGTERM 1226 1227 SIGTRAP 1227 SIGTTIN 1227 SIGTTOU 1227 SIGURG 1227 SIGUSR1 1227 SIGUSR2 1227 SIGVTALRM 1227 SIGXCPU 1227 SIGXFSZ 1227 SIG_ATOMIC_MAX 1086 SIG_ATOMIC_MIN 1086 sig_atomic_t 1086 1225 SIG_DFL 1231 SIG_ERR 1231 SIG_IGN 1231 SIZE_MAX 1086 size_t 1086 1169 snprintf() 1278 sprintf() 1278 srand() 1136 sscanf() 1287 SSIZE_MAX 1075 stdarg.h 1123 stdbool.h 1168 stddef.h 1169 stderr 1260 stdint.h 1077 stdio 1260 stdio.h 1254 stdlib.h 1127 stdout 1260 strcat() 1181 strchr() 1193 strcmp() 1186 strcoll() 1188 strcpy() 1175 strcspn() 1198

strdup() 1178 strerror() 1219 strerror_r() 1220
strftime() 1249 string.h 1170 strlen() 1221
strncat() 1182 strncmp() 1188 strncpy() 1176
strpbrk() 1200 strrchr() 1195 strspn() 1197
strstr() 1202 strtod() 1130 strtof() 1130
strtoimax() 1166 strtok() 1204 strtok_r() 1211
strtol() 1130 strtold() 1130 strtoll() 1130
strtouimax() 1166 strtoul() 1130 strtoull() 1130
struct tm 1242 strxfrm() 1190 system() 1142
tempnam() 1261 time() 1243 time.h 1239 time_t 1241
1243 tmpfile() 1261 tmpnam() 1261 TMP_MAX 1256
toascii() 1120 tolower() 1117 toupper() 1118 true
1168 UCHAR_MAX 1071 UINT16_C() 1079 UINT16_MAX 1078
uint16_t 1078 UINT32_C() 1079 UINT32_MAX 1078
uint32_t 1078 UINT64_C() 1079 UINT64_MAX 1078
uint64_t 1078 UINT8_C() 1079 UINT8_MAX 1078
uint8_t 1078 UINTMAX_C() 1085 UINTMAX_MAX 1085
uintmax_t 1085 UINTPTR_MAX 1084 uintptr_t 1084
UINT_FAST16_MAX 1082 uint_fast16_t 1082
UINT_FAST32_MAX 1082 uint_fast32_t 1082
UINT_FAST64_MAX 1082 uint_fast64_t 1082
UINT_FAST8_MAX 1082 uint_fast8_t 1082
UINT_LEAST16_MAX 1079 uint_least16_t 1079
UINT_LEAST32_MAX 1079 uint_least32_t 1079
UINT_LEAST64_MAX 1079 uint_least64_t 1079
UINT_LEAST8_MAX 1079 uint_least8_t 1079 UINT_MAX
1071 ULLONG_MAX 1071 ULONG_MAX 1071 ungetc() 1289
USHRT_MAX 1071 va_arg() 1123 va_copy() 1123
va_end() 1123 va_list 1123 va_start() 1123

vfprintf() 1279 vfscanf() 1288 vprintf() 1279
vscanf() 1288 vsnprintf() 1279 vsprintf() 1279
vsscanf() 1288 WCHAR_MAX 1086 WCHAR_MIN 1086
wchar_t 1086 1169 wcstoimax() 1166 wcstombs() 1153
wcstouimax() 1166 wctomb() 1151 WINT_MAX 1086
WINT_MIN 1086 wint_t 1086 WORD_BIT 1075 xor 1167
xor_eq 1167 _Exit() 1141 _IOFBF 1256 _IOLBF 1256
_IONBF 1256 _POSIX2_... 1075 _POSIX_... 1075
XOPEN... 1075 __bool_true_false_are_defined
1168 __udivdi3() 1069 __umoddi3() 1069 %+... 1270 %...c
1270 %...d 1270 %...e 1270 %...f 1270 %...g 1270 %...hd 1270
%...hhd 1270 %...hhi 1270 %...hhn 1270 %...hho 1270 %...hhu
1270 %...hhx 1270 %...hi 1270 %...hn 1270 %...ho 1270 %...hu
1270 %...hx 1270 %...i 1270 %...lc 1270 %...ld 1270 %...Le 1270
%...Lf 1270 %...Lg 1270 %...li 1270 %...lld 1270 %...lli 1270
%...lln 1270 %...llo 1270 %...llu 1270 %...llx 1270 %...ln
1270 %...lo 1270 %...ls 1270 %...lu 1270 %...lx 1270 %...n 1270
%...o 1270 %...s 1270 %...u 1270 %...x 1270 %0... 1270 %-... 1270

Complessivamente, la libreria C è ciò che consente l'uso di funzioni, macroistruzioni e macro-variabili definite dallo standard (ed eventualmente dalle estensioni presenti nel proprio contesto). Generalmente le funzioni vengono fornite già compilate all'interno di una libreria dinamica o statica (per esempio possono essere i file '`/lib/libc.so`' o '`/usr/lib/libc.a`'), ma dal punto di vista formale, la libreria standard è percepita attraverso i file di intestazione.

Per la precisione, lo standard stabilisce che si debba fare riferimento a delle «intestazioni» nel sorgente di un programma scritto in linguaggio C, ma il contesto particolare può essere tale per cui queste potrebbero non esistere fisicamente come ci si attenderebbe da un sistema operativo tradizionale. Anche per questo, nella documentazione standard ci si riferisce solo a intestazioni, senza precisare che debba trattarsi di file.

In pratica, i file di intestazione, o ciò che ne fa la funzione, sono sempre necessari e al loro interno si dichiarano le macro-variabili, le macroistruzioni e i prototipi delle funzioni, le quali normalmente sono già precompilate in un file separato. A ogni modo, di norma il compilatore è predisposto per utilizzare automaticamente i file precompilati necessari.

Nei capitoli successivi vengono descritti alcuni dei file di intestazione previsti dallo standard del linguaggio, mostrando come potrebbero essere realizzati e, in alcuni casi, anche fornendo una soluzione completa per le funzioni (gli esempi dovrebbero essere disponibili a partire da *allegati/c/*).

La libreria C viene estesa dallo standard POSIX con componenti aggiuntivi. In alcuni casi, nei capitoli successivi, si fa riferimento anche a estensioni POSIX, con le annotazioni appropriate al riguardo. Va però osservato che per scrivere un programma in linguaggio C, che abbia la massima portabilità fra sistemi operativi molto differenti tra loro, occorre evitare il più possibile le estensioni di qualunque genere.

Ciò che non si vede negli esempi dei capitoli successivi è la tecnica comune che si usa per evitare di includere ricorsivamente lo stesso file di intestazione più volte: si associa a ogni file una macro-variabile e se all'inizio della lettura questa non risulta dichiarata, il contenuto viene acquisito, altrimenti viene ignorato semplicemente, perché deve essere già stato incluso in precedenza. L'esempio seguente riguarda il file '`limits.h`':

```
#ifndef _LIMITS_H
#define _LIMITS_H       1
    ...
    contenuto_del_file
    ...
#endif // _LIMITS_H
```

In pratica viene verificato se la macro-variabile ***_LIMITS_H*** è già stata definita; se lo è, il contenuto del file viene ignorato. Se invece la macro-variabile non è stata dichiarata, questa allora viene dichiarata e quindi si procede con il lavoro normale del file.

Tabella 69.2. File di intestazione standard.

Intestazione	Descrizione	Riferimenti
`assert.h`	Verifica diagnostica di un'espressione (asserzione).	sezione 69.2
`complex.h`	Aritmetica complessa.	--
`ctype.h`	Classificazione dei caratteri.	sezione 69.7
`errno.h`	Definizione degli errori.	sezione 69.5
`fenv.h`	Gestione di valori in virgola mobile.	--

Intestazione	Descrizione	Riferimenti
`float.h`	Limiti dei valori in virgola mobile.	--
`inttypes.h`	Estensione di ‘`stdint.h`’.	sezione 69.10
`iso646.h`	Macro-variabili da usare in sostituzione di vari operatori.	sezione 69.11
`limits.h`	Limiti per i numeri interi.	sezione 69.3
`locale.h`	Gestione della configurazione locale (nel senso di «localizzazione»).	sezione 69.6
`math.h`	Funzioni matematiche comuni.	--
`setjmp.h`	Funzionalità per il salto incondizionato.	--
`signal.h`	Gestione dei segnali.	sezione 69.15
`stdarg.h`	Gestione degli argomenti variabili.	sezione 69.8
`stdbool.h`	Tipo e valori booleani.	sezione 69.12
`stddef.h`	Definizioni comuni; in particolare i tipi ‘**`size_t`**’, ‘**`wchar_t`**’ e il puntatore nullo ‘**`NULL`**’.	sezione 69.13
`stdint.h`	Definizioni di interi con un rango prestabilito, assieme ai valori minimi e massimi.	sezione 69.4
`stdio.h`	Gestione di input e output dei dati.	sezione 69.17
`stdlib.h`	Funzioni, macro e tipi di utilità generale.	sezione 69.9

Intestazione	Descrizione	Riferimenti
`string.h`	Gestione delle stringhe.	sezione 69.14
`tgmath.h`	macroistruzioni matematiche, indipendenti dal tipo.	--
`time.h`	Gestione di date e orari.	sezione 69.16
`wchar.h`	Gestione facilitata di caratteri estesi.	--
`wctype.h`	Classificazione dei caratteri estesi.	--

69.1 Funzionalità di libreria non dichiarate

Può succedere che il compilatore, per assolvere a funzionalità che figurano essere indipendenti da librerie, debba invece avvalersi di funzioni esterne che non sono previste dallo standard. In particolare, questo problema può verificarsi di fronte alla necessità di svolgere calcoli al di fuori della portata normale del microprocessore.

A titolo di esempio, il compilatore GNU C per la piattaforma x86-32 prevede un tipo intero '**`long long int`**' da 64 bit. Quando si vuole ottenere una divisione intera o il resto di una divisione con variabili di questo tipo, il compilatore GNU C richiama rispettivamente le funzioni ***__udivdi3()*** e ***__umoddi3()***. In generale il problema non si avverte, ma se si vuole scrivere la propria libreria C, senza tali funzioni, in pratica non è possibile usare questo tipo intero molto grande.

Si vedano eventualmente i listati della sezione 95.2, relativi a os32, in cui si realizzano queste funzioni con il solo ausilio del linguaggio C.

69.2 File «assert.h»

«

Il file ‘`assert.h`’ della libreria standard definisce la macroistruzione ***assert()***, da usare per generare informazioni diagnostiche, sulla base dell’esito della valutazione di un’espressione.

La macroistruzione ***assert()*** viene definita in due modi alternativi, in base alla presenza o meno della macro-variabile ***NDEBUG***. Per la precisione, in presenza della macro-variabile ***NDEBUG*** la macroistruzione ***assert()*** deve risultare inerte.

```
#include <stdio.h>
#ifdef NDEBUG
#define assert(ignore) ((void)0)
#else
#define assert(ASSERTION)                                    \
      ({if ((ASSERTION)==0)                                  \
        fprintf (stderr,                                     \
                 "Assertion failed: " # ASSERTION            \
                 ", function %s, file %s, line %u.\n", \
                 __func__, __FILE__, __LINE__);})
#endif
```

69.2.1 Utilizzo

«

La macroistruzione ***assert()*** va usata con la sintassi seguente, dove il parametro indica un’espressione di tipo non specificato, purché di tipo scalare:

```
void assert (espressione);
```

Se l’espressione si traduce in un valore *Falso*, ovvero pari a zero, la macroistruzione emette, attraverso lo standard error, un messaggio

contenente l'espressione stessa e altre indicazioni. Precisamente, oltre all'espressione deve apparire: il nome della funzione in cui ci si trova, il nome del file (sorgente) e il numero della riga.

Tuttavia, se la macro-variabile ***NDEBUG*** risulta definita, prima dell'inclusione del file '`assert.h`', la macroistruzione ***assert()*** deve essere trasformata dal compilatore come un'istruzione inerte, ovvero l'equivalente di '**`((void) 0)`**'.

Segue un l'esempio di un programma completo in cui si utilizza ***assert()***:

```
#include <stdio.h>
int
main (int argc, char *argv[])
{
    assert (123==124);
    return 0;
}
```

L'espressione verificata da ***assert()*** non può essere vera, pertanto, se non è stata dichiarata la macro-variabile ***NDEBUG***, questo programma dovrebbe produrre un messaggio come quello seguente:

```
Assertion failed: 123==124, function main, file assert.c,
line 5.
```

69.3 File «limits.h»

Il file '`limits.h`' della libreria standard definisce delle macro-variabili che riepilogano i limiti dei valori rappresentabili con le variabili scalari intere. Lo standard prescrive dei limiti minimi per la conformità, ma le realizzazioni comuni consentono mediamente di rappresentare valori più grandi (in senso assoluto), a parità di tipo

di intero. Infatti, i limiti effettivi dipendono principalmente dalla dimensione della parola del microprocessore e dal modo in cui si rappresentano i valori negativi. Si può osservare che nelle architetture comuni, in cui i valori negativi si rappresentano con il complemento a due, il valore negativo più grande (in senso assoluto) di una variabile è pari a una unità in più rispetto al valore positivo massimo (per esempio il tipo '**`signed char`**' va solitamente da −128 a 127).

L'esempio proposto si riferisce a un'architettura a 32 bit con i valori negativi rappresentati attraverso il complemento a due.

```
#define CHAR_UNSIGNED   0

#define CHAR_BIT        8

#define SCHAR_MIN       (-0x80)
#define SCHAR_MAX         0x7F
#define UCHAR_MAX         0xFF

#ifdef CHAR_UNSIGNED
#   define CHAR_MIN     0
#   define CHAR_MAX     UCHAR_MAX
#else
#   define CHAR_MIN     SCHAR_MIN
#   define CHAR_MAX     SCHAR_MAX
#endif

#define MB_LEN_MAX      16

#define SHRT_MIN        (-0x8000)
#define SHRT_MAX          0x7FFF
#define USHRT_MAX         0xFFFF

#define INT_MIN         (-0x80000000)
#define INT_MAX           0x7FFFFFFF
```

```
#define UINT_MAX          0xFFFFFFFFU

#define LONG_MIN        (-0x80000000L)
#define LONG_MAX          0x7FFFFFFFL
#define ULONG_MAX         0xFFFFFFFFUL

#define LLONG_MIN       (-0x8000000000000000LL)
#define LLONG_MAX         0x7FFFFFFFFFFFFFFFLL
#define ULLONG_MAX        0xFFFFFFFFFFFFFFFFULL
```

Tabella 69.7. Macro-variabili standard per la rappresentazione dei limiti riferiti a variabili scalari intere.

Macro-variabile	Descrizione
`CHAR_BIT`	Indica la quantità di bit utilizzata per rappresentare il tipo '**`char`**', con o senza segno. In altri termini è l'unità di memorizzazione più piccola con cui si può gestire l'insieme di caratteri minimo. Di norma si tratta di 8 bit.
`SCHAR_MIN` `SCHAR_MAX`	Indicano il valore minimo e il valore massimo rappresentabile in una variabile '**`signed char`**'.
`UCHAR_MAX`	Indica il valore massimo rappresentabile in una variabile '**`unsigned char`**'. Il valore minimo è zero.
`CHAR_MIN` `CHAR_MAX`	Indicano il valore minimo e il valore massimo rappresentabile in una variabile '**`char`**'. Questi valori dipendono dal fatto che il tipo '**`char`**' sia da intendere equivalente a un tipo '**`unsigned char`**' o '**`signed char`**', da cui ereditano i limiti.
`MB_LEN_MAX`	Indica la quantità massima di byte che possono essere usati per rappresentare un carattere multibyte, qualunque sia la configurazione locale.

Macro-variabile	Descrizione
`SHRT_MIN` `SHRT_MAX`	Indicano il valore minimo e il valore massimo rappresentabile in una variabile '**`short int`**'.
`USHRT_MAX`	Indica il valore massimo rappresentabile in una variabile '**`unsigned short int`**'. Il valore minimo è zero.
`INT_MIN` `INT_MAX`	Indicano il valore minimo e il valore massimo rappresentabile in una variabile '**`int`**'.
`UINT_MAX`	Indica il valore massimo rappresentabile in una variabile '**`unsigned int`**'. Il valore minimo è zero.
`LONG_MIN` `LONG_MAX`	Indicano il valore minimo e il valore massimo rappresentabile in una variabile '**`long int`**'.
`ULONG_MAX`	Indica il valore massimo rappresentabile in una variabile '**`unsigned long int`**'. Il valore minimo è zero.
`LLONG_MIN` `LLONG_MAX`	Indicano il valore minimo e il valore massimo rappresentabile in una variabile '**`long long int`**'.
`ULLONG_MAX`	Indica il valore massimo rappresentabile in una variabile '**`unsigned long long int`**'. Il valore minimo è zero.

Eventualmente si veda la realizzazione di questo file nei sorgenti di os32 (listato 95.1.6).

69.3.1 Confronto tra architetture

Per avere un'idea di come potrebbero svilupparsi i valori del file '`limits.h`' tra le varie architetture, viene mostrata una tabella in cui si possono paragonare quelli minimi stabiliti dallo standard con quelli usati nei sistemi GNU/Linux con architetture x86-32 e x86-64. Per semplicità si indicano solo i valori senza segno:

Macro-variabile	Standard	GNU/Linux x86-32	GNU/Linux x86-64
`UCHAR_MAX`	2^8-1	2^8-1	2^8-1
`USHRT_MAX`	$2^{16}-1$	$2^{16}-1$	$2^{16}-1$
`UINT_MAX`	$2^{16}-1$	$2^{32}-1$	$2^{32}-1$
`ULONG_MAX`	$2^{32}-1$	$2^{32}-1$	$2^{64}-1$
`ULLONG_MAX`	$2^{64}-1$	$2^{64}-1$	$2^{128}-1$

69.3.2 Estensioni POSIX

Per lo standard POSIX, il file '`limits.h`' serve anche per annotare limiti numerici relativi al funzionamento del sistema operativo, come per esempio la quantità massima di file aperti simultaneamente per ogni processo.

POSIX

Un gruppo di macro-variabili definite nel file '`limits.h`', caratterizzate per avere il prefisso ***_POSIX_***…, ***_POSIX2_***… e ***_XOPEN_***…, definisce dei limiti minimi di compatibilità con lo standard. Per esempio, la macro-variabile ***_POSIX_LINK_MAX*** deve tradursi nel numero 8 e stabilisce che deve essere consentita la creazione di al-

meno otto collegamenti fisici per ogni file, in qualunque sistema POSIX.

Un secondo gruppo di macro-variabili definite nel file '`limits.h`', dichiara i limiti massimi effettivi del sistema, riconducibili ai minimi già fissati nel primo gruppo già descritto. Per esempio, la macro-variabile ***LINK_MAX*** indica il numero massimo effettivo di collegamenti fisici per file, tenendo conto che deve essere necessariamente maggiore o uguale al valore di ***_POSIX_LINK_MAX***.

Le macro-variabili del secondo gruppo sono facoltative, in quanto i limiti effettivi del sistema, per le varie voci, possono dipendere da fattori dinamici di funzionamento. In ogni caso, devono essere garantiti i valori minimi delle macro-variabili del primo gruppo.

Nell'ambito delle dichiarazioni che fanno già parte dello standard C, va osservato che lo standard POSIX richiede che il byte sia esattamente di 8 bit, pertanto la macro-variabile ***CHAR_BIT*** deve tradursi necessariamente nel numero otto. Inoltre, si aggiungono anche qui alcune macro-variabili:

```
#define WORD_BIT   32
#define LONG_BIT   32
#define SSIZE_MAX  0x7FFFFFFFL
```

Tabella 69.10. Alcune macro-variabili aggiunte dallo standard POSIX.

Macro-variabile	Descrizione
`WORD_BIT`	Rappresenta la quantità di bit utilizzata per rappresentare il tipo '**`int`**', con o senza segno. Il minimo valore accettabile è 32.

Macro-variabile	Descrizione
`LONG_BIT`	Rappresenta la quantità di bit utilizzata per rappresentare il tipo '**`long int`**', con o senza segno. Il minimo valore accettabile è 32.
`SSIZE_MAX`	Rappresenta il valore positivo massimo che possa esprimere una variabile di tipo '**`ssize_t`**' (un tipo come '**`size_t`**', ma con segno). Il minimo valore accettabile è quello della macro-variabile ***_POSIX_SSIZE_MAX***, ovvero $7FFF_{16}$.

69.4 File «stdint.h»

Il file '`stdint.h`' della libreria standard definisce principalmente dei tipi interi, alternativi a quelli tradizionali, riferiti in modo più diretto al rango. Assieme a questi tipi interi definisce anche delle macro-variabili che consentono di conoscere i limiti esatti di tali tipi, oltre ad altre macro-variabili con i limiti di tipi interi speciali, dichiarati in altri file (si veda eventualmente la realizzazione di questo file nei sorgenti di os32, listato 95.1.13).

Lo standard prescrive che alcuni dei tipi definiti nel file '`stdint.h`' siano opzionali, purché sia rispettato un certo ordine (per esempio, se viene definito un tipo intero con segno, deve essere prevista anche una versione di quel tipo senza segno e viceversa). A tale proposito, le macro-variabili con le quali si possono verificare i limiti, servono anche per consentire al programmatore di verificare la disponibilità o meno del tipo relativo, attraverso istruzioni del precompilatore del tipo '**`#ifdef`**'.

L'esempio proposto si riferisce a un elaboratore x86-32 ed è abbastanza conforme alla configurazione che si può trovare in un sistema

GNU/Linux.

69.4.1 Tipi interi ad ampiezza esatta

«

Lo standard prescrive un gruppo facoltativo di tipi interi il cui rango è definito precisamente dal nome. Si tratta dei tipi '**int*n*_t**' (con segno) e '**uint*n*_t**' (senza segno), dove ***n*** esprime la quantità di bit che compone l'intero. Si tratta necessariamente di tipi facoltativi, perché non è possibile stabilire in modo sicuro che in ogni architettura siano gestibili tipi interi di una data quantità di bit; per esempio, in una certa achitettura «X» potrebbero essere gestiti tipi interi a 8, 16 e 32 bit, mentre in un'architettura «Y» i tipi disponibili effettivamente potrebbero essere a 8, 16, 24 e 32 bit.

Nei sistemi POSIX, questi tipi sono invece obbligatori, costringendo così ad avere byte esattamente di otto bit.

POSIX

```
typedef signed char                int8_t;
typedef short int                  int16_t;
typedef int                        int32_t;          // x86-32
typedef long long int              int64_t;          // x86-32

typedef unsigned char              uint8_t;
typedef unsigned short int         uint16_t;
typedef unsigned int               uint32_t;         // x86-32
typedef unsigned long long int     uint64_t;         // x86-32
```

Le macro-variabili usate per definire i limiti di questi valori interi hanno nomi del tipo '**INT*n*_MIN**', '**INT*n*_MAX**' e '**UINT*n*_MAX**', per indicare rispettivamente: il valore minimo dei tipi con segno; il valore massimo dei tipi con segno; il valore massimo dei tipi senza segno.

Lo standard prescrive precisamente questi valori minimi e massimi, intendendo implicitamente che i valori negativi si rappresentino con il complemento a due:

Macro-variabile	Valore standard
`INT`***n***`_MIN`	$-(2^{n-1})$
`INT`***n***`_MAX`	$2^{n-1}-1$
`UINT`***n***`_MAX`	$2^{n}-1$

```
#define INT8_MIN                (-0x80)
#define INT16_MIN               (-0x8000)
#define INT32_MIN               (-0x80000000)
#define INT64_MIN               (-0x8000000000000000LL)

#define INT8_MAX                  0x7F
#define INT16_MAX                 0x7FFF
#define INT32_MAX                 0x7FFFFFFF
#define INT64_MAX                 0x7FFFFFFFFFFFFFFFLL

#define UINT8_MAX                 0xFF
#define UINT16_MAX                0xFFFF
#define UINT32_MAX                0xFFFFFFFFU
#define UINT64_MAX                0xFFFFFFFFFFFFFFFFULL
```

69.4.2 Tipi interi di rango minimo

Un gruppo richiesto espressamente dallo standard riguarda tipi interi il cui rango sia tale da garantire la rappresentazione di almeno ***n*** bit, utilizzando comunque la quantità minima possibile di bit. In questo caso i nomi sono '**`int_least`*n*`_t`**' per i tipi con segno

e '**uint_least*n*_t**' per quelli senza segno. Lo standard prescrive che siano previsti necessariamente i tipi a 8, 16, 32 e 64 bit, mentre ammette che ne siano disponibili anche altri.

```
typedef signed char                int_least8_t;
typedef short int                  int_least16_t;
typedef int                        int_least32_t;   // x86-32
typedef long long int              int_least64_t;   // x86-32

typedef unsigned char              uint_least8_t;
typedef unsigned short int         uint_least16_t;
typedef unsigned int               uint_least32_t;  // x86-32
typedef unsigned long long int     uint_least64_t;  // x86-32
```

Le macro-variabili usate per definire i limiti di questi valori interi hanno nomi del tipo '**INT_LEAST*n*_MIN**', '**INT_LEAST*n*_MAX**' e '**UINT_LEAST*n*_MAX**', per indicare rispettivamente: il valore minimo dei tipi con segno; il valore massimo dei tipi con segno; il valore massimo dei tipi senza segno. Lo standard attribuisce questi limiti in modo indipendente dalla rappresentazione dei valori negativi, ma tali limiti possono essere estesi in senso assoluto:

Macro-variabile	Valore standard che può essere superato in termini assoluti
INT_LEAST*n*_MIN	$-(2^{n-1}-1)$
INT_LEAST*n*_MAX	$2^{n-1}-1$
UINT_LEAST*n*_MAX	$2^{n}-1$

```
#define INT_LEAST8_MIN          (-0x80)
#define INT_LEAST16_MIN         (-0x8000)
#define INT_LEAST32_MIN         (-0x80000000)
#define INT_LEAST64_MIN         (-0x8000000000000000LL)
```

```
#define INT_LEAST8_MAX              0x7F
#define INT_LEAST16_MAX             0x7FFF
#define INT_LEAST32_MAX             0x7FFFFFFF
#define INT_LEAST64_MAX             0x7FFFFFFFFFFFFFFFLL

#define UINT_LEAST8_MAX             0xFF
#define UINT_LEAST16_MAX            0xFFFF
#define UINT_LEAST32_MAX            0xFFFFFFFFU
#define UINT_LEAST64_MAX            0xFFFFFFFFFFFFFFFFULL
```

Ai tipi interi di rango minimo sono associate anche delle macroistruzioni, il cui scopo è quello di consentire la rappresentazione corretta dei valori costanti:

```
INTn_C(valore)
```

```
UINTn_C(valore)
```

In pratica, per indicare il valore costante 1234567890, precisando che va inteso come un tipo '**`uint_least64_t`**', si deve scrivere: '**`UINT64_C(1234567890)`**'. Il valore costante in sé, può essere espresso in qualunque modo, purché sia ammissibile nel contesto comune.

```
#define INT8_C(VAL)      VAL
#define INT16_C(VAL)     VAL
#define INT32_C(VAL)     VAL                 // x86-32
#define INT64_C(VAL)     VAL ## LL           // x86-32

#define UINT8_C(VAL)     VAL
#define UINT16_C(VAL)    VAL
#define UINT32_C(VAL)    VAL ## U            // x86-32
#define UINT64_C(VAL)    VAL ## ULL          // x86-32
```

È evidente che, nel caso dell'esempio mostrato, '**`UINT64_C(1234567890)`**' corrisponde a '**`1234567890ULL`**'.

69.4.3 Tipi interi «veloci»

«

Un altro gruppo di tipi richiesti dallo standard è quello il cui rango è tale da consentire la rappresentazione di almeno ***n*** bit, utilizzando la quantità minima di bit che garantisce tempi ottimali di elaborazione. In questo caso i nomi sono '**`int_fast`*n*`_t`**' per i tipi con segno e '**`uint_fast`*n*`_t`**' per quelli senza segno. Lo standard prescrive che siano previsti necessariamente i tipi a 8, 16, 32 e 64 bit, mentre ammette che ne siano disponibili anche altri.

```
typedef signed char               int_fast8_t;
typedef int                       int_fast16_t;   // x86-32
typedef int                       int_fast32_t;   // x86-32
typedef long long int             int_fast64_t;   // x86-32

typedef unsigned char             uint_fast8_t;
typedef unsigned int              uint_fast16_t;  // x86-32
typedef unsigned int              uint_fast32_t;  // x86-32
typedef unsigned long long int    uint_fast64_t;  // x86-32
```

Come suggerisce l'esempio, è ragionevole pensare che, dove possibile, il rango usato effettivamente sia quello del tipo intero

normale.

Le macro-variabili usate per definire i limiti di questi valori interi hanno nomi del tipo '**INT_FAST*n*_MIN**', '**INT_FAST*n*_MAX**' e '**UINT_FAST*n*_MAX**', per indicare rispettivamente: il valore minimo dei tipi con segno; il valore massimo dei tipi con segno; il valore massimo dei tipi senza segno. Lo standard attribuisce questi limiti in modo indipendente dalla rappresentazione dei valori negativi, ma tali limiti possono essere estesi in senso assoluto:

Macro-variabile	Valore standard che può essere superato in termini assoluti
`INT_FAST`***n***`_MIN`	$-(2^{n-1}-1)$
`INT_FAST`***n***`_MAX`	$2^{n-1}-1$
`UINT_FAST`***n***`_MAX`	$2^{n}-1$

```
#define INT_FAST8_MIN          (-0x80)
#define INT_FAST16_MIN         (-0x80000000)
#define INT_FAST32_MIN         (-0x80000000)
#define INT_FAST64_MIN         (-0x8000000000000000LL)

#define INT_FAST8_MAX            0x7F
#define INT_FAST16_MAX           0x7FFFFFFF
#define INT_FAST32_MAX           0x7FFFFFFF
#define INT_FAST64_MAX           0x7FFFFFFFFFFFFFFFLL

#define UINT_FAST8_MAX           0xFF
#define UINT_FAST16_MAX          0xFFFFFFFFU
#define UINT_FAST32_MAX          0xFFFFFFFFU
#define UINT_FAST64_MAX          0xFFFFFFFFFFFFFFFFULL
```

69.4.4 Tipi interi per rappresentare dei puntatori

«

Sono previsti due tipi opzionali interi, adatti a contenere il valore di un puntatore, garantendo che la conversione da e verso '**`void *`**' avvenga sempre correttamente. Per la precisione si tratta di '**`intptr_t`**' e '**`uintptr_t`**', dove il primo è un intero con segno, mentre il secondo è senza segno.

```
typedef int                 intptr_t;       // x86-32
typedef unsigned int        uintptr_t;      // x86-32
```

Le macro-variabili usate per definire i limiti di questi valori interi sono ***INTPTR_MIN***, ***INTPTR_MAX*** e ***UINTPTR_MAX***, per indicare rispettivamente: il valore minimo con segno, il valore massimo con segno e il valore massimo senza segno. Lo standard attribuisce dei limiti riferiti ad architetture in grado di indirizzare al massimo con 16 bit e ovviamente vanno adattati alla realtà dell'architettura effettiva:

Macro-variabile	Valore standard che può essere superato in termini assoluti
`INTPTR_MIN`	$-(2^{15}-1)$
`INTPTR_MAX`	$2^{15}-1$
`UINTPTR_MAX`	$2^{16}-1$

```
#define INTPTR_MIN          (-0x80000000)
#define INTPTR_MAX            0x7FFFFFFF
#define UINTPTR_MAX           0xFFFFFFFFU
```

69.4.5 Tipi interi di rango massimo

Per poter rappresentare in modo indipendente dall'architettura degli interi di rango massimo, sono previsti due tipi specifici, richiesti espressamente dallo standard: '**intmax_t**' e '**uintmax_t**'. Le macro-variabili che definiscono i limiti sono: ***INTMAX_MIN***, ***INTMAX_MAX*** e ***UINTMAX_MAX***. Lo standard prescrive che si tratti di variabili con un rango di almeno 64 bit.

```
typedef long long int           intmax_t;     // x86-32
typedef unsigned long long int  uintmax_t;    // x86-32
```

Macro-variabile	Valore standard che può essere superato in termini assoluti
`INTMAX_MIN`	$-(2^{63}-1)$
`INTMAX_MAX`	$2^{63}-1$
`UINTMAX_MAX`	$2^{64}-1$

Anche ai tipi interi di rango massimo sono associate delle macroistruzioni per facilitare la rappresentazione corretta dei valori costanti:

```
INTMAX_C(valore)
```

```
UINTMAX_C(valore)
```

In pratica, per indicare il valore costante 1234567890, precisando che va inteso come un tipo '**uintmax_t**', si deve scrivere:

‘**`UINTMAX_C(1234567890)`**’. Il valore costante in sé, può essere espresso in qualunque modo, purché sia ammissibile nel contesto comune.

```
#define INTMAX_C(VAL)   VAL ## LL       // x86-32
#define UINTMAX_C(VAL)  VAL ## ULL      // x86-32
```

A questo punto, anche la definizione dei valori minimi e massimi diventa più agevole:

```
// x86-32
#define INTMAX_MIN      (-INTMAX_C(0x8000000000000000))
#define INTMAX_MAX       (INTMAX_C(0x7FFFFFFFFFFFFFFF))
#define UINTMAX_MAX     (UINTMAX_C(0xFFFFFFFFFFFFFFFF))
```

69.4.6 Limiti per altri tipi interi

«

Altri tipi interi dichiarati al di fuori del file ‘`stdint.h`’ hanno i limiti definiti qui. Ne viene mostrata solo una tabella riepilogativa.

Macro-variabile	Descrizione
`PTRDIFF_MIN` `PTRDIFF_MAX`	Indicano il valore minimo e il valore massimo rappresentabile in una variabile ‘**`ptrdiff_t`**’.
`SIG_ATOMIC_MIN` `SIG_ATOMIC_MAX`	Indicano il valore minimo e il valore massimo rappresentabile in una variabile ‘**`sig_atomic_t`**’.
`SIZE_MAX`	Indica il valore massimo rappresentabile in una variabile ‘**`size_t`**’, la quale è destinata a contenere valori senza segno.
`WCHAR_MIN` `WCHAR_MAX`	Indicano il valore minimo e il valore massimo rappresentabile in una variabile ‘**`wchar_t`**’.

Macro-variabile	Descrizione
`WINT_MIN` `WINT_MAX`	Indicano il valore minimo e il valore massimo rappresentabile in una variabile '**`wint_t`**'.

L'esempio seguente riporta i valori usati in un sistema GNU/Linux con architettura x86-32, a parte il caso di '**`wchar_t`**' e '**`wint_t`**' che si intendono rispettivamente a 32 bit senza segno e 64 bit con segno:

```
#define PTRDIFF_MIN                (-0x80000000)    // x86-32
#define PTRDIFF_MAX                  0x7FFFFFFF     // x86-32

#define SIG_ATOMIC_MIN             (-0x80000000)    // x86-32
#define SIG_ATOMIC_MAX               0x7FFFFFFF     // x86-32

#define SIZE_MAX                     0xFFFFFFFFU    // x86-32

#define WCHAR_MIN                    0x00000000
#define WCHAR_MAX                    0xFFFFFFFFU    // x86-32

#define WINT_MIN          (-0x8000000000000000LL)   // x86-32
#define WINT_MAX            0x7FFFFFFFFFFFFFFFLL    // x86-32
```

69.5 File «errno.h»

Il file '`errno.h`' della libreria standard definisce principalmente delle macro-variabili per rappresentare simbolicamente delle situazioni di errore. Queste macro-variabili si espandono in un numero intero e positivo, di tipo '**`int`**', ma la corrispondenza tra l'errore simbolico rappresentato dalla macro-variabile e il numero in cui questa si deve espandere, dipende dalle convenzioni del sistema operativo (si veda eventualmente la realizzazione del file '`errno.h`' nei sorgenti

di os32, sezione 95.5, tenendo conto che lì si aggiungono delle funzioni non standard, con le quali è più facile l'individuazione della posizione del sorgente in cui l'errore si è manifestato).

Lo standard del linguaggio prescrive poche macro-variabili, da cui dipendono le librerie standard, mentre tutte le altre sono competenza delle convenzioni del sistema operativo.

Oltre alle macro-variabili che rappresentano le situazioni di errore previste, il file '`errno.h`' deve dichiarare ***errno***, in qualità di espressione che si traduca in una variabile scalare. In pratica può trattarsi di una variabile esterna o di una macro-variabile che si traduce in qualunque cosa consenta di assegnarvi un valore. Il valore iniziale che si può leggere da ***errno*** è zero (con cui si intende l'assenza di qualunque tipo di situazione di errore) e viene modificato dalle funzioni che, di volta in volta, possono avere bisogno di annotare uno stato di errore.

Si osservi che i numeri che si vedono associati alle macro-variabili sono stati tratti, come esempio, dalla configurazione di un sistema GNU/Linux.

```
extern int errno;

#define EDOM    33
#define EILSEQ  84
#define ERANGE  34
```

Il nome ***errno***, in un sistema che consenta l'esecuzione di programmi suddivisi in più thread, deve tradursi in un'espressione tale da rappresentare una variabile scalare individuale per ogni thread, in modo che i thread non possano interferire tra di loro a questo proposito. Evidentemente, l'esempio mostrato non offre questa

accortezza.

Tabella 69.31. Macro-variabili standard per la rappresentazione di situazioni di errore.

Macro-variabile	Descrizione
`EDOM`	Errore di dominio: l'argomento di una funzione matematica ha un valore al di fuori del dominio previsto.
`EILSEQ`	Errore di codifica: la sequenza dei byte che deve rappresentare una certa codifica contiene un errore.
`ERANGE`	Errore nell'intervallo di valori: il risultato di un'espressione matematica non può essere rappresentato nell'intervallo di valori previsto (ovvero nella variabile che deve riceverlo).

Il file di intestazione '`errno.h`' di un sistema POSIX è più articolato, in quanto contiene un elenco numeroso di macro-variabili. Valgono naturalmente le stesse considerazioni per la variabile globale ***errno***, a proposito dei thread multipli. Nell'esempio successivo, i numeri associati alle varie macro-variabili non fanno riferimento ad alcun sistema operativo reale; va osservato inoltre che ogni sistema POSIX aggiunge propri tipi di errore, necessari per le proprie caratteristiche specifiche.

POSIX

```
extern int errno;

#define E2BIG           1 // Argument list too long.
#define EACCES          2 // Permission denied.
#define EADDRINUSE      3 // Address in use.
#define EADDRNOTAVAIL   4 // Address not available.
#define EAFNOSUPPORT    5 // Address family not supported.
#define EAGAIN          6 // Resource unavailable, try
                          // again.
#define EALREADY        7 // Connection already in
```

```
                                // progress.
#define EBADF            8      // Bad file descriptor.
#define EBADMSG          9      // Bad message.
#define EBUSY           10      // Device or resource busy.
#define ECANCELED       11      // Operation canceled.
#define ECHILD          12      // No child processes.
#define ECONNABORTED    13      // Connection aborted.
#define ECONNREFUSED    14      // Connection refused.
#define ECONNRESET      15      // Connection reset.
#define EDEADLK         16      // Resource deadlock would occur.
#define EDESTADDRREQ    17      // Destination address required.
#define EDOM            18      // Mathematics argument out of
                                // domain of function.
#define EDQUOT          19      // Reserved.
#define EEXIST          20      // File exists.
#define EFAULT          21      // Bad address.
#define EFBIG           22      // File too large.
#define EHOSTUNREACH    23      // Host is unreachable.
#define EIDRM           24      // Identifier removed.
#define EILSEQ          25      // Illegal byte sequence.
#define EINPROGRESS     26      // Operation in progress.
#define EINTR           27      // Interrupted function.
#define EINVAL          28      // Invalid argument.
#define EIO             29      // I/O error.
#define EISCONN         30      // Socket is connected.
#define EISDIR          31      // Is a directory.
#define ELOOP           32      // Too many levels of symbolic
                                // links.
#define EMFILE          33      // Too many open files.
#define EMLINK          34      // Too many links.
#define EMSGSIZE        35      // Message too large.
#define EMULTIHOP       36      // Reserved.
#define ENAMETOOLONG    37      // Filename too long.
#define ENETDOWN        38      // Network is down.
```

```
#define ENETRESET       39 // Connection aborted by network.
#define ENETUNREACH     40 // Network unreachable.
#define ENFILE          41 // Too many files open in system.
#define ENOBUFS         42 // No buffer space available.
#define ENODATA         43 // No message is available on the
                           // stream head read queue.
#define ENODEV          44 // No such device.
#define ENOENT          45 // No such file or directory.
#define ENOEXEC         46 // Executable file format error.
#define ENOLCK          47 // No locks available.
#define ENOLINK         48 // Reserved.
#define ENOMEM          49 // Not enough space.
#define ENOMSG          50 // No message of the desired
                           // type.
#define ENOPROTOOPT     51 // Protocol not available.
#define ENOSPC          52 // No space left on device.
#define ENOSR           53 // No stream resources.
#define ENOSTR          54 // Not a stream.
#define ENOSYS          55 // Function not supported.
#define ENOTCONN        56 // The socket is not connected.
#define ENOTDIR         57 // Not a directory.
#define ENOTEMPTY       58 // Directory not empty.
#define ENOTSOCK        59 // Not a socket.
#define ENOTSUP         60 // Not supported.
#define ENOTTY          61 // Inappropriate I/O control
                           // operation.
#define ENXIO           62 // No such device or address.
#define EOPNOTSUPP      63 // Operation not supported on
                           // socket.
#define EOVERFLOW       64 // Value too large to be stored
                           // in data type.
#define EPERM           65 // Operation not permitted.
#define EPIPE           66 // Broken pipe.
#define EPROTO          67 // Protocol error.
```

```
#define EPROTONOSUPPORT 68 // Protocol not supported.
#define EPROTOTYPE      69 // Protocol wrong type for
                           // socket.
#define ERANGE          70 // Result too large.
#define EROFS           71 // Read-only file system.
#define ESPIPE          72 // Invalid seek.
#define ESRCH           73 // No such process.
#define ESTALE          74 // Reserved.
#define ETIME           75 // Stream ioctl() timeout.
#define ETIMEDOUT       76 // Connection timed out.
#define ETXTBSY         77 // Text file busy.
#define EWOULDBLOCK     78 // Operation would block (may
                           // be the same as EAGAIN).
#define EXDEV           79 // Cross-device link.
```

69.6 File «locale.h»

«

Il file ‘`locale.h`’ della libreria standard definisce delle macro-variabili, un tipo di struttura e alcune funzioni, relative alla gestione della configurazione locale del programma. Se non si fa uso di tale configurazione, il proprio programma opera in quella che è nota come «configurazione locale C», ovvero il minimo indispensabile.

Nell’ambito di un sistema operativo Unix o simile, la configurazione locale avviene principalmente attraverso l’impostazione di variabili di ambiente il cui nome inizia per ‘**`LC_`**...’. Tuttavia, questa configurazione non viene ereditata automaticamente dal programma scritto in linguaggio C, perché questo deve acquisirla espressamente, se vuole.

La definizione della configurazione locale avviene attraverso una stringa contenente delle sigle che esprimono la lingua, la nazionalità e la codifica da utilizzare per rappresentare i caratteri. Per esem-

pio, '**it_CH.UTF-8**' rappresenta la lingua italiana, la nazionalità svizzera e la codifica UTF-8.

69.6.1 Impostazione della configurazione locale

Per modificare l'impostazione della configurazione locale del proprio programma, si utilizza la funzione ***setlocale()*** che richiede l'indicazione di un numero intero, a rappresentare la ***categoria*** nella quale intervenire. La categoria viene definita formalmente attraverso delle macro-variabili il cui nome inizia per ***LC_***... e si tratta degli stessi nomi usati nelle variabili di ambiente di un sistema Unix o simile. Viene proposto un esempio di dichiarazione delle macro-variabili indispensabili, ma l'associazione al numero varia molto da un sistema all'altro:

```
LC_ALL          0
LC_COLLATE      1
LC_CTYPE        2
LC_MONETARY     3
LC_NUMERIC      4
LC_TIME         5
```

La macro variabile successiva è un'estensione usata nei sistemi POSIX:

POSIX

```
LC_MESSAGES     6
```

La funzione ***setlocale()*** con cui si cambia la configurazione locale ha il prototipo seguente:

```
char *setlocale (int category, const char *locale);
```

Si prevedono due situazioni diverse di utilizzo della funzione. Per cominciare può essere usata per interrogare la configurazione attuale, come nell'esempio seguente:

```
…
char *p;
p = setlocale (LC_COLLATE, NULL);
…
```

Fornendo un puntatore nullo, al posto della stringa che deve indicare la configurazione locale, si ottiene un puntatore alla stringa che descrive quella attuale. In questo caso, se si utilizza la categoria '**LC_ALL**', si ottiene una stringa che descrive tutte le altre categorie, ammesso che ci siano delle differenze. Se invece la funzione non è in grado di dare questa informazione, si ottiene semplicemente un puntatore nullo.

Naturalmente, la funzione serve anche per cambiare la configurazione locale, specificando in tal caso la stringa che la descrive. Per esempio, nel modo seguente si interviene nella categoria '**LC_NUMERIC**':

```
…
char *p;
p = setlocale (LC_NUMERIC, "it_IT.UTF-8");
…
```

Anche in questo caso si ottiene un puntatore che descrive la categoria scelta, ma se l'operazione fallisce, si ottiene invece il puntatore nullo.

Normalmente è più probabile che, nell'impostazione della configurazione locale si voglia indicare una modalità unica per tutte le categorie; pertanto, in tal caso va usato '**LC_ALL**':

```
…
p = setlocale (LC_ALL, "it_IT.UTF-8");
…
```

Viene mostrato un esempio di programma completo, in cui si imposta prima la configurazione locale complessiva, poi se ne cambia una e quindi si interroga la situazione:

```
#include <stdio.h>
#include <locale.h>

int main (void)
{
    setlocale (LC_ALL, "it_IT.UTF-8");
    setlocale (LC_MONETARY, "en_US.UTF-8");

    printf ("LC_COLLATE:  \"%s\"\n",
            setlocale (LC_COLLATE, NULL));
    printf ("LC_CTYPE:    \"%s\"\n",
            setlocale (LC_CTYPE, NULL));
    printf ("LC_MONETARY: \"%s\"\n",
            setlocale (LC_MONETARY, NULL));
    printf ("LC_NUMERIC:  \"%s\"\n",
            setlocale (LC_NUMERIC, NULL));
    printf ("LC_TIME:     \"%s\"\n",
            setlocale (LC_TIME, NULL));
    printf ("LC_ALL:      \"%s\"\n",
            setlocale (LC_ALL, NULL));
#ifdef LC_MESSAGES
    printf ("LC_MESSAGES: \"%s\"\n",
            setlocale (LC_MESSAGES, NULL));
#endif

    return 0;
}
```

Ecco cosa si può ottenere:

```
LC_COLLATE:  "it_IT.UTF-8"
LC_CTYPE:    "it_IT.UTF-8"
LC_MONETARY: "en_US.UTF-8"
LC_NUMERIC:  "it_IT.UTF-8"
LC_TIME:     "it_IT.UTF-8"
LC_ALL:      "LC_CTYPE=it_IT.UTF-8;LC_NUMERIC=it_IT.UTF-8;↩
↪LC_TIME=it_IT.UTF-8;LC_COLLATE=it_IT.UTF-8;LC_MONETARY=en_US.UTF
↪LC_MESSAGES=it_IT.UTF-8"
LC_MESSAGES: "it_IT.UTF-8"
```

Per fare sì che il programma erediti la configurazione locale dal contesto in cui si trova a funzionare (quindi dalla configurazione locale del sistema operativo), si può indicare la stringa nulla al posto della definizione:

```
…
p = setlocale (LC_ALL, "");
…
```

Pertanto, questo è il modo appropriato per iniziare la configurazione all'interno di un programma.

69.6.2 Composizione dei valori numerici

Il modo in cui si rappresenta testualmente un valore numerico, con o senza indicazione della valuta (la moneta), dipende dalla configurazione locale. La funzione ***localeconv()*** restituisce il puntatore a una struttura che contiene i dettagli riguardo alle modalità di rappresentazione dei valori numerici, secondo la configurazione locale. L'utilizzo consentito di questa struttura si limita all'interrogazione dei valori, perché la modifica dipende dalla gestione della configurazione locale.

```
struct lconv {char *decimal_point;
              char *thousands_sep;
              char *grouping;
              char *mon_decimal_point;
              char *mon_thousands_sep;
              char *mon_grouping;
              char *positive_sign;
              char *negative_sign;
              char *currency_symbol;
              char frac_digits;
              char p_cs_precedes;
              char n_cs_precedes;
              char p_sep_by_space;
              char n_sep_by_space;
              char p_sign_posn;
              char n_sign_posn;
              char *int_curr_symbol;
              char int_frac_digits;
              char int_p_cs_precedes;
              char int_n_cs_precedes;
              char int_p_sep_by_space;
              char int_n_sep_by_space;
              char int_p_sign_posn;
              char int_n_sign_posn;
};
```

A titolo di esempio vengono descritti solo alcuni membri della struttura; per gli altri si deve consultare la documentazione dello standard:

Membro	Descrizione
`decimal_point`	Stringa contenente il carattere usato per separare la parte decimale in un numero per uso generale.

Membro	Descrizione
`thousand_sep`	Stringa contenente il carattere usato per separare le cifre della parte intera di un numero per uso generale.
`positive_sign` `negative_sign`	Stringa contenente il carattere usato per rappresentare il segno, positivo o negativo, di un numero usato per le valute.
`int_curr_symbol`	Stringa composta da quattro caratteri, di cui i primi tre indicano la sigla internazionale della valuta (‘**`USD`**’, ‘**`EUR`**’, ecc.) e il quarto è solo un carattere di separazione da usare tra tale sigla e il valore numerico a cui questa si riferisce.

I membri che rappresentano delle stringhe (puntatori a carattere), quando si riferiscono a dati facoltativi, possono essere vuoti (nel senso di stringhe nulle). I membri di tipo ‘**`char`**’ vengono usati in modo numerico.

```
struct lconv *localeconv (void);
```

Come si vede dal prototipo, la funzione ***localeconv()*** serve esclusivamente per ottenere il puntatore alla struttura ‘**`lconv`**’, da usare per la sua consultazione. Viene mostrato un esempio molto semplice per il suo utilizzo:

```
#include <stdio.h>
#include <locale.h>

int main (void)
{
    struct lconv *lc;
```

```
    setlocale (LC_ALL, "it_IT.UTF-8");
    lc = localeconv ();

    printf ("decimal_point:\t\t\"%s\"\n",
            lc->decimal_point);
    printf ("thousand_sep:\t\t\"%s\"\n",
            lc->thousands_sep);
    printf ("grouping:\t\t\"%s\"\n",
            lc->grouping);
    printf ("mon_decimal_point:\t\"%s\"\n",
            lc->mon_decimal_point);
    printf ("mon_thousands_sep:\t\"%s\"\n",
            lc->mon_thousands_sep);
    printf ("mon_grouping:\t\t\"%s\"\n",
            lc->mon_grouping);
    printf ("positive_sign:\t\t\"%s\"\n",
            lc->positive_sign);
    printf ("negative_sign:\t\t\"%s\"\n",
            lc->negative_sign);
    printf ("currency_symbol:\t\"%s\"\n",
            lc->currency_symbol);                  // Multibyte.
    printf ("frac_digits:\t\t%i\n",
            lc->frac_digits);
    printf ("p_cs_precedes:\t\t%i\n",
            lc->p_cs_precedes);
    printf ("n_cs_precedes:\t\t%i\n",
            lc->n_cs_precedes);
    printf ("p_sep_by_space:\t\t%i\n",
            lc->p_sep_by_space);
    printf ("n_sep_by_space:\t\t%i\n",
            lc->n_sep_by_space);
    printf ("p_sign_posn:\t\t%i\n",
            lc->p_sign_posn);
    printf ("n_sign_posn:\t\t%i\n",
```

```
            lc->n_sign_posn);
    printf ("int_curr_symbol:\t\"%s\"\n",
            lc->int_curr_symbol);
    printf ("int_frac_digit:\t\t%i\n",
            lc->int_frac_digit);
    printf ("int_p_cs_precedes:\t\t%i\n",
            lc->int_p_cs_precedes);
    printf ("int_n_cs_precedes:\t\t%i\n",
            lc->int_n_cs_precedes);
    printf ("int_p_sep_by_space:\t\t%i\n",
            lc->int_p_sep_by_space);
    printf ("int_n_sep_by_space:\t\t%i\n",
            lc->int_n_sep_by_space);
    printf ("int_p_sign_posn:\t\t%i\n",
            lc->int_p_sign_posn);
    printf ("int_n_sign_posn:\t\t%i\n",
            lc->int_n_sign_posn);

    return 0;
}
```

Il risultato che si ottiene dovrebbe essere molto simile a quello seguente:

```
decimal_point:          ","
thousand_sep:           ""
grouping:               ""
mon_decimal_point:      ","
mon_thousands_sep:      "."
mon_grouping:           ""
positive_sign:          ""
negative_sign:          "-"
currency_symbol:        "€"
frac_digits:            2
```

```
p_cs_precedes:          1
n_cs_precedes:          1
p_sep_by_space:         1
n_sep_by_space:         1
p_sign_posn:            1
n_sign_posn:            1
int_curr_symbol:        "EUR "
int_frac_digit:         2
int_p_cs_precedes:      1
int_n_cs_precedes:      1
int_p_sep_by_space:     1
int_n_sep_by_space:     1
int_p_sign_posn:        1
int_n_sign_posn:        4
```

Si osservi che, nell'esempio, la stringa a cui si accede tramite il membro '**currency_symbol**' è una sequenza «multibyte», nel senso che utilizza più byte per rappresentare un solo carattere.

69.7 File «ctype.h»

Il file '`ctype.h`' della libreria standard definisce alcune funzioni per la classificazione e la trasformazione dei caratteri (intesi come '**char**'). Gli esempi proposti qui riguardano esclusivamente l'insieme di caratteri corrispondente alla codifica ASCII e, di conseguenza, la configurazione locale '**C**'. Tuttavia va ricordato che il linguaggio C non impone che l'insieme di caratteri minimo sia descritto attraverso la codifica ASCII, mentre così è invece nello standard POSIX. (si veda eventualmente la realizzazione del file '`ctype.h`' nei sorgenti di os32, listato 95.1.5).

Le funzioni di questo file hanno in comune il parametro, costituito da un valore intero di tipo '**int**', usato per rappresentare il caratte-

re.[1] Le funzioni del tipo ***is**...()* restituiscono un valore intero diverso da zero (corrispondente a *Vero*) se la condizione riferita al carattere fornito si verifica. Le funzioni ***to**...()* restituiscono un valore intero, corrispondente al carattere fornito e trasformato nel modo richiesto, se ciò è possibile.

Listato 69.47. Prototipi delle funzioni dichiarate nel file '`ctype.h`'.

```
int isalnum  (int c);
int isalpha  (int c);
int isblank  (int c);
int iscntrl  (int c);
int isdigit  (int c);
int isgraph  (int c);
int islower  (int c);
int isprint  (int c);
int ispunct  (int c);
int isspace  (int c);
int isupper  (int c);
int isxdigit (int c);
int tolower  (int c);
int toupper  (int c);
```

Listato 69.48. Prototipi aggiuntivi dello standard POSIX.

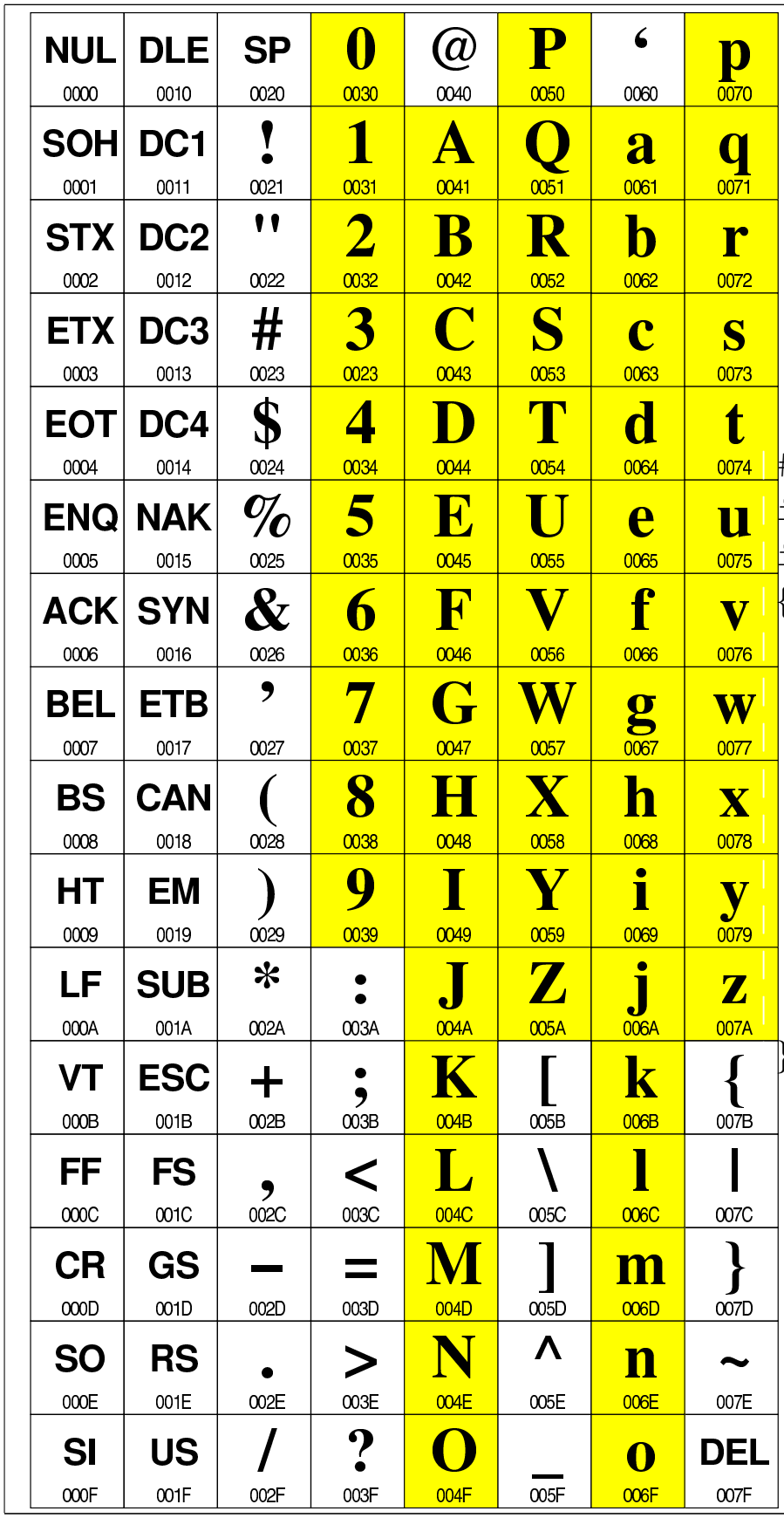
```
int isascii  (int c);           // POSIX
int toascii  (int c);           // POSIX
```

69.7.1 Funzioni «is...()»

«

Il gruppo di funzioni ***is**...()* restituisce un valore intero diverso da zero (corrispondente a *Vero*) se la condizione riferita al carattere fornito si verifica. Vengono proposte le varie soluzioni, affiancando la tabella ASCII con i caratteri validi evidenziati.

Listato 69.49. Funzione *isalnum()*.




```
#include <ctype.h>
int
isalnum (int c)
{
  if (isalpha (c)
      || isdigit (c))
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

Listato 69.50. Funzione ***isalpha()***.

NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	‘ 0060	p 0070
SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
ETX 0003	DC3 0013	# 0023	3 0023	C 0043	S 0053	c 0063	s 0073
EOT 0004	DC4 0014	$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074
ENQ 0005	NAK 0015	% 0025	5 0035	E 0045	U 0055	e 0065	u 0075
ACK 0006	SYN 0016	& 0026	6 0036	F 0046	V 0056	f 0066	v 0076
BEL 0007	ETB 0017	’ 0027	7 0037	G 0047	W 0057	g 0067	w 0077
BS 0008	CAN 0018	(0028	8 0038	H 0048	X 0058	h 0068	x 0078
HT 0009	EM 0019	) 0029	9 0039	I 0049	Y 0059	i 0069	y 0079
LF 000A	SUB 001A	* 002A	: 003A	J 004A	Z 005A	j 006A	z 007A
VT 000B	ESC 001B	+ 002B	; 003B	K 004B	[005B	k 006B	{ 007B
FF 000C	FS 001C	, 002C	< 003C	L 004C	\ 005C	l 006C	| 007C
CR 000D	GS 001D	– 002D	= 003D	M 004D	] 005D	m 006D	} 007D
SO 000E	RS 001E	. 002E	> 003E	N 004E	^ 005E	n 006E	~ 007E
SI 000F	US 001F	/ 002F	? 003F	O 004F	_ 005F	o 006F	DEL 007F

```
int
isalpha (int c)
{
  if (isupper (c)
      || islower (c))
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

Listato 69.51. Funzione *isblank()*.

NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	‘ 0060	p 0070
SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
ETX 0003	DC3 0013	# 0023	3 0023	C 0043	S 0053	c 0063	s 0073
EOT 0004	DC4 0014	$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074
ENQ 0005	NAK 0015	% 0025	5 0035	E 0045	U 0055	e 0065	u 0075
ACK 0006	SYN 0016	& 0026	6 0036	F 0046	V 0056	f 0066	v 0076
BEL 0007	ETB 0017	’ 0027	7 0037	G 0047	W 0057	g 0067	w 0077
BS 0008	CAN 0018	(0028	8 0038	H 0048	X 0058	h 0068	x 0078
HT 0009	EM 0019	) 0029	9 0039	I 0049	Y 0059	i 0069	y 0079
LF 000A	SUB 001A	* 002A	: 003A	J 004A	Z 005A	j 006A	z 007A
VT 000B	ESC 001B	+ 002B	; 003B	K 004B	[005B	k 006B	{ 007B
FF 000C	FS 001C	, 002C	< 003C	L 004C	\ 005C	l 006C	| 007C
CR 000D	GS 001D	- 002D	= 003D	M 004D	] 005D	m 006D	} 007D
SO 000E	RS 001E	. 002E	> 003E	N 004E	^ 005E	n 006E	~ 007E
SI 000F	US 001F	/ 002F	? 003F	O 004F	_ 005F	o 006F	DEL 007F

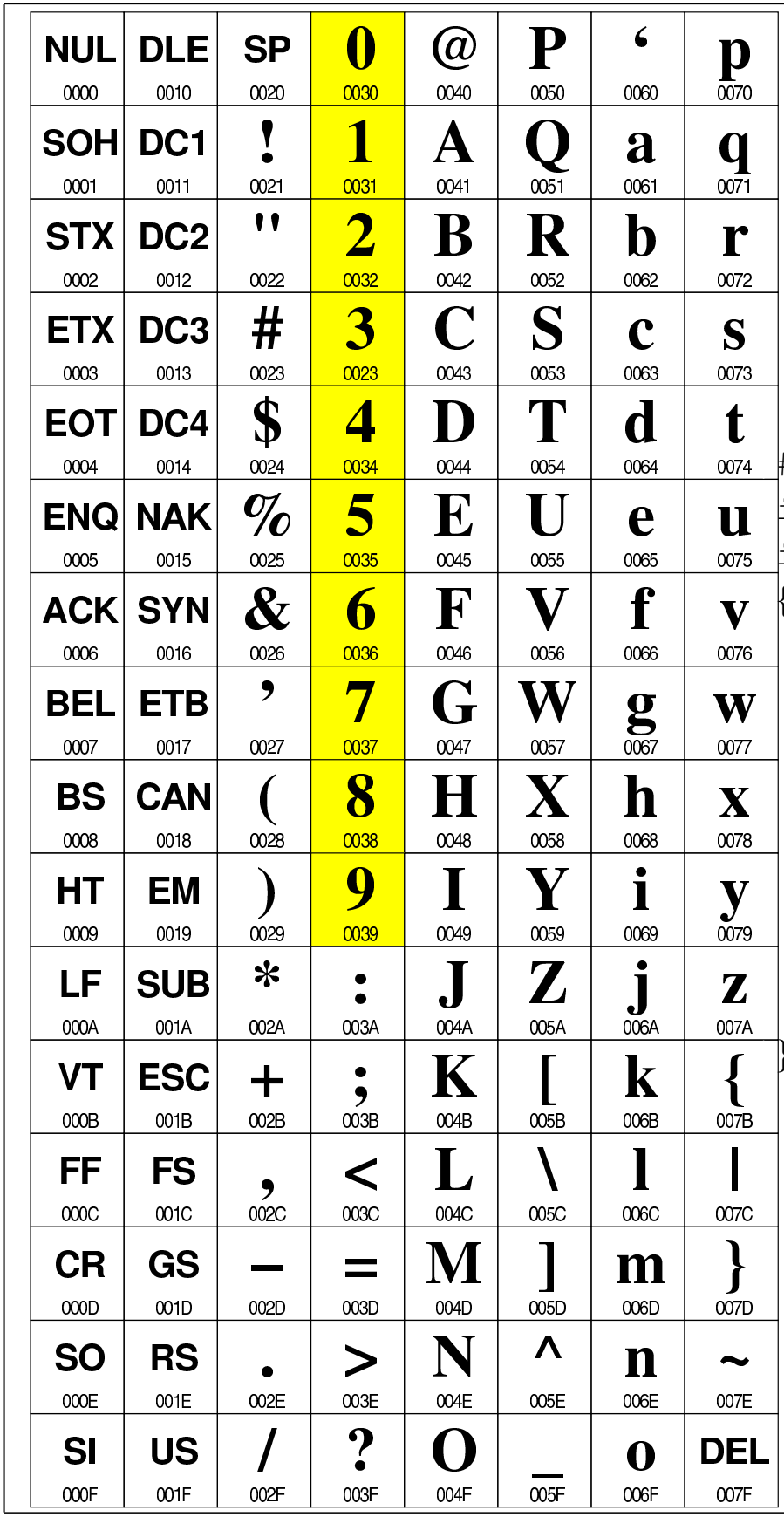
```
#include <ctype.h>
int
isblank (int c)
{
  if (c == ' '
      || c == '\t' )
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

Listato 69.52. Funzione *iscntrl()*.

NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	‘ 0060	p 0070
SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
ETX 0003	DC3 0013	# 0023	3 0023	C 0043	S 0053	c 0063	s 0073
EOT 0004	DC4 0014	$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074
ENQ 0005	NAK 0015	% 0025	5 0035	E 0045	U 0055	e 0065	u 0075
ACK 0006	SYN 0016	& 0026	6 0036	F 0046	V 0056	f 0066	v 0076
BEL 0007	ETB 0017	’ 0027	7 0037	G 0047	W 0057	g 0067	w 0077
BS 0008	CAN 0018	(0028	8 0038	H 0048	X 0058	h 0068	x 0078
HT 0009	EM 0019	) 0029	9 0039	I 0049	Y 0059	i 0069	y 0079
LF 000A	SUB 001A	* 002A	: 003A	J 004A	Z 005A	j 006A	z 007A
VT 000B	ESC 001B	+ 002B	; 003B	K 004B	[005B	k 006B	{ 007B
FF 000C	FS 001C	, 002C	< 003C	L 004C	\ 005C	l 006C	| 007C
CR 000D	GS 001D	- 002D	= 003D	M 004D	] 005D	m 006D	} 007D
SO 000E	RS 001E	. 002E	> 003E	N 004E	^ 005E	n 006E	~ 007E
SI 000F	US 001F	/ 002F	? 003F	O 004F	_ 005F	o 006F	DEL 007F

```
#include <ctype.h>
int
iscntrl (int c)
{
  if (((c >= 0x00)
       && (c <= 0x1F))
      || (c == 0x7F))
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

Listato 69.53. Funzione *isdigit()*.



NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	‘ 0060	p 0070
SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
ETX 0003	DC3 0013	# 0023	3 0023	C 0043	S 0053	c 0063	s 0073
EOT 0004	DC4 0014	$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074
ENQ 0005	NAK 0015	% 0025	5 0035	E 0045	U 0055	e 0065	u 0075
ACK 0006	SYN 0016	& 0026	6 0036	F 0046	V 0056	f 0066	v 0076
BEL 0007	ETB 0017	’ 0027	7 0037	G 0047	W 0057	g 0067	w 0077
BS 0008	CAN 0018	(0028	8 0038	H 0048	X 0058	h 0068	x 0078
HT 0009	EM 0019	) 0029	9 0039	I 0049	Y 0059	i 0069	y 0079
LF 000A	SUB 001A	* 002A	: 003A	J 004A	Z 005A	j 006A	z 007A
VT 000B	ESC 001B	+ 002B	; 003B	K 004B	[005B	k 006B	{ 007B
FF 000C	FS 001C	, 002C	< 003C	L 004C	\ 005C	l 006C	| 007C
CR 000D	GS 001D	– 002D	= 003D	M 004D	] 005D	m 006D	} 007D
SO 000E	RS 001E	. 002E	> 003E	N 004E	^ 005E	n 006E	~ 007E
SI 000F	US 001F	/ 002F	? 003F	O 004F	_ 005F	o 006F	DEL 007F

```
#include <ctype.h>
int
isdigit (int c)
{
  if ((c >= 0x30)
      && (c <= 0x39))
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

Listato 69.54. Funzione ***isgraph()***.

NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	‘ 0060	p 0070
SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
ETX 0003	DC3 0013	# 0023	3 0023	C 0043	S 0053	c 0063	s 0073
EOT 0004	DC4 0014	$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074
ENQ 0005	NAK 0015	% 0025	5 0035	E 0045	U 0055	e 0065	u 0075
ACK 0006	SYN 0016	& 0026	6 0036	F 0046	V 0056	f 0066	v 0076
BEL 0007	ETB 0017	’ 0027	7 0037	G 0047	W 0057	g 0067	w 0077
BS 0008	CAN 0018	(0028	8 0038	H 0048	X 0058	h 0068	x 0078
HT 0009	EM 0019	) 0029	9 0039	I 0049	Y 0059	i 0069	y 0079
LF 000A	SUB 001A	* 002A	: 003A	J 004A	Z 005A	j 006A	z 007A
VT 000B	ESC 001B	+ 002B	; 003B	K 004B	[005B	k 006B	{ 007B
FF 000C	FS 001C	, 002C	< 003C	L 004C	\ 005C	l 006C	| 007C
CR 000D	GS 001D	– 002D	= 003D	M 004D	] 005D	m 006D	} 007D
SO 000E	RS 001E	. 002E	> 003E	N 004E	^ 005E	n 006E	~ 007E
SI 000F	US 001F	/ 002F	? 003F	O 004F	_ 005F	o 006F	DEL 007F

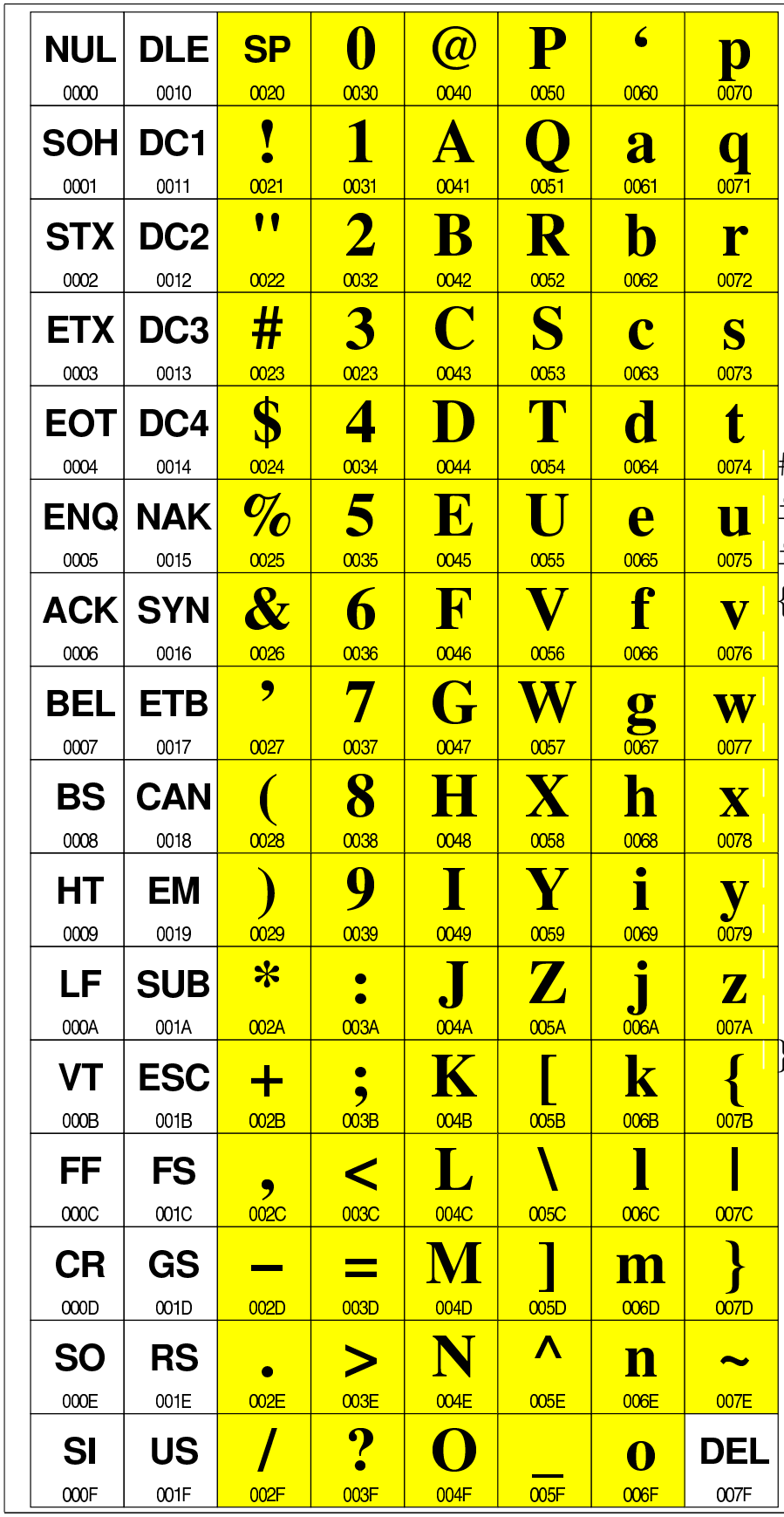
```
#include <ctype.h>
int
isgraph (int c)
{
  if ((c >= 0x21)
      && (c <= 0x7E))
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

Listato 69.55. Funzione ***islower()***.

NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	‘ 0060	p 0070
SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
ETX 0003	DC3 0013	# 0023	3 0023	C 0043	S 0053	c 0063	s 0073
EOT 0004	DC4 0014	$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074
ENQ 0005	NAK 0015	% 0025	5 0035	E 0045	U 0055	e 0065	u 0075
ACK 0006	SYN 0016	& 0026	6 0036	F 0046	V 0056	f 0066	v 0076
BEL 0007	ETB 0017	’ 0027	7 0037	G 0047	W 0057	g 0067	w 0077
BS 0008	CAN 0018	(0028	8 0038	H 0048	X 0058	h 0068	x 0078
HT 0009	EM 0019	) 0029	9 0039	I 0049	Y 0059	i 0069	y 0079
LF 000A	SUB 001A	* 002A	: 003A	J 004A	Z 005A	j 006A	z 007A
VT 000B	ESC 001B	+ 002B	; 003B	K 004B	[005B	k 006B	{ 007B
FF 000C	FS 001C	, 002C	< 003C	L 004C	\ 005C	l 006C	| 007C
CR 000D	GS 001D	– 002D	= 003D	M 004D	] 005D	m 006D	} 007D
SO 000E	RS 001E	. 002E	> 003E	N 004E	^ 005E	n 006E	~ 007E
SI 000F	US 001F	/ 002F	? 003F	O 004F	_ 005F	o 006F	DEL 007F

```
#include <ctype.h>
int
islower (int c)
{
  if ((c >= 0x61)
      && (c <= 0x7A))
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

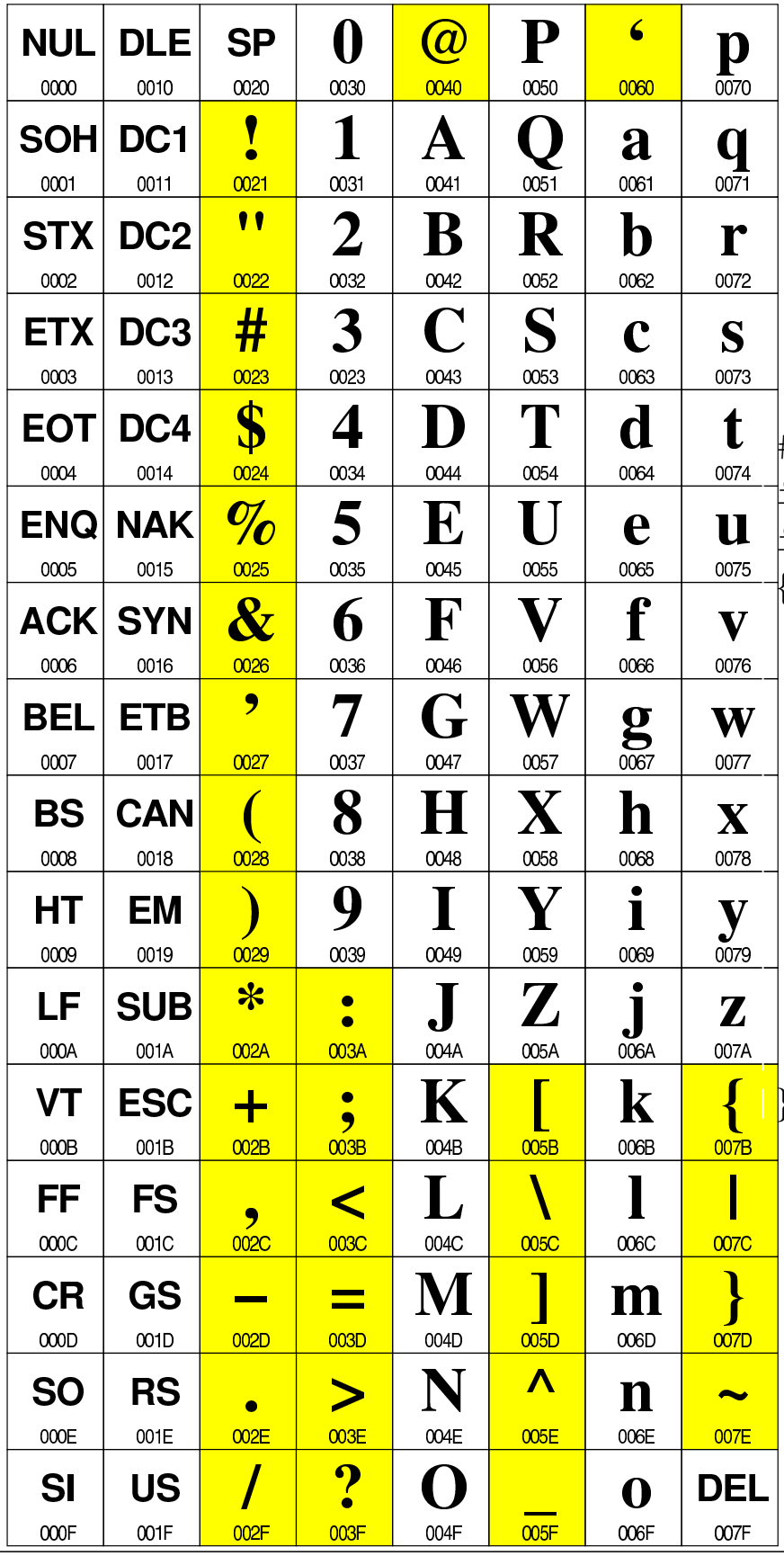
Listato 69.56. Funzione ***isprint()***.



NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	‘ 0060	p 0070
SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
ETX 0003	DC3 0013	# 0023	3 0023	C 0043	S 0053	c 0063	s 0073
EOT 0004	DC4 0014	$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074
ENQ 0005	NAK 0015	% 0025	5 0035	E 0045	U 0055	e 0065	u 0075
ACK 0006	SYN 0016	& 0026	6 0036	F 0046	V 0056	f 0066	v 0076
BEL 0007	ETB 0017	’ 0027	7 0037	G 0047	W 0057	g 0067	w 0077
BS 0008	CAN 0018	(0028	8 0038	H 0048	X 0058	h 0068	x 0078
HT 0009	EM 0019	) 0029	9 0039	I 0049	Y 0059	i 0069	y 0079
LF 000A	SUB 001A	* 002A	: 003A	J 004A	Z 005A	j 006A	z 007A
VT 000B	ESC 001B	+ 002B	; 003B	K 004B	[005B	k 006B	{ 007B
FF 000C	FS 001C	, 002C	< 003C	L 004C	\ 005C	l 006C	| 007C
CR 000D	GS 001D	– 002D	= 003D	M 004D	] 005D	m 006D	} 007D
SO 000E	RS 001E	. 002E	> 003E	N 004E	^ 005E	n 006E	~ 007E
SI 000F	US 001F	/ 002F	? 003F	O 004F	_ 005F	o 006F	DEL 007F

```
#include <ctype.h>
int
isprint (int c)
{
  if ((c >= 0x20)
      && (c <= 0x7E))
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

Listato 69.57. Funzione ***ispunct()***.



NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	‘ 0060	p 0070
SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
ETX 0003	DC3 0013	# 0023	3 0023	C 0043	S 0053	c 0063	s 0073
EOT 0004	DC4 0014	$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074
ENQ 0005	NAK 0015	% 0025	5 0035	E 0045	U 0055	e 0065	u 0075
ACK 0006	SYN 0016	& 0026	6 0036	F 0046	V 0056	f 0066	v 0076
BEL 0007	ETB 0017	’ 0027	7 0037	G 0047	W 0057	g 0067	w 0077
BS 0008	CAN 0018	(0028	8 0038	H 0048	X 0058	h 0068	x 0078
HT 0009	EM 0019	) 0029	9 0039	I 0049	Y 0059	i 0069	y 0079
LF 000A	SUB 001A	* 002A	: 003A	J 004A	Z 005A	j 006A	z 007A
VT 000B	ESC 001B	+ 002B	; 003B	K 004B	[005B	k 006B	{ 007B
FF 000C	FS 001C	, 002C	< 003C	L 004C	\ 005C	l 006C	| 007C
CR 000D	GS 001D	– 002D	= 003D	M 004D	] 005D	m 006D	} 007D
SO 000E	RS 001E	. 002E	> 003E	N 004E	^ 005E	n 006E	~ 007E
SI 000F	US 001F	/ 002F	? 003F	O 004F	_ 005F	o 006F	DEL 007F

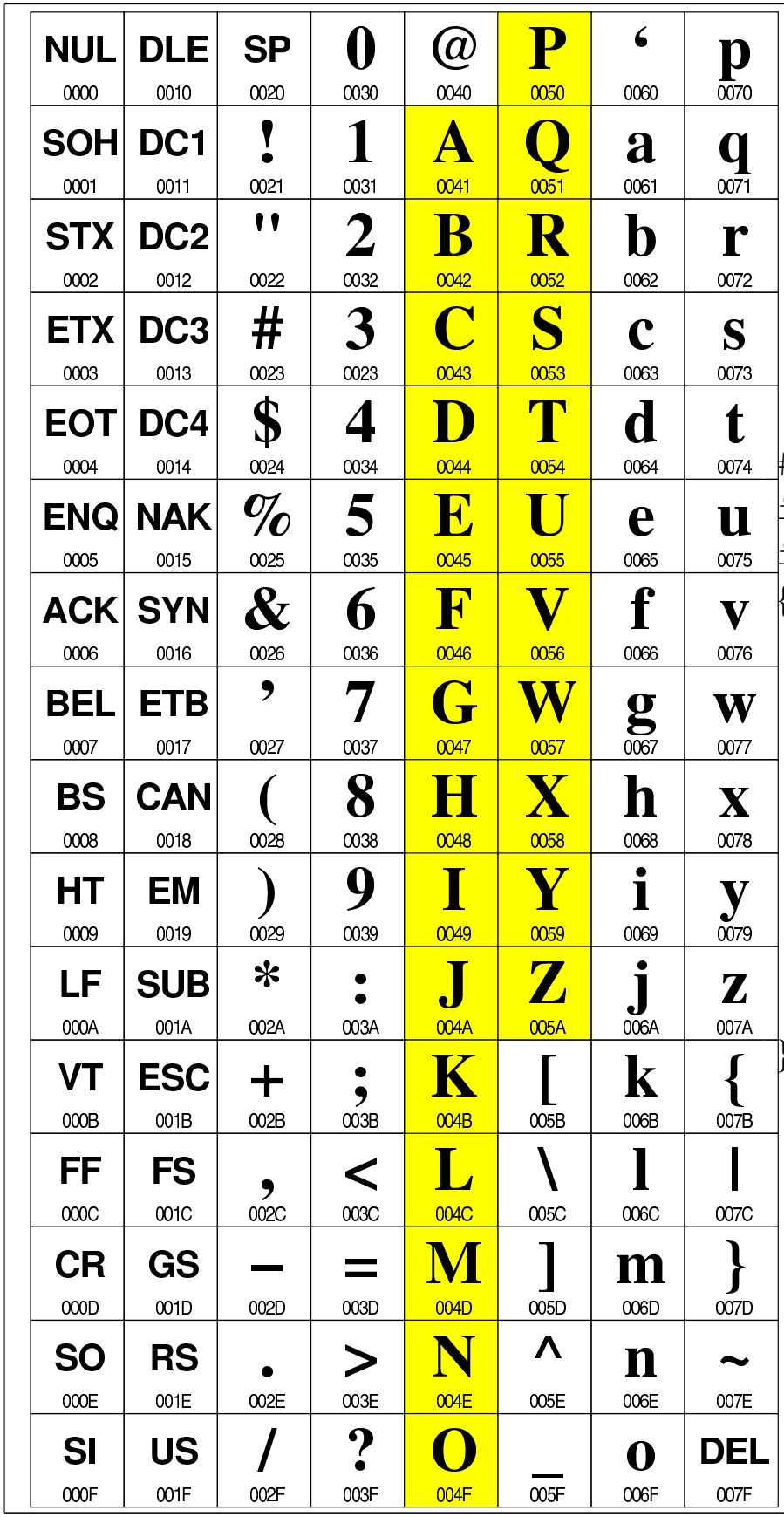
```
#include <ctype.h>
int
ispunct (int c)
{
  if (isgraph (c)
      && (! isspace (c))
      && (! isalnum(c)))
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

Listato 69.58. Funzione *isspace()*.

NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	‘ 0060	p 0070
SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
ETX 0003	DC3 0013	# 0023	3 0023	C 0043	S 0053	c 0063	s 0073
EOT 0004	DC4 0014	$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074
ENQ 0005	NAK 0015	% 0025	5 0035	E 0045	U 0055	e 0065	u 0075
ACK 0006	SYN 0016	& 0026	6 0036	F 0046	V 0056	f 0066	v 0076
BEL 0007	ETB 0017	’ 0027	7 0037	G 0047	W 0057	g 0067	w 0077
BS 0008	CAN 0018	(0028	8 0038	H 0048	X 0058	h 0068	x 0078
HT 0009	EM 0019	) 0029	9 0039	I 0049	Y 0059	i 0069	y 0079
LF 000A	SUB 001A	* 002A	: 003A	J 004A	Z 005A	j 006A	z 007A
VT 000B	ESC 001B	+ 002B	; 003B	K 004B	[005B	k 006B	{ 007B
FF 000C	FS 001C	, 002C	< 003C	L 004C	\ 005C	l 006C	| 007C
CR 000D	GS 001D	– 002D	= 003D	M 004D	] 005D	m 006D	} 007D
SO 000E	RS 001E	. 002E	> 003E	N 004E	^ 005E	n 006E	~ 007E
SI 000F	US 001F	/ 002F	? 003F	O 004F	_ 005F	o 006F	DEL 007F

```
#include <ctype.h>
int
isspace (int c)
{
  if (  (c == ' ')
      || (c == '\f')
      || (c == '\n')
      || (c == '\r')
      || (c == '\t')
      || (c == '\v'))
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

Listato 69.59. Funzione ***isupper()***.



NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	‘ 0060	p 0070
SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
ETX 0003	DC3 0013	# 0023	3 0023	C 0043	S 0053	c 0063	s 0073
EOT 0004	DC4 0014	$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074
ENQ 0005	NAK 0015	% 0025	5 0035	E 0045	U 0055	e 0065	u 0075
ACK 0006	SYN 0016	& 0026	6 0036	F 0046	V 0056	f 0066	v 0076
BEL 0007	ETB 0017	’ 0027	7 0037	G 0047	W 0057	g 0067	w 0077
BS 0008	CAN 0018	(0028	8 0038	H 0048	X 0058	h 0068	x 0078
HT 0009	EM 0019	) 0029	9 0039	I 0049	Y 0059	i 0069	y 0079
LF 000A	SUB 001A	* 002A	: 003A	J 004A	Z 005A	j 006A	z 007A
VT 000B	ESC 001B	+ 002B	; 003B	K 004B	[005B	k 006B	{ 007B
FF 000C	FS 001C	, 002C	< 003C	L 004C	\ 005C	l 006C	| 007C
CR 000D	GS 001D	– 002D	= 003D	M 004D	] 005D	m 006D	} 007D
SO 000E	RS 001E	. 002E	> 003E	N 004E	^ 005E	n 006E	~ 007E
SI 000F	US 001F	/ 002F	? 003F	O 004F	_ 005F	o 006F	DEL 007F

```
#include <ctype.h>
int
isupper (int c)
{
  if ((c >= 0x41)
      && (c <= 0x5A))
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

Listato 69.60. Funzione *isxdigit()*.

NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	‘ 0060	p 0070
SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
ETX 0003	DC3 0013	# 0023	3 0023	C 0043	S 0053	c 0063	s 0073
EOT 0004	DC4 0014	$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074
ENQ 0005	NAK 0015	% 0025	5 0035	E 0045	U 0055	e 0065	u 0075
ACK 0006	SYN 0016	& 0026	6 0036	F 0046	V 0056	f 0066	v 0076
BEL 0007	ETB 0017	’ 0027	7 0037	G 0047	W 0057	g 0067	w 0077
BS 0008	CAN 0018	(0028	8 0038	H 0048	X 0058	h 0068	x 0078
HT 0009	EM 0019	) 0029	9 0039	I 0049	Y 0059	i 0069	y 0079
LF 000A	SUB 001A	* 002A	: 003A	J 004A	Z 005A	j 006A	z 007A
VT 000B	ESC 001B	+ 002B	; 003B	K 004B	[005B	k 006B	{ 007B
FF 000C	FS 001C	, 002C	< 003C	L 004C	\ 005C	l 006C	| 007C
CR 000D	GS 001D	– 002D	= 003D	M 004D	] 005D	m 006D	} 007D
SO 000E	RS 001E	. 002E	> 003E	N 004E	^ 005E	n 006E	~ 007E
SI 000F	US 001F	/ 002F	? 003F	O 004F	_ 005F	o 006F	DEL 007F

```
#include <ctype.h>
int
isxdigit (int c)
{
  if (((c >= 0x30)
       && (c <= 0x39))
      || ((c >= 0x41)
          && (c <= 0x46))
      || ((c >= 0x61)
          && (c <= 0x66)))
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

Listato 69.61. Nello standard POSIX si aggiunge la funzione *isascii()*.


```
#include <ctype.h>
int
isascii (int c)
{
  if ((c >= 0x00)
      && (c <= 0x7F))
    {
      return 1;
    }
  else
    {
      return 0;
    }
}
```

POSIX

69.7.2 macroistruzioni «is...()»

«

In alternativa a delle funzioni vere e proprie, si possono realizzare semplicemente delle macroistruzioni per verificare le condizioni riferite al carattere. Il listato seguente è conforme a quanto già visto nella sezione precedente:

```
#define isblank(C)  ((int) (C == ' ' || C == '\t'))
#define isspace(C)  ((int) (C == ' ' \
                            || C == '\f' \
                            || C == '\n' \
                            || C == '\n' \
                            || C == '\r' \
                            || C == '\t' \
                            || C == '\v'))
#define isdigit(C)  ((int) (C >= '0' && C <= '9'))
#define isxdigit(C) ((int) ((C >= '0' && C <= '9') \
                             || (C >= 'A' && C <= 'F') \
                             || (C >= 'a' && C <= 'f')))
#define isupper(C)  ((int) (C >= 'A' && C <= 'Z'))
#define islower(C)  ((int) (C >= 'a' && C <= 'z'))
#define iscntrl(C)  ((int) ((C >= 0x00 && C <= 0x1F)
                           || C == 0x7F))
#define isgraph(C)  ((int) (C >= 0x21 && C <= 0x7E))
#define isprint(C)  ((int) (C >= 0x20 && C <= 0x7E))
#define isalpha(C)  (isupper (C) || islower (C))
#define isalnum(C)  (isalpha (C) || isdigit (C))
#define ispunct(C)  (isgraph (C) && (!isspace (C))
                                 && (!isalnum (C)))
#define isascii(C)  ((int) (C >= 0x00 && C <= 0x7F))
```

69.7.3 Funzioni di conversione

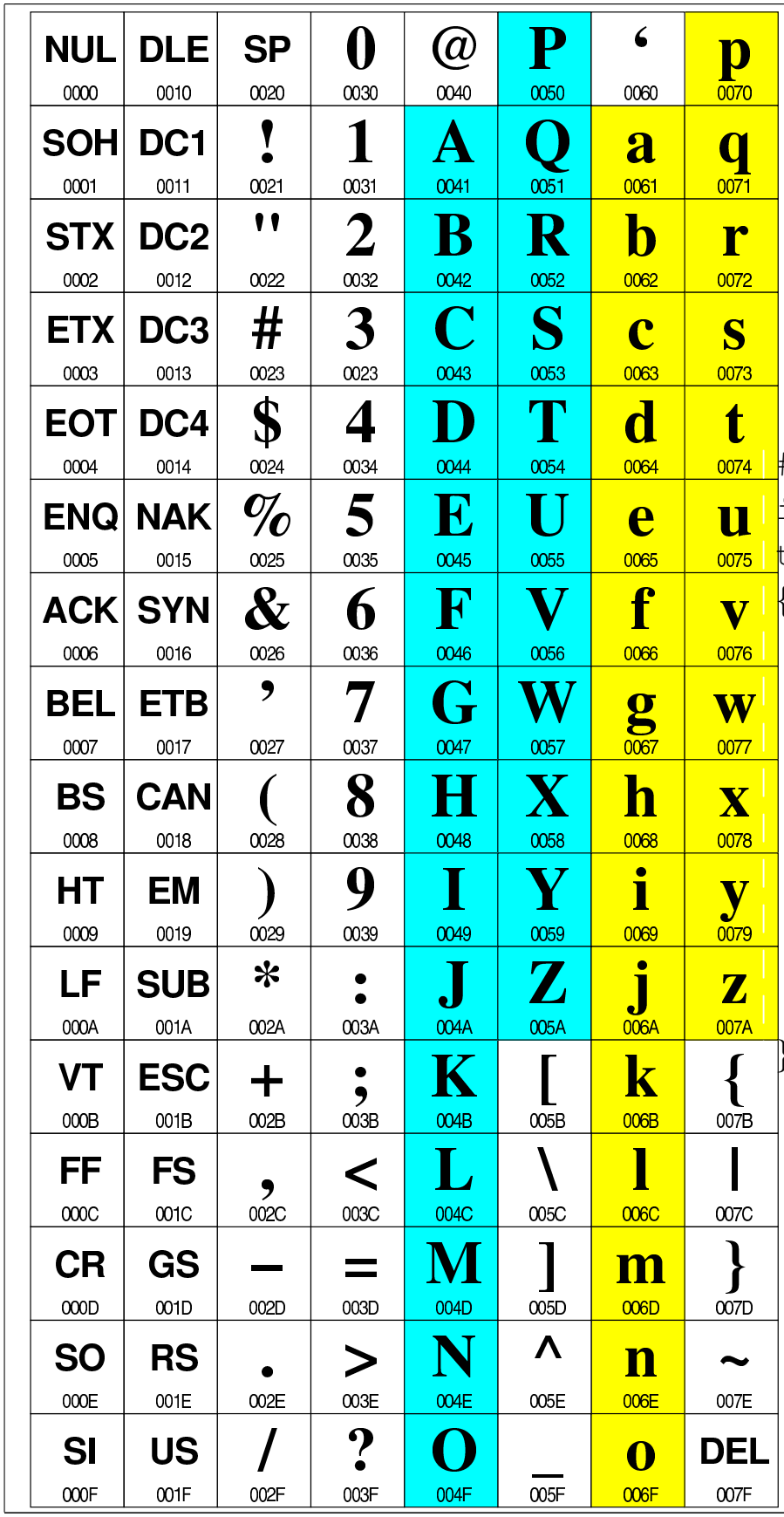
Le due funzioni ***tolower()*** e ***toupper()*** si occupano di convertire un carattere, rispettivamente, in minuscolo o in maiuscolo.

Listato 69.63. Funzione *tolower()*.

NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	‘ 0060	p 0070
SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
ETX 0003	DC3 0013	# 0023	3 0023	C 0043	S 0053	c 0063	s 0073
EOT 0004	DC4 0014	$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074
ENQ 0005	NAK 0015	% 0025	5 0035	E 0045	U 0055	e 0065	u 0075
ACK 0006	SYN 0016	& 0026	6 0036	F 0046	V 0056	f 0066	v 0076
BEL 0007	ETB 0017	’ 0027	7 0037	G 0047	W 0057	g 0067	w 0077
BS 0008	CAN 0018	(0028	8 0038	H 0048	X 0058	h 0068	x 0078
HT 0009	EM 0019	) 0029	9 0039	I 0049	Y 0059	i 0069	y 0079
LF 000A	SUB 001A	* 002A	: 003A	J 004A	Z 005A	j 006A	z 007A
VT 000B	ESC 001B	+ 002B	; 003B	K 004B	[005B	k 006B	{ 007B
FF 000C	FS 001C	, 002C	< 003C	L 004C	\ 005C	l 006C	| 007C
CR 000D	GS 001D	- 002D	= 003D	M 004D	] 005D	m 006D	} 007D
SO 000E	RS 001E	. 002E	> 003E	N 004E	^ 005E	n 006E	~ 007E
SI 000F	US 001F	/ 002F	? 003F	O 004F	_ 005F	o 006F	DEL 007F

```
#include <ctype.h>
int
tolower (int c)
{
  if (isupper (c))
    {
      return
        (c + 0x20);
    }
  else
    {
      return c;
    }
}
```

Listato 69.64. Funzione ***toupper()***.




```
#include <ctype.h>
int
toupper (int c)
{
  if (islower (c))
    {
      return
        (c - 0x20);
    }
  else
    {
      return c;
    }
}
```

Lo standard POSIX aggiunge anche la funzione ***toascii()*** che si limita ad azzerare i bit più significativi, dopo il settimo.

POSIX

Listato 69.65. Funzione ***toascii()***.

```
#include <ctype.h>
int
toascii (int c)
{
  return (c & 0x7F);
}
```

69.7.4 macroistruzioni di conversione

«

Anche le funzioni ***toupper()***, ***tolower()*** e ***toascii()*** possono essere rappresentate agevolmente in forma di macroistruzioni. Il listato seguente è conforme a quanto già visto nella sezione precedente:

```
#define tolower(C)   (isupper (C) ? ((C) + 0x20) : (C))
#define toupper(C)   (islower (C) ? ((C) - 0x20) : (C))
#define toascii(C)   (C & 0x7F)
```

Ma nel caso dello standard POSIX, in questo caso vanno ancora aggiunge due macroistruzioni, a cui non fanno capo funzioni con lo stesso nome:

POSIX

```
#define _tolower(C)  (isupper (C) ? ((C) + 0x20) : (C))
#define _toupper(C)  (islower (C) ? ((C) - 0x20) : (C))
```

69.7.5 Esempio di utilizzo delle funzioni

«

Viene proposto un programma molto semplice che utilizza tutte le funzioni dichiarate nel file ‘`ctype.h`’, ma solo secondo lo standard C:

```
#include <stdio.h>
#include <ctype.h>
```

```
int
main (int argc, char *argv[])
{
  int c;
  for (c = 0; c <= 0x7F; c++)
    {
      printf ("%02x", c);
      printf ("\t"); if (iscntrl (c))  printf ("cntrl");
                     if (isprint (c))  printf ("print");
      printf ("\t"); if (isblank (c))  printf ("blank");
                     if (isgraph (c))  printf ("graph");
      printf ("\t"); if (isspace (c))  printf ("space");
                     if (ispunct (c))  printf ("punct");
                     if (isupper (c))  printf ("upper");
                     if (islower (c))  printf ("lower");
                     if (isdigit (c))  printf ("digit");
      printf ("\t"); if (isalnum (c))  printf ("alnum");
      printf ("\t"); if (isxdigit (c)) printf ("xdigit");
      printf ("\t"); if (isalpha (c))  printf ("alpha");
      printf ("\n");
    }
  printf ("\n");
  printf ("ASCII:\n");
  for (c = 0; c <= 0x7F; c++)
    {
      if (isprint (c)) printf ("%c", c);
    }
  printf ("\n");
  printf ("\n");
  printf ("to upper:\n");
  for (c = 0; c <= 0x7F; c++)
    {
      if (isprint (c)) printf ("%c", toupper (c));
```

```
    }
  printf ("\n");
  printf ("\n");
  printf ("to lower:\n");
  for (c = 0; c <= 0x7F; c++)
    {
      if (isprint (c)) printf ("%c", tolower (c));
    }
  printf ("\n");
  return 0;
}
```

Una volta compilato il programma, avviandolo si deve ottenere un testo come quello che si vede nell'estratto seguente:

```
…
44      print   graph   upper   alnum   xdigit  alpha
45      print   graph   upper   alnum   xdigit  alpha
46      print   graph   upper   alnum   xdigit  alpha
47      print   graph   upper   alnum           alpha
48      print   graph   upper   alnum           alpha
…
ASCII:
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJ↩
↪KLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~

to upper:
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJ↩
↪KLMNOPQRSTUVWXYZ[\]^_`ABCDEFGHIJKLMNOPQRSTUVWXYZ{|}~

to lower:
!"#$%&'()*+,-./0123456789:;<=>?@abcdefghij↩
↪klmnopqrstuvwxyz[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

69.8 File «stdarg.h»

Il file ‘`stdarg.h`’ della libreria standard definisce principalmente delle macroistruzioni per gestire gli argomenti variabili passati a una funzione, assieme a un tipo di variabile, ‘**`va_list`**’, specifico per gestire il puntatore a tali parametri non dichiarati (si veda eventualmente la realizzazione del file ‘`stdarg.h`’ nei sorgenti di os32, listato 95.1.10).

Tabella 69.70. macroistruzioni standard per la gestione di argomenti variabili.

Macroistruzione	Descrizione
`void va_start (va_list` ***ap***`,` `  `***parametro_n***`);`	Inizializza la variabile ***ap***, di tipo ‘**`va_list`**’, in modo che punti all’area di memoria immediatamente successiva al parametro indicato, il quale deve essere l’ultimo.
tipo `va_arg (va_list` ***ap***`,` ***tipo***`);`	Restituisce il contenuto dell’area di memoria a cui punta ***ap***, utilizzando il tipo indicato, incrementando contestualmente il puntatore in modo che, al termine, si trovi nell’area di memoria immediatamente successiva.
`void va_copy (va_list` ***dst***`,` `  va_list` ***org***`);`	Copia il puntatore ***org*** nella variabile ***dst***.

Macroistruzione	Descrizione
`void va_end (va_list ap);`	Conclude l'utilizzo del puntatore ***ap***.

69.8.1 Realizzazione

«

Il listato successivo è tutto ciò che serve per realizzare la libreria:

```
typedef unsigned char * va_list;

#define va_start(AP, LAST) \
    ((void) ((AP) = ((va_list) &(LAST)) + (sizeof (LAST))))

#define va_end(AP) ((void) ((AP) = 0))
#define va_copy(DEST, SRC) \
    ((void) ((DEST) = (va_list) (SRC)))

#define va_arg(AP, TYPE) \
    (((AP) = (AP) + (sizeof (TYPE))), \
        *((TYPE *) ((AP) - (sizeof (TYPE)))))
```

Delle macroistruzioni mostrate nell'esempio, la più difficile da interpretare potrebbe essere ***va_arg***, la quale deve restituire il valore dell'area di memoria puntata inizialmente, ma garantendo di lasciare il puntatore pronto per l'area successiva. In pratica, prima viene incrementato il puntatore per l'area successiva, quindi viene dereferenziato ricalcolando lo spazio necessario a raggiungere la posizione precedente. In altri termini è come scrivere:

```
…
va_list ap;
…
//
```

```
// va_start
//
ap = (va_list) &ultimo_parametro;
ap = ap + (sizeof (tipo_ultimo_parametro);
…
//
// va_arg
//
ap = ap + (sizeof tipo_successivo);
var = (tipo_successivo *)
      (ap - (sizeof (tipo_successivo)));
…
//
// va_end
//
ap = 0;
…
```

69.8.2 Esempio di utilizzo delle macro

Viene riproposto un programma molto semplice, già apparso in altri capitoli, per dimostrare l'utilizzo delle macroistruzioni dichiarate nel file '`stdarg.h`'.

```
#include <stdio.h>
#include <stdarg.h>

void
f (int w,...)
{
  long double x;     // Dichiara le variabili che servono
  long long int y;   // a contenere gli argomenti per i
  int z;             // quali mancano i parametri formali.
```

```
  va_list ap;         // Dichiara il puntatore agli argomenti.

  va_start (ap, w); // Posiziona il puntatore dopo la fine
                    // di «w».

  x = va_arg (ap, long double); // Estrae l'argomento
                                // successivo portando
                                // avanti il puntatore
                                // di conseguenza.

  printf ("w = %i; ", w);    // Mostra il valore del primo
                             // parametro.
  printf ("x = %Lf; ", x);   // Mostra il valore
                             // dell'argomento successivo.

  y = va_arg (ap, long long int);  // Estrapola e mostra
  printf ("y = %lli; ", y);        // il terzo argomento.

  z = va_arg (ap, int);            // Estrapola e mostra
  printf ("z = %i\n", z);          // il quarto argomento.

  va_end (ap);                     // Conclude la scansione.

  return;
}

int
main (int argc, char *argv[])
{
  f (10, (long double)12.34, (long long int)13, 14);
  return 0;
}
```

Avviando il programma di esempio si deve visualizzare il messaggio

seguente:

```
w = 10; x = 12.340000; y = 13; z = 14
```

69.8.3 Promozione

Va ricordato che gli argomenti delle chiamate alle funzioni vengono adattati in modo tale da facilitare l'uso della pila dei dati. Pertanto, i valori che prevedono una rappresentazione in memoria troppo piccola, subiscono quella che è nota come «promozione».

La funzione che ha degli argomenti variabili, dovrebbe gestire solo valori che non possono subire una trasformazione di questo tipo, altrimenti, quando poi utilizza la macroistruzione ***va_arg*** deve indicare un tipo adeguato alla promozione che si prevede sia applicata ai valori degli argomenti.

A questo proposito si può notare che nell'esempio di utilizzo che appare nella sezione 69.8.2, non si fa mai uso di tipi di dati di rango inferiore a '**`int`**'.

69.9 File «stdlib.h»

Il file '`stdlib.h`' della libreria standard definisce alcuni tipi di dati, varie funzioni di utilità generale e alcune macro-variabili. Viene proposto un esempio di questo file, e di alcune delle funzioni a cui si riferisce, indicando per le altre solo i prototipi (si veda eventualmente la realizzazione del file '`stdlib.h`' e di alcune delle sue funzioni, nei sorgenti di os32, sezione 95.19).

Lo standard POSIX estende significativamente il contenuto del file '`stdlib.h`', ma qui non si fa riferimento ad alcuna di tali estensioni.

69.9.1 Tipi di dati speciali

«

I tipi di dati che il file ‘`stdlib.h`’ definisce sono ‘**`size_t`**’ e ‘**`wchar_t`**’, già descritti nel file ‘`stddef.h`’ (sezione 69.13), oltre a ‘**`div_t`**’, ‘**`ldiv_t`**’ e ‘**`lldiv_t`**’. I tipi ‘...**`div_t`**’ sono delle strutture il cui scopo è quello di contenere il risultato di una divisione, espresso come quoziente e resto. Questi tipi di dati si usano per contenere il valore restituito dalle funzioni ***div()***, ***ldiv()*** e ***lldiv()***. La distinzione tra i tre tipi deriva dalla capienza dei membri della struttura. Ecco come potrebbero essere dichiarati:

```
typedef struct {
    int quot;
    int rem;
} div_t;
typedef struct {
    long int quot;
    long int rem;
} ldiv_t;
typedef struct {
    long long int quot;
    long long int rem;
} lldiv_t;
```

69.9.2 Macro-variabili

«

Il file ‘`stdlib.h`’ dichiara nuovamente la macro-variabile ***NULL***, come già avviene nel file ‘`stddef.h`’ (sezione 69.13); inoltre definisce quelle seguenti:

Macro	Descrizione
`EXIT_SUCCESS` `EXIT_FAILURE`	Rappresentano un numero intero che possa essere usato come argomento della funzione ***exit()***, per rappresentare il successo o l'insuccesso dell'attività svolta. Normalmente, i valori in cui si espandono le due macro-variabili sono rispettivamente zero e uno.
`RAND_MAX`	Rappresenta il valore massimo che possa essere generato dalla funzione ***rand()*** e generalmente corrisponde al valore massimo che può assumere un numero intero di tipo '**`int`**'.
`MB_CUR_MAX`	Rappresenta la quantità massima di byte che possono essere utilizzati in una sequenza multibyte, in base alla configurazione locale. Questo valore è un intero che deve essere di tipo '**`size_t`**' è non può superare il limite rappresentato dalla macro-variabile ***MB_LEN_MAX*** (dichiarata nel file '`limits.h`', descritto nella sezione 69.3).

Merita un po' di attenzione la macro-variabile ***MB_CUR_MAX***. La sequenza multibyte è una sequenza di byte che, in base alla configurazione locale, deve essere interpretata come un carattere singolo. Per esempio, questo meccanismo si utilizza nella codifica UTF-8 e in altre; ma proprio perché esistono più metodi alternativi, per quanto superati possano essere rispetto a UTF-8, la configurazione locale stabilisce le regole particolari per interpretare tali sequenze e i limiti rispetto a queste. Pertanto, la macro-variabile ***MB_CUR_MAX***

dovrebbe espandersi in una funzione che restituisce il valore desiderato, in relazione alla configurazione locale che si trova a essere attiva in un certo momento. Per semplicità, nell'esempio che viene proposto si associa il valore di questa macro-variabile a quello massimo accettabile in assoluto.

```
#define EXIT_FAILURE    1
#define EXIT_SUCCESS    0

#define RAND_MAX        INT_MAX

#define MB_CUR_MAX      ((size_t) MB_LEN_MAX)  // [1]
//
// [1] Sarebbe meglio una funzione.
//
```

Nell'esempio proposto viene usata la macro-variabile ***MB_LEN_MAX***, pertanto, in questo modo si rende necessaria l'inclusione del file '`limits.h`' che deve contenere la sua dichiarazione.

69.9.3 Conversioni numeriche

«

Un gruppo di funzioni del file '`stdlib.h`' permette di convertire una stringa in un valore numerico. In particolare, le funzioni con nomi del tipo ***ato...()*** (*ASCII to ...*) non eseguono controlli particolari e non modificano la variabile ***errno*** (sezione 69.5); invece, le funzioni con nomi ***strto...()*** (*string to ...*) sono più sofisticate.

Le funzioni ***ato...()*** interpretano una stringa e convertono il suo contenuto in un numero intero o in un numero a virgola mobile. Le funzioni sono ***atoi()***, ***atol()***, ***atoll()*** e ***atof()***, che convertono rispettivamente in un tipo '**`int`**', '**`long int`**', '**`long long int`**' e '**`double`**'. Ecco i prototipi:

```
int           atoi  (const char *nptr);
long int      atol  (const char *nptr);
long long int atoll (const char *nptr);
double        atof  (const char *nptr);
```

Viene proposta una soluzione per queste funzioni di conversione:

```
#include <stdlib.h>
#include <ctype.h>
int
atoi (const char *nptr)
{
  int i;
  int sign = +1;
  int n;

  for (i = 0; isspace (nptr[i]); i++)
    {
      ;       // Si limita a saltare gli spazi iniziali.
    }

  if (nptr[i] == '+')
    {
      sign = +1;
      i++;
    }
  else if (nptr[i] == '-')
    {
      sign = -1;
      i++;
    }

  for (n = 0; isdigit (nptr[i]); i++)
    {
      n = (n * 10) + (nptr[i] - '0'); // Accumula il valore.
    }
```

```
  return sign * n;
}
```

Logicamente, le funzioni ***atol()*** e ***atoll()*** sono praticamente uguali, con la differenza che la variabile automatica ***n*** deve essere dello stesso tipo restituito dalla funzione; pertanto si passa alla soluzione proposta per la funzione ***atof()***:

```
#include <stdlib.h>
#include <ctype.h>
double
atof (const char *nptr)
{
  int i;
  int sign = +1;
  double n;             // Il risultato sarà:  n / d.
  double d;             //

  for (i = 0; isspace (nptr[i]); i++)
    {
      ;         // Si limita a saltare gli spazi iniziali.
    }

  if (nptr[i] == '+')
    {
      sign = +1;
      i++;
    }
  else if (nptr[i] == '-')
    {
      sign = -1;
      i++;
    }
```

```
  for (n = 0.0; isdigit (nptr[i]); i++)
    {
      n = (n * 10.0) + (nptr[i] - '0');      // Accumula
                                             // il valore.
    }

  if (nptr[i] == '.')
    {
      i++;
    }

  for (d = 1.0; isdigit (nptr[i]); i++)
    {
      // La variabile "d" viene inizializzata in ogni caso.

      n = (n * 10.0) + (nptr[i] - '0');
      d = d * 10.0;    // Tiene conto di quanto dovrà essere
                       // diviso il risultato.
    }

  return sign * n / d;
}
```

Le funzioni ***strto...()*** sono più complesse rispetto a quelle ***ato...()***. Per dare una descrizione sommaria, si può osservare che, oltre alla stringa da scandire ricevono un puntatore di puntatore all'ultimo elemento utile di tale stringa;[2] se poi questo è nullo, la scansione avviene normalmente nella stringa, entro il limite del carattere nullo di terminazione. Se fallisce il riconoscimento del valore da tradurre, il puntatore all'inizio della stringa viene copiato nell'area di memoria a cui punta il puntatore di puntatore, a meno che questo, inizialmen-

te, sia già nullo (potrebbe essere nullo il puntatore principale o il contenuto dell'area a cui punta).

In caso di errore nell'interpretazione del valore, queste funzioni utilizzano la variabile ***errno*** per annotare il tipo di problema riscontrato.

I valori che possono essere convertiti sono esprimibili in notazione decimale o esadecimale; inoltre, le funzioni che convertono in valori a virgola mobile, accettano una notazione esponenziale e delle parole chiave per rappresentare l'infinito e NaN (*Not a number*). Nel caso particolare delle funzioni che convertono in un numero intero, esiste un terzo parametro per specificare la base di numerazione attesa.

Vengono presentati solo i prototipi di queste funzioni:

```
float                   strtof   (const char * restrict nptr,
                                  char ** restrict endptr);
double                  strtod   (const char * restrict nptr,
                                  char ** restrict endptr);
long double             strtold  (const char * restrict nptr,
                                  char ** restrict endptr);
```

```
long int                strtol   (const char * restrict nptr,
                                  char ** restrict endptr,
                                  int base);
long long int           strtoll  (const char * restrict nptr,
                                  char ** restrict endptr,
                                  int base);
unsigned long int       strtoul  (const char * restrict nptr,
                                  char ** restrict endptr,
                                  int base);
unsigned long long int  strtoull (const char * restrict nptr,
                                  char ** restrict endptr,
                                  int base);
```

Per una descrizione completa si vedano le pagine di manuale *strtof(3)*, *strtod(3)*, *strtold(3)*, *strtol(3)*, *strtoll(3)*, *strtoul(3)* e *strtoull(3)*, oltre alla documentazione standard citata alla fine del capitolo.

L'esempio seguente mostra l'uso delle funzioni ***ato...()***:

```
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
  int              i;
  long int        li;
  long long int lli;
  double           d;
  char           s[] = "  -987654.3210";

  i = atoi (s);
  li = atol (s);
  lli = atoll (s);
  d = atof (s);

  printf ("\"%s\" = %i, %li, %lli, %f\n", s, i, li, lli, d);

  return 0;
}
```

Il risultato che ci si attende di visualizzare è questo:

```
"  -987654.3210" = -987654, -987654, -987654, -987654.321000
```

69.9.4 Funzioni per la generazione di numeri in modo pseudo-casuale

«

La libreria standard deve disporre, nel file ‘`stdlib.h`’, di due funzioni per la generazione di numeri pseudo-casuali. Si tratta precisamente di ***rand()*** che restituisce un numero intero casuale (di tipo ‘**`int`**’, ma sono ammessi solo valori positivi) e di ***srand()*** che serve a cambiare il «seme» di generazione di tali numeri. Lo standard prescrive anche che per uno stesso seme, la sequenza di numeri pseudo-casuali sia la stessa e che il seme predefinito iniziale sia pari a uno.

Nella descrizione dello standard si fa riferimento al fatto che il valore che può essere generato deve andare da zero a ‘**`RAND_MAX`**’, escludendo quindi valori negativi. Considerando che la funzione ***rand()*** restituisce un valore di tipo ‘**`int`**’ e che questo non può essere negativo, significa che ‘**`RAND_MAX`**’ deve essere inferiore o uguale al massimo numero positivo rappresentabile con il tipo ‘**`int`**’.

```
int  rand  (void);
void srand (unsigned int seed);
```

Viene proposta una soluzione molto semplice e anche molto scadente sul piano della sequenza casuale generata. Tuttavia garantisce che il valore ottenuto vada effettivamente da zero a ‘**`RAND_MAX`**’ incluso:

```
#include <stdlib.h>
static unsigned int _srand = 1; // Il rango di «_srand» deve
                                // essere maggiore o uguale
                                // a quello di «RAND_MAX»
                                // e di «unsigned int».
int
rand (void)
{
  _srand = _srand * 1234567 + 12345;
  return _srand % ((unsigned int) RAND_MAX + 1);
}

void
srand (unsigned int seed)
{
  _srand = seed;
}
```

L'esempio seguente consente di verificare sommariamente il lavoro delle funzioni per la generazione di numeri casuali. Per la precisione, si vogliono ottenere valori che vanno da 0 a 99 inclusi:

```
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
  int r;
  int i;
  int j;
  const int max = 99;

  srand (123);

  for (i = 0; i < 25; i++)
  {
```

```
      for (j = 0; j < 26; j++)
        {
          r = (rand () % (max + 1));
          printf ("%2d ", r);
        }
      printf ("\n");
  }

  return 0;
}
```

Si dovrebbe ottenere un risultato simile a quello seguente:

```
86 67 22 15 22 47 94 91 10 99  6 31 22 51 98 11 14 11 22  3
22 11 82  7 54 11 62 75 10 63 50 75 98 91 90 79 74 87 18 15
34  7 54 71 42 47 10 23 82 67 50 19 14 31 34 27 58 71 38 43
14 87 38 71 82 95 50 99  2  7 30  7 98 87  2 19 46 71 78 83
34 43 58 99 70 79 30 75 58 99 46 31 78 91 34 79 98 75 14 99
66 63 82 99  2 51 46 11 74 39 90 31 54 63 78 39 14 31 50 55
26 87  2 51 70 15 98 67 54 27 34 55 14  7 74 71 42  7 30 87
70 67 34 15 26 47 90 91 30 19 66 27 86 15 26 15  6 99 46  7
14 51 22 43 54 95 58  7 18 15 50 83 26 71  2 27 38 71 30  7
94 19  6 47 62 19 90 39 54 39 42 15 86 91 82 87 38 27 34 87
74 87 50 87 82 67 78 63 22  3 26 87 86 11 94 71 26 35 10 83
78 71 46 63 82 47 74 63 70 11 54 95 62 31 74 87 30 71 54 15
18 19 14 79 86 23 58  7 98 47 10 63 46 39 26 19 10 67 54 99
 6 47 70 11 26  7 66 23 10 51 66 31 58 91 74 87 22 35 74 11
 6 39 58 39 26 35 94 55 82 47 78 67 98 91 30 67 18 75 74 83
18 23 54 51 26 91 38 11 22 23 66 91 18 99 74 27 82 99 62 35
58 35 54 95  2 95 86 95 82 47 90 51 90 51 54 23 62 91 90 99
18 19 62 51 10 59 26 39 82 71 94 83 98 27 38  7 22 15 90 79
18 31 58 71 22  3 58 39 98 63 62 55 38 51 86 51  2  7 86 87
94 43 54 67 62 23 42 31 10  3 42 47 74 87 26 27 54 43 54 95
18 31 34 27 58 47 66 47 70 75 22 35 82  7 54  7 62 19 18 91
22 63 94 83 98 27 86 59 78 91 78 35 62 67 90 67 62 35 78 95
```

```
86 99 54 47 14  3 62 19 58 63 74 11 62 99 46 71 86  7 34  3
34 59 18  3 62 43 62 27 26 95 46  3 22 15 54 35  6 75 86 27
58 51 70 71 82 15 50 39 38 91 14 99 66 67 66 91 54 31 34  3
30 55 98 27 50  7 38 11 26 11 42 47 26 31 10 91 90 15 54 19
50 87 70 35 14 79 10 47 74 55 34 51 54 95 58 23 46 59 78 91
38  3 94 51 70  3 54 51 86 51 14 95 70 55 50 99 70 63 86 15
 6 35 98 67 74 91 54 99 30  7 94 35 10 43 54 55 50 11 58 75
26 27 78 99 98 55 94 79 26 83 34 87 38 47  6 55 74 23 10 11
26 87  6 59 10 71 86 31 78 55 30 39 66 47 46 11 30 47  2 11
74 55 42 59  2 59 42 63 78 71  2 19  6 83 30 23  2 31 54 47
62 31 26 67  6 67 70 63 14 91
```

69.9.5 Funzioni standard per la generazione di numeri pseudo-casuali

Il documento che descrive lo standard descrive una versione della funzione ***rand()*** che corrisponde al listato successivo. I valori usati nei calcoli sono tali da essere adatti a un contesto in cui i limiti degli interi sono quelli minimi previsti.

Listato 69.89. Esempio di realizzazione delle funzioni ***rand()*** e ***srand()***, tratto dal documento che descrive lo standard del linguaggio.

```
// RAND_MAX assumed to be 32767

static unsigned long int next = 1;

int
rand(void)
{
  next = next * 1103515245 + 12345;
  return (unsigned int)(next/65536) % 32768;
}
```

```
void
srand(unsigned int seed)
{
  next = seed;
}
```

69.9.6 Amministrazione della memoria

Un gruppo di funzioni dichiarate nel file ‘`strlib.h`’ consente di utilizzare dinamicamente la memoria. Si tratta di ***malloc()***, ***calloc()***, ***realloc()*** e ***free()***. Le prime tre funzioni restituiscono un puntatore di tipo ‘**`void *`**’ all’area di memoria allocata, oppure il puntatore nullo nel caso l’operazione di allocazione fallisca; la funzione ***free()*** libera un’area di memoria allocata, indicando come argomento il puntatore che inizialmente la rappresentava.

```
void *malloc  (size_t size);
void *calloc  (size_t nmemb, size_t size);
void *realloc (void *ptr, size_t size);
void free     (void *ptr);
```

Rispetto ai prototipi mostrati, la funzione ***malloc()*** richiede l’allocazione di una quantità di byte espressa dal parametro ***size***; ***calloc()*** richiede una quantità di ***nmemb*** elementi da ***size*** byte (pertanto serve solo a facilitare l’allocazione di uno spazio necessario a un array); ***realloc()*** richiede la riallocazione della memoria già allocata precedentemente a partire dall’indirizzo ***ptr*** per avere ***size*** byte, con l’intento di non perdere le informazioni precedenti (a meno che si tratti di una riduzione della dimensione); infine, ***free()*** si limita a deallocare la memoria a cui punta ***ptr***.

Nel linguaggio C, la memoria deve essere allocata e liberata espressamente, in quanto non esiste alcun sistema automatico al riguardo.

La gestione della memoria dipende strettamente dal sistema operativo, pertanto la realizzazione delle funzioni non può essere generalizzata. Per i dettagli che riguardano il comportamento di queste funzioni nel proprio sistema operativo vanno consultate le pagine di manuale *malloc(3)*, *calloc(3)*, *realloc(3)* e *free(3)*.

69.9.7 Conclusione forzata del programma

Alcune funzioni si occupano di interrompere il funzionamento del programma al di fuori della conclusione naturale della funzione ***main()***. In generale si possono distinguere i casi in cui la conclusione del programma viene gestita in modo gentile, oppure viene forzata brutalmente.

Per una conclusione corretta di un programma, è possibile predisporre un elenco di funzioni da eseguire automaticamente nel momento della conclusione. Ciò avviene attraverso la funzione ***atexit()*** che accumula un elenco di puntatori a funzione; successivamente, attraverso la chiamata alla funzione ***exit()*** si ottiene l'esecuzione delle funzioni dell'elenco, senza argomenti, secondo l'ordine di inserimento. Quindi, la funzione ***exit()*** conclude con la chiusura dei file e con la restituzione del valore passatole come argomento.

Una conclusione brutale si ottiene con la funzione ***_Exit()***, che si limita a terminare il programma, ma senza fare altro, soprattutto senza garantire che i file aperti siano chiusi correttamente.

Per ottenere una conclusione brutale del funzionamento di un programma si può usare anche la funzione ***abort()*** che però è legata alla gestione dei segnali (sezione 69.15) e qui non viene spiegato il suo utilizzo.

```
int  atexit (void (*func) (void));
void exit   (int status);
void _Exit  (int status);
void abort  (void);
```

La funzione ***atexit()*** riceve come unico argomento il puntatore a una funzione, la quale non restituisce alcun valore (di tipo '**void**') e non si attende alcun argomento (ancora il tipo '**void**'). La funzione ***atexit()*** restituisce un valore numerico da intendere come *Vero* o *Falso*, per comunicare il successo o l'insuccesso dell'operazione, dato che la quantità di puntatori a funzione che possono essere accumulati può avere un limite.

Per una descrizione completa dell'uso di queste funzioni si vedano le pagine di manuale *abort(3)*, *atexit(3)*, *exit(3)* e *_Exit(3)*.

69.9.8 Funzioni di comunicazione con l'ambiente

«

Nel file '`stdlib.h`' sono dichiarate due funzioni per interagire con il sistema operativo, ***getenv()*** e ***system()***, dove la prima consente di interrogare le variabili di ambiente (nel senso inteso nei sistemi Unix ed equivalenti) e la seconda consente di eseguire dei comandi attraverso la shell.

Nel documento che descrive lo standard del linguaggio C, il concetto viene generalizzato, ma in pratica, il contesto da cui derivano queste funzioni è quello dei sistemi Unix.

```
char *getenv (const char *name);
int   system (const char *string);
```

La funzione ***getenv()*** si aspetta di ricevere come argomento il nome di una variabile di ambiente (o di qualcosa di comparabile, nel contesto di un altro tipo di sistema operativo), restituendo il puntatore al contenuto di tale variabile. La funzione ***system()*** può essere usata indicando un puntatore nullo e in tal caso restituisce un valore diverso da zero se il sistema operativo è in grado di recepire dei comandi testuali. Se invece viene passata una stringa, la funzione tenta di farla eseguire come comando del sistema operativo: in un sistema Unix o equivalente si tratta di un comando che deve essere eseguito da '`/bin/sh`'. L'esito della funzione ***system()*** dipende da quello del comando impartito e generalmente si ottiene lo stesso valore restituito dal comando eseguito.

Si vedano le pagine di manuale *getenv(3)* e *system(3)*.

69.9.9 Funzioni di ricerca e riordino

Il file '`stdlib.h`' prevede la dichiarazione di due funzioni per il riordino degli array e per la ricerca all'interno di array ordinati. Si tratta precisamente delle funzioni ***qsort()*** e ***bsearch()***, dove i nomi richiamano evidentemente gli algoritmi tradizionali noti come *quick sort* e *binary search*.

Le funzioni della libreria standard generalizzano il problema dell'ordinamento e della ricerca utilizzando puntatori di tipo '**`void *`**' e scandendo la memoria a blocchi di una dimensione determinata. Ma dal momento che l'area di memoria da scandire non ha la personalità di un array di un qualche tipo, occorre fornire a entrambe queste

funzioni il puntatore a una funzione diversa, in grado di confrontare due valori nel contesto di proprio interesse.

```
void qsort (void *base,
            size_t nmemb,
            size_t size,
            int (*compar) (const void *, const void *));

void *bsearch (const void *key,
               const void *base,
               size_t nmemb,
               size_t size,
               int (*compar) (const void *, const void *));
```

Prima di descrivere il significato dei parametri delle due funzioni, conviene vedere un esempio in cui queste si utilizzano. Per la precisione viene scandito un piccolo array di elementi di tipo '**int**': prima viene ordinato, poi si cerca un elemento al suo interno.

```
#include <stdio.h>
#include <stdlib.h>

int confronta (const void *a, const void *b)
{
  int x = *((int *) a);
  int y = *((int *) b);
  return x - y;
}

int main (void)
{
  int a[] = {3, 1, 5, 2};
  int cercato = 5;
  void *p;

  qsort (&a[0], 4, sizeof (int), confronta);
```

```
  printf ("%i %i %i %i\n", a[0], a[1], a[2], a[3]);

  p = bsearch (&cercato, &a[0], sizeof (int), 4, confronta);

  printf ("&a[0] = %u; \"%i\" si trova in %u.\n",
          (unsigned int) &a[0], cercato, (unsigned int) p);

  return 0;
}
```

Nell'esempio viene dichiarata la funzione ***confronta()*** che riceve due argomenti e restituisce un valore che può essere: minore, pari o maggiore di zero, se il primo argomento, rispetto al secondo, è minore, pari o maggiore. Questo è il modo in cui deve comportarsi la funzione da passare come argomento a ***qsort()*** e a ***bsearch()***, tenendo conto che è da tali funzioni che riceve gli argomenti.

La funzione ***qsort()*** vuole ricevere il puntatore alla prima posizione in memoria da riordinare (il parametro ***base***), la quantità degli elementi da riordinare (***nmemb***, ovvero *Number of memory blocks*), la dimensione di tali elementi (***size***) e la funzione da usare per la loro comparazione.

La funzione ***bsearch()*** vuole ricevere il puntatore alla chiave di ricerca (il parametro ***key***), il puntatore alla prima posizione in memoria da scandire (***base***), la quantità degli elementi da scandire (***nmemb***), la dimensione di tali elementi (***size***) e la funzione da usare per la loro comparazione, tenendo conto che questa riceve la chiave di ordinamento come primo argomento.

L'esempio mostrato esegue un ordinamento crescente e il testo visualizzato che si ottiene deve essere simile a quello seguente:

```
1 2 3 5
&a[0] = 3218927260; "5" si trova in 3218927272.
```

È sufficiente invertire il risultato della funzione di comparazione per ottenere un ordinamento decrescente e per scandire un array ordinato in modo decrescente:

```
int confronta (const void *a, const void *b)
{
  int x = *((int *) a);
  int y = *((int *) b);
  return y - x;
}
```

In tal caso, il testo che viene emesso deve essere simile a quello seguente:

```
5 3 2 1
&a[0] = 3218593340; "5" si trova in 3218593340.
```

69.9.10 Funzioni per l'aritmetica con i numeri interi

«

Un gruppo di funzioni il cui nome termina per ...***abs()*** si occupa di calcolare il valore assoluto di un numero intero. Le funzioni sono precisamente: ***abs()*** per gli interi di tipo '**int**', ***labs()*** per gli interi di tipo '**long int**' e ***llabs()*** per gli interi di tipo '**long long int**'.

```
int abs             (int j);
long int labs       (long int j);
long long int llabs (long long int j);
```

Evidentemente, la realizzazione di queste funzioni è estremamente banale. Viene presentato solo il caso di ***abs()***:

```
#include <stdlib.h>
int
abs (int j)
{
  if (j < 0)
    {
      return -j;
    }
  else
    {
      return j;
    }
}
```

Un gruppo di funzioni il cui nome termina per ...***div()*** si occupa di dividere due interi, calcolando il quoziente e il resto. Le funzioni sono precisamente: ***div()*** per gli interi di tipo '**int**', ***ldiv()*** per gli interi di tipo '**long int**' e ***lldiv()*** per gli interi di tipo '**long long int**'. Il risultato viene restituito in una variabile strutturata che contiene sia il quoziente, sia il resto: rispettivamente si tratta dei tipi '**div_t**', '**ldiv_t**' e '**lldiv_t**'.

```
div_t   div   (int numer, int denom);
ldiv_t  ldiv  (long int numer, long int denom);
lldiv_t lldiv (long long int numer, long long int denom);
```

I tre tipi creati appositamente per contenere il risultato di queste funzioni contengono i membri '**quot**' e '**rem**' che rappresentano, rispettivamente, il quoziente e il resto. Anche la realizzazione di queste funzioni è molto semplice banale. Viene presentato solo il caso di ***div()***:

```
#include <stdlib.h>
div_t
div (int numer, int denom)
{
  div_t d;
  d.quot = numer / denom;
  d.rem = numer % denom;
  return d;
}
```

69.9.11 Funzioni per la gestione di caratteri estesi e sequenze multibyte

«

Il linguaggio C distingue tra una gestione dei caratteri basata sul byte, tale da consentire la gestione di un insieme minimo, come quello della codifica ASCII, e una gestione a byte multipli, o multibyte. Per esempio, la codifica UTF-8 è ciò che si intende per «multibyte», ma esistono anche altre codifiche che sfruttano questo meccanismo.

Quando il contesto richiede l'interpretazione dei byte secondo una codifica multibyte, è necessario stabilire un punto di riferimento per iniziare l'interpretazione e occorre poterne conservare lo stato quando la lettura di un carattere viene interrotta e ripresa a metà. Nella documentazione dello standard, nell'ambito delle sequenze multibyte, lo stato viene definito *shift state*.

Per gestire internamente la codifica universale, il C utilizza un tipo specifico, '**`wchar_t`**', corrispondente a un intero di rango sufficiente a rappresentare tutti i caratteri che si intendono gestire. Di conseguenza, le stringhe letterali, precedute dalla lettera '**`L`**' (per esempio '**`L"àèìòùé"`**'), sono array di elementi '**`wchar_t`**'.

Le funzioni che riguardano la gestione di caratteri estesi e sequen-

ze multibyte del file ‘`stdlib.h`’, servono principalmente per convertire sequenze multibyte nel tipo ‘**`wchar_t`**’ e viceversa. Tuttavia, occorre tenere presente che la configurazione locale deve essere tale da prevedere l’uso di caratteri da rappresentare attraverso sequenze multibyte, altrimenti le conversioni diventano prive di utilità.

Listato 69.102. Prototipi delle funzioni relative alla gestione multibyte.

```
int    mblen    (const char *s, size_t n);
int    mbtowc   (wchar_t *restrict pwc,
                 const char *restrict s,
                 size_t n);
int    wctomb   (char *s, wchar_t wc);
size_t mbstowcs (wchar_t *restrict pwcs,
                 const char *restrict s,
                 size_t n);
size_t wcstombs (char *restrict s,
                 const wchar_t *restrict pwcs,
                 size_t n);
```

69.9.12 Funzione «mblen()»

La funzione ***mblen()*** si usa normalmente per contare quanti byte sono presenti nella stringa ***s*** fornita come primo argomento, per comporre il primo carattere (multibyte) della stringa stessa, limitando la scansione a un massimo di ***n*** byte (il secondo argomento richiesto). Se al posto di indicare una stringa si fornisce il puntatore nullo, si ottiene un valore che può essere uno o zero, a seconda che sia prevista o meno una codifica multibyte con una gestione dello stato (*shift state*).

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
```

```
int main (void)
{
  int n;
  setlocale (LC_ALL, "en_US.UTF-8");
  n = mblen (NULL, 0);
  printf ("Gestione dello stato: %i\n", n);
  n = mblen ("€", 8);
  printf ("Il carattere %s richiede %i byte.\n", "€", n);
  return 0;
}
```

L'esempio mostrato dovrebbe chiarire alcune cose. La funzione richiede un argomento di tipo stringa di caratteri, perché un argomento di tipo '**`char`**' singolo, non consentirebbe di annotare una sequenza multibyte. Le sequenze multibyte sono stringhe normali, trattate come tali, salvo quando è necessario interpretare il loro contenuto; a questo proposito, si vede che la funzione ***printf()*** riceve una stringa multibyte e si limita a trattarla come una stringa normale.

Se la codifica in cui è scritto il sorgente è la stessa usata dal programma durante il suo funzionamento, si può ottenere il testo seguente:

```
Gestione dello stato: 0
Il carattere € richiede 3 byte.
```

Nell'uso normale della funzione ***mblen()***, se la stringa che si fornisce contiene una sequenza multibyte errata o incompleta, il valore restituito è -1.

69.9.13 Funzioni «mbtowc()» e «wctomb()»

Le due funzioni ***mbtowc()*** e ***wctomb()*** si compensano a vicenda, fornendo il mezzo elementare di conversione dei caratteri da una sequenza multibyte a un numero intero di tipo '**wchar_t**' e viceversa. Quando a queste funzioni, al posto del puntatore alla stringa multibyte, si fornisce il puntatore nullo, si ottiene un funzionamento analogo a quello di ***mblen()***, con un valore pari a uno se la configurazione locale prevede l'uso di sequenze multibyte e una gestione dello stato, oppure zero se questo problema non sussiste. Inoltre, per entrambe le funzioni, se la sequenza multibyte è errata o incompleta, si ottiene la restituzione del valore -1. Infine, se la conversione ha successo, si ottiene la quantità dei byte che compongono la sequenza multibyte (di origine o di destinazione, a seconda della funzione usata).

Viene mostrato un esempio molto semplice che dimostra l'uso delle due funzioni. In particolare viene convertita la sequenza multibyte che rappresenta la lettera «ä» in un numero '**wchar_t**', quindi il numero viene incrementato e riconvertito in una nuova sequenza multibyte, per ottenere il carattere «å».

```
#include <locale.h>
#include <stdlib.h>
#include <stdio.h>

int main (void)
{
  int n;
  wchar_t wc;
  char mb[20] = {}; // Inizializza l'array a zero.

  setlocale (LC_ALL, "en_US.UTF-8");
```

```
  n = mbtowc (&wc, NULL, 8);
  printf ("Gestione dello stato: %i\n", n);
  n = wctomb (NULL, wc);
  printf ("Gestione dello stato: %i\n", n);

  n = mbtowc (&wc, "ä", 8);
  printf ("Il carattere \"ä\" si rappresenta con %i byte ",
          n);
  printf ("in una sequenza multibyte e con il numero %i ",
          (int) wc);
  printf ("in una variabile di tipo \"wchar_t\".\n");

  wc++;
  n = wctomb (mb, wc);
  printf ("Il carattere \"%s\" si rappresenta con %i byte ",
          mb, n);
  printf ("in una sequenza multibyte e con il numero %i ",
          (int) wc);
  printf ("in una variabile di tipo \"wchar_t\".\n");

  return 0;
}
```

Si dovrebbe ottenere un testo come quello seguente:

```
Gestione dello stato: 0
Gestione dello stato: 0
Il carattere "ä" si rappresenta con 2 byte in una ↩
↪sequenza multibyte e con il numero 228 ↩
↪in una variabile di tipo "wchar_t".
Il carattere "å" si rappresenta con 2 byte in una ↩
↪sequenza multibyte e con il numero 229 ↩
↪in una variabile di tipo "wchar_t".
```

Per gli approfondimenti eventuali, si vedano le pagine di manuale

mbtowc(3) e *wctomb(3)*.

69.9.14 Funzioni «mbstowcs()» e «wcstombs()»

Le funzioni ***mbstowcs()*** e '**`wcstombs`**' servono rispettivamente per convertire una stringa multibyte in un stringa estesa (un array di elementi '**`wchar_t`**') e per fare l'opposto. Entrambe le funzioni richiedono tre argomenti: l'array di destinazione, l'array di origine e la quantità di elementi da utilizzare nell'array di destinazione. Entrambe le funzioni restituiscono un numero che esprime la quantità di elementi di destinazione convertiti, escluso ciò che costituisce il carattere nullo di terminazione. Entrambe restituiscono un valore pari a '**`(size_t) (-1)`**' se la conversione produce un errore.[3]

Per convertire correttamente una stringa (multibyte o estesa), occorre che il numero di elementi di destinazione previsto includa anche il carattere nullo di terminazione. L'esempio seguente dovrebbe aiutare a comprendere il problema:

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
  size_t n;
  wchar_t wca[] = {1, 2, 3, 4, 5, 6, 7};
  wchar_t wcb[] = {1, 2, 3, 4, 5, 6, 7};
  char mba[] = "**************";
  char mbb[] = "**************";

  setlocale (LC_ALL, "en_US.UTF-8");

  n = mbstowcs (wca, "äåâ", 3);
```

```
  printf ("mbstowcs: %i: %i %i %i %i %i\n", n,
          wca[0], wca[1], wca[2], wca[3], wca[4]);

  n = mbstowcs (wcb, "äåâ", 5);
  printf ("mbstowcs: %i: %i %i %i %i %i\n", n,
          wcb[0], wcb[1], wcb[2], wcb[3], wcb[4]);

  n = wcstombs (mba, L"äåâ", 6);
  printf ("wcstombs: %i: \"%s\"\n", n, mba);

  n = wcstombs (mbb, L"äåâ", 9);
  printf ("wcstombs: %i: \"%s\"\n", n, mbb);

  return 0;
}
```

Nell'esempio, la funzione ***mbstowcs()*** viene usata due volte, per convertire una stringa multibyte, composta da tre caratteri, se non si conta quello di terminazione. Nel primo caso, viene specificato che si vogliono convertire esattamente tre caratteri, ma questo significa che nell'array di destinazione rimane il contenuto originale a partire dal quarto elemento. In modo analogo, la funzione ***wcstombs()*** viene usata due volte per convertire una stringa estesa in una stringa multibyte. La stringa estesa si compone di tre caratteri che nella conversione vanno a occupare esattamente sei byte, con l'aggiunta eventuale del carattere nullo di terminazione (che sarebbe il settimo). Si può vedere che quando si chiede una conversione di sei elementi, la stringa ricevente mantiene il contenuto precedente nella parte restante. Ecco cosa si dovrebbe vedere eseguendo il programma:

```
mbstowcs: 3: 228 229 226 4 5
mbstowcs: 3: 228 229 226 0 5
wcstombs: 6: "äåâ********"
wcstombs: 6: "äåâ"
```

Se al posto della destinazione (il primo argomento) viene posto il puntatore nullo, si ottiene la simulazione dell'operazione, senza memorizzare alcunché e senza tenere conto della quantità massima di elementi che si annota come ultimo argomento. Ciò ha lo scopo di contare quanti elementi servirebbero per produrre una conversione completa. L'esempio seguente modifica quello già visto, sfruttando questa funzionalità:

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
  size_t n;
  size_t max;
  wchar_t wca[] = {1, 2, 3, 4, 5, 6, 7};
  char mba[] = "**************";      // 15 byte total.

  setlocale (LC_ALL, "en_US.UTF-8");

  max = mbstowcs (NULL, "äåâ", 0);
  if (max <= 6)
    {
      n = mbstowcs (wca, "äåâ", max + 1);
      printf ("mbstowcs: %i: %i %i %i %i %i\n", n,
              wca[0], wca[1], wca[2], wca[3], wca[4]);
    }
```

```
  max = wcstombs (NULL, L"äåâ", 0);
  if (max <= 14)
    {
      n = wcstombs (mba, L"äåâ", max + 1);
      printf ("wcstombs: %i: \"%s\"\n", n, mba);
    }

  return 0;
}
```

Ecco cosa si dovrebbe vedere eseguendo il programma:

```
mbstowcs: 3: 228 229 226 0 5
wcstombs: 6: "äåâ"
```

Per gli approfondimenti eventuali, si vedano le pagine di manuale *mbstowcs(3)* e *wcstombs(3)*.

69.10 File «inttypes.h»

«

Il file ‘`inttypes.h`’ della libreria standard serve principalmente a completare le funzionalità di ‘`stdint.h`’, per ciò che riguarda la gestione dei valori numerici interi, il cui rango è controllabile. Infatti, il problema principale nell’uso di interi definiti in modo alternativo allo standard del linguaggio C, privo di librerie, sta nell’uso appropriato degli specificatori di conversione nelle funzioni come ***printf()*** e ***scanf()***. È proprio per risolvere questo problema che nel file ‘`inttypes.h`’ vanno definite, soprattutto, delle macro-variabili da usare in sostituzione degli specificatori di conversione basati sui tipi elementari (si veda eventualmente la realizzazione del file ‘`inttypes.h`’, ma senza le funzioni che lo riguardano, nei sorgenti di os32, listato 95.8).[4]

Gli esempi proposti per descrivere la libreria che fa capo al file ‘`inttypes.h`’ si riferiscono a quanto già definito nella sezione 69.4 a proposito del file ‘`stdint.h`’.

Inizialmente, il file ‘`inttypes.h`’ deve includere ‘`stdint.h`’, inoltre dichiara il tipo ‘**`wchar_t`**’, già descritto nel file ‘`stddef.h`’:

```
#include <stdint.h>
typedef unsigned int wchar_t;
```

69.10.1 Divisione intera con interi di rango massimo

Nel file ‘`inttypes.h`’ viene definito il tipo ‘**`imaxdiv_t`**’ che va ad affiancarsi ai tipi ‘**`div_t`**’, ‘**`ldiv_t`**’ e ‘**`ldiv_t`**’, definiti nel file ‘`stdlib.h`’. In pratica si tratta di una struttura il cui scopo è quello di contenere il risultato di una divisione, espresso come quoziente e resto, quando il tipo intero usato è quello massimo:

```
typedef struct {
    intmax_t quot;
    intmax_t rem;
} imaxdiv_t;
```

Il tipo strutturato ‘**`imaxdiv_t`**’ serve alle funzioni ***imaxdiv()*** e ***uimaxdiv()***, le quali sono sostanzialmente equivalenti alle altre funzioni ***...div()*** del file ‘`stdlib.h`’:

```
imaxdiv_t imaxdiv (intmax_t numer, intmax_t denom);
```

```
#include <inttypes.h>
imaxdiv_t
imaxdiv   (intmax_t numer, intmax_t denom)
{
  imaxdiv_t d;
  d.quot = numer / denom;
  d.rem = numer % denom;
  return d;
}
```

69.10.2 Macro-variabili in qualità di specificatori di conversione

«

Come accennato all'inizio del capitolo, per poter usare le funzioni *...printf()* e *...scanf()*, occorrono degli specificatori di conversione, ma non ne esistono per i tipi interi a rango controllato, pertanto, per questi, servono delle macro-variabili coerenti con il tipo relativo.

Le macro che iniziano per ***PRI...*** si usano come parte terminale di specificatori di conversione per la composizione dell'output (*...printf()*), mentre le macro che iniziano per ***SCN...*** sono adatte per l'interpretazione dell'input (*...scanf()*).

Le macro '**PRI*x*n**' e '**SCN*x*n**' terminano gli specificatori di conversione '**%...*x***', per i tipi interi '**[u]int*n*_t**'; le macro '**PRI*x*LEAST*n***' e '**SCN*x*LEAST*n***' riguardano i tipi '**[u]int_least*n*_t**'; le macro '**PRI*x*FAST*n***' e '**SCN*x*FAST*n***' riguardano i tipi '**[u]int_fast*n*_t**'; le macro '**PRI*x*MAX*n***' e '**SCN*x*MAX*n***' riguardano i tipi '**[u]intmax_t**'; le macro '**PRI*x*PTR*n***' e '**SCN*x*PTR*n***' riguardano i tipi '**[u]intptr_t**'.

L'esempio seguente dovrebbe dimostrare il significato di queste macro-variabili, attraverso l'uso di ***printf()***:

```
#include <stdio.h>
#include <inttypes.h>
int
main (int argc, char *argv[])
{
  uint64_t num = INT64_C(1234567890);
  printf ("Il valore della variabile \"num\" "
          "corrisponde a "
          "%020" PRIu64 ".\n", num);
  return 0;
}
```

Il listato seguente mostra come possono essere dichiarate queste macro-variabili:

```
// Composizione dell'output.

#define PRId8           "d"
#define PRId16          "d"
#define PRId32          "d"
#define PRId64          "lld"

#define PRIdLEAST8      "d"
#define PRIdLEAST16     "d"
#define PRIdLEAST32     "d"
#define PRIdLEAST64     "lld"

#define PRIdFAST8       "d"
#define PRIdFAST16      "d"
#define PRIdFAST32      "d"
#define PRIdFAST64      "lld"

#define PRIdMAX         "lld"
```

```
#define PRIdPTR         "d"

#define PRIi8           "i"
#define PRIi16          "i"
#define PRIi32          "i"
#define PRIi64          "lli"

#define PRIiLEAST8      "i"
#define PRIiLEAST16     "i"
#define PRIiLEAST32     "i"
#define PRIiLEAST64     "lli"

#define PRIiFAST8       "i"
#define PRIiFAST16      "i"
#define PRIiFAST32      "i"
#define PRIiFAST64      "lli"

#define PRIiMAX         "lli"
#define PRIiPTR         "i"

#define PRIo8           "o"
#define PRIo16          "o"
#define PRIo32          "o"
#define PRIo64          "llo"

#define PRIoLEAST8      "o"
#define PRIoLEAST16     "o"
#define PRIoLEAST32     "o"
#define PRIoLEAST64     "llo"

#define PRIoFAST8       "o"
#define PRIoFAST16      "o"
#define PRIoFAST32      "o"
#define PRIoFAST64      "llo"
```

```
#define PRIoMAX         "llo"
#define PRIoPTR         "o"

#define PRIu8           "u"
#define PRIu16          "u"
#define PRIu32          "u"
#define PRIu64          "llu"

#define PRIuLEAST8      "u"
#define PRIuLEAST16     "u"
#define PRIuLEAST32     "u"
#define PRIuLEAST64     "llu"

#define PRIuFAST8       "u"
#define PRIuFAST16      "u"
#define PRIuFAST32      "u"
#define PRIuFAST64      "llu"

#define PRIuMAX         "llu"
#define PRIuPTR         "u"

#define PRIx8           "x"
#define PRIx16          "x"
#define PRIx32          "x"
#define PRIx64          "llx"

#define PRIxLEAST8      "x"
#define PRIxLEAST16     "x"
#define PRIxLEAST32     "x"
#define PRIxLEAST64     "llx"

#define PRIxFAST8       "x"
#define PRIxFAST16      "x"
```

```
#define PRIxFAST32      "x"
#define PRIxFAST64      "llx"

#define PRIxMAX         "llx"
#define PRIxPTR         "x"

#define PRIX8           "X"
#define PRIX16          "X"
#define PRIX32          "X"
#define PRIX64          "llX"

#define PRIXLEAST8      "X"
#define PRIXLEAST16     "X"
#define PRIXLEAST32     "X"
#define PRIXLEAST64     "llX"

#define PRIXFAST8       "X"
#define PRIXFAST16      "X"
#define PRIXFAST32      "X"
#define PRIXFAST64      "llX"

#define PRIXMAX         "llX"
#define PRIXPTR         "X"

// Interpretazione dell'input.

#define SCNd8           "hhd"
#define SCNd16          "hd"
#define SCNd32          "d"
#define SCNd64          "lld"

#define SCNdLEAST8      "hhd"
#define SCNdLEAST16     "hd"
#define SCNdLEAST32     "d"
```

```
#define SCNdLEAST64     "lld"

#define SCNdFAST8       "hhd"
#define SCNdFAST16      "d"
#define SCNdFAST32      "d"
#define SCNdFAST64      "lld"

#define SCNdMAX         "lld"
#define SCNdPTR         "d"

#define SCNi8           "hhi"
#define SCNi16          "hi"
#define SCNi32          "i"
#define SCNi64          "lli"

#define SCNiLEAST8      "hhi"
#define SCNiLEAST16     "hi"
#define SCNiLEAST32     "i"
#define SCNiLEAST64     "lli"

#define SCNiFAST8       "hhi"
#define SCNiFAST16      "i"
#define SCNiFAST32      "i"
#define SCNiFAST64      "lli"

#define SCNiMAX         "lli"
#define SCNiPTR         "i"

#define SCNo8           "hho"
#define SCNo16          "ho"
#define SCNo32          "o"
#define SCNo64          "llo"

#define SCNoLEAST8      "hho"
```

```
#define SCNoLEAST16     "ho"
#define SCNoLEAST32     "o"
#define SCNoLEAST64     "llo"

#define SCNoFAST8       "hho"
#define SCNoFAST16      "o"
#define SCNoFAST32      "o"
#define SCNoFAST64      "llo"

#define SCNoMAX         "llo"
#define SCNoPTR         "o"

#define SCNu8           "hhu"
#define SCNu16          "hu"
#define SCNu32          "u"
#define SCNu64          "llu"

#define SCNuLEAST8      "hhu"
#define SCNuLEAST16     "hu"
#define SCNuLEAST32     "u"
#define SCNuLEAST64     "llu"

#define SCNuFAST8       "hhu"
#define SCNuFAST16      "u"
#define SCNuFAST32      "u"
#define SCNuFAST64      "llu"

#define SCNuMAX         "llu"
#define SCNuPTR         "u"

#define SCNx8           "hhx"
#define SCNx16          "hx"
#define SCNx32          "x"
#define SCNx64          "llx"
```

```
#define SCNxLEAST8      "hhx"
#define SCNxLEAST16     "hx"
#define SCNxLEAST32     "x"
#define SCNxLEAST64     "llx"

#define SCNxFAST8       "hhx"
#define SCNxFAST16      "x"
#define SCNxFAST32      "x"
#define SCNxFAST64      "llx"

#define SCNxMAX         "llx"
#define SCNxPTR         "x"
```

69.10.3 Valore assoluto

Nel file ‘`stdlib.h`’ si trovano dichiarate alcune funzioni per il calcolo del valore assoluto: ...***abs()***. Nel file ‘`inttypes.h`’ si aggiunge la funzione ***imaxabs()***, da usare per i valori interi massimi:

```
intmax_t imaxabs (intmax_t j);
```

```
#include <inttypes.h>
intmax_t
imaxabs (intmax_t j)
{
  if (j < 0)
    {
      return -j;
    }
  else
    {
      return j;
    }
}
```

69.10.4 Conversione da stringa a numero intero

«

Per convertire una stringa contenente un valore numerico intero, quando si vuole fare riferimento all'intero di dimensione massima, si possono usare le funzioni ***strtoimax()***, ***strtouimax()***, ***wcstoimax()*** e ***wcstouimax()***, dichiarate nel file '`inttypes.h`'. Come il nome suggerisce, le prime due funzioni sono destinate alla conversione di stringhe «normali», mentre le altre sono specifiche per le stringhe estese.

Evidentemente si tratta di funzioni che si abbinano alle altre ***strto...()*** del file '`stdlib.h`' e alle funzioni ***wcsto...()*** del file '`wchar.h`'.

```
intmax_t  strtoimax  (const char *restrict nptr,
                      char **restrict endptr, int base);
uintmax_t strtouimax (const char *restrict nptr,
                      char **restrict endptr, int base);
intmax_t  wcstoimax  (const wchar_t *restrict nptr,
                      wchar_t **restrict endptr, int base);
uintmax_t wcstouimax (const wchar_t *restrict nptr,
                      wchar_t **restrict endptr, int base);
```

Come si vede, i parametri delle funzioni sono gli stessi; quello che cambia è il tipo di stringa, che nelle funzioni ***strto**...()* è normale, mentre nelle funzioni ***wcsto**...()* è di tipo esteso. Nel caso di funzioni ...***touimax()*** si ottiene un valore intero senza segno, mentre con le funzioni ...***toimax()*** si ottiene un valore intero con segno.

Il comportamento di queste funzioni è analogo a quello delle altre funzioni ***strto**...()* e ***wcsto**...()*, per ciò che riguarda l'interpretazione di valori interi, con la differenza che si fa riferimento al valore intero più grande. Il valore restituito è zero se non si può procedere alla conversione; se invece il valore è al di fuori dell'intervallo rappresentabile, a seconda dei casi si può avere il valore corrispondente a ***INTMAX_MAX***, ***INTMAX_MIN*** o ***UINTMAX_MIN***, con l'aggiornamento della variabile ***errno*** al valore rappresentato da ***ERANGE***.

69.11 File «iso646.h»

«

Il file '`iso646.h`' della libreria standard definisce alcune macrovariabili da usare in sostituzione di simboli che potrebbero mancare nella propria tastiera, anche se ciò è comunque poco probabile.[5]

Macro	Corrispondenza	Codice
and	&&	#define and &&
and_eq	&=	#define and_eq &=
bitand	&	#define bitand &
bitor	|	#define bitor |
compl	~	#define compl ~
not	!	#define not !
not_eq	!=	#define not_eq !=
or	||	#define or ||
or_eq	|=	#define or_eq |=
xor	^	#define xor ^
xor_eq	^=	#define xor_eq ^=

69.12 File «stdbool.h»

«

Il file ‘`stdbool.h`’ della libreria standard definisce alcune macro-variabili da usare per la gestione dei valori logici (*Vero* e *Falso*); in particolare consente di utilizzare il nome ***bool*** al posto di ‘**`_Bool`**’ (si veda eventualmente la realizzazione di questo file nei sorgenti di os32, listato 95.1.11).[6].

```
#define bool    _Bool
#define true    1
#define false   0
#define __bool_true_false_are_defined   1
```

Come si può vedere, la macro-variabile ***__bool_true_false_are_defined*** consente di sapere se le macro-variabili ***bool***, ***true*** e ***false***, sono definite.

69.13 File «stddef.h»

«

Il file ‘`stddef.h`’ della libreria standard definisce alcuni tipi di dati e delle macro fondamentali (si veda eventualmente la realizzazione del file ‘`stddef.h`’ nei sorgenti di os32, listato 95.1.12).[7]

```
typedef long int          ptrdiff_t;
typedef unsigned long int size_t;
typedef unsigned int      wchar_t;

#define NULL                        0
#define offsetof(TYPE, MEMBER)   \
    ((size_t) &((TYPE *)0)->MEMBER)
```

Di tutte le definizioni merita attenzione la macroistruzione ‘**`offsetof`**’ che serve a misurare lo scostamento di un membro di una struttura, per la quale è il caso di scomporre i suoi componenti:

- l’espressione ‘**`((`*`tipo_struttura`*` *`**`)0)`**’ rappresenta un puntatore nullo trasformato, con un cast, in un puntatore nullo al tipo di struttura alla quale si sta facendo riferimento;
- l’espressione ‘**`((`*`tipo_struttura`*` *`**`)0)->`*`nome_membro`*’ rappresenta il contenuto del membro indicato, preso a partire dall’indirizzo zero;

- l'espressione '**`&((`*tipo_struttura*` *`)0)->`*nome_membro***' rappresenta l'indirizzo del membro indicato, preso a partire dall'indirizzo zero.

Pertanto, l'indirizzo del membro, relativo all'indirizzo zero, corrisponde anche al suo scostamento a partire dall'inizio della struttura. Così, tale valore viene convertito con un cast nel tipo '**`size_t`**'.

69.14 File «string.h»

«

Il file '`string.h`' della libreria standard definisce il tipo '**`size_t`**', la macro-variabile ***NULL*** (come dal file '`stddef.h`', descritto nella sezione 69.13) e una serie di funzioni per il trattamento delle stringhe o comunque di sequenze di caratteri (si veda eventualmente la realizzazione del file '`string.h`' e di alcune delle sue funzioni nei sorgenti di os32, sezione 95.20)).

69.14.1 Copia

«

Seguono i prototipi delle funzioni disponibili per la copia:

```
void *memcpy  (void *restrict dst,
               const void *restrict org, size_t n);
void *memmove (void *dst, const void *org, size_t n);
char *strcpy  (char *restrict dst,
               const char *restrict org);
char *strncpy (char *restrict dst,
               const char *restrict org, size_t n);
```

POSIX Lo standard POSIX aggiunge anche i prototipi seguenti:

```
void *memccpy (void *restrict dst, const void *restrict org,
               int c, size_t n);
char *strdup  (const char *org);
```

69.14.1.1 Funzione «memcpy()» (memory copy)

La funzione ***memcpy()*** copia ***n*** caratteri a partire dall'indirizzo indicato da ***org***, per riprodurli a partire dall'indirizzo ***dst***, alla condizione che i due insiemi non risultino sovrapposti. La funzione restituisce l'indirizzo ***dst***.

```
#include <string.h>
void *
memcpy (void *restrict dst, const void *restrict org,
        size_t n)
{
  unsigned char *d = (unsigned char *) dst;
  unsigned char *o = (unsigned char *) org;
  size_t i;
  for (i = 0; n > 0 && i < n; i++)
    {
      d[i] = o[i];
    }
  return dst;
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, dove la variabile ***y*** viene sovrascritta dal contenuto di ***x***, ma questo attraverso la copia dei byte (si intende che gli interi siano da 32 bit).

```
#include <stdio.h>
#include <string.h>

int
main (void)
{
  int  x = 0x12345678;
  int  y = 0xFFFFFFFF;
```

```
  printf ("prima: %x\n", y);
  memcpy (&y, &x, sizeof (int));
  printf ("dopo:  %x\n", y);
  return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
prima: ffffffff
dopo:  12345678
```

69.14.1.2 Funzione «memccpy()»

La funzione ***memccpy()*** appartiene allo standard POSIX e si distingue rispetto a ***memcpy()*** perché la copia termina al raggiungimento di un certo carattere (il parametro ***c***), restituendo il puntatore alla posizione successiva nella destinazione.

POSIX

```
#include <string.h>
void *
memccpy (void *restrict dst, const void *restrict org,
         int c, size_t n)
{
  unsigned char *d = (unsigned char *) dst;
  unsigned char *o = (unsigned char *) org;
  size_t i;
  for (i = 0; n > 0 && i < n; i++)
    {
      d[i] = o[i];
      if (o[i] == (unsigned char) c)
        {
          return (dst + i + 1);
        }
```

```
    }
  return NULL;
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, dove l'array ***y*** viene sovrascritto dal contenuto di ***x***, fino a quando si raggiunge il carattere '**7**'. Nell'esempio vengono anche visualizzati i valori dei puntatori di ***y*** e della posizione raggiunta all'interno di ***y*** alla fine della copia, sempre in forma di puntatore.

```
#include <stdio.h>
#include <string.h>
#include <inttypes.h>

int
main (void)
{
  char x[11] = { '0', '1', '2', '3', '4', '5',
                 '6', '7', '8', '9', '\0'};
  char y[11] = { 'a', 'b', 'c', 'd', 'e', 'f',
                 'g', 'h', 'i', 'j', '\0'};
  char *z;
  printf ("prima: \"%s\" %" PRIuPTR "\n", y, (uintptr_t) y);
  z = memccpy (y, x, (int) '7', (size_t) 11);
  printf ("dopo:  \"%s\" %" PRIuPTR "\n", y, (uintptr_t) z);
  return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
prima: "abcdefghij" 3218609914
```

```
dopo:  "01234567ij" 3218609922
```

Come si può vedere dal risultato, l'array ***y*** inizia a partire dal puntatore 3218609914 e, alla fine del trasferimento parziale dall'array ***x***, che si ferma al carattere '**7**', la funzione restituisce il puntatore 3218609922 che individua la posizione successiva al carattere copiato, corrispondente in pratica al carattere '**i**'.

69.14.1.3 Funzione «memmove()» (memory move)

La funzione ***memmove()*** opera in modo simile a ***memcpy()***, con la differenza che le due aree di memoria coinvolte possono sovrapporsi. In pratica la copia avviene prima in un'area temporanea, quindi, dall'area temporanea viene ricopiata nella destinazione. La funzione restituisce l'indirizzo ***dst***.

```
#include <string.h>
void *
memmove (void *dst, const void *org, size_t n)
{
  char temp[n];
  unsigned char *d = (unsigned char *) dst;
  unsigned char *o = (unsigned char *) org;
  size_t i;
  for (i = 0; i < n; i++)
    {
      temp[i] = o[i];
    }
  for (i = 0; n > 0 && i < n; i++)
    {
      d[i] = temp[i];
    }
  return dst;
}
```

Per osservare il comportamento della funzione si può riutilizzare lo stesso programma usato per ***memcpy()***, con la modifica del nome della funzione chiamata. Il risultato atteso è lo stesso:

```
…
  printf ("prima: %x\n", y);
  memmove (&y, &x, sizeof (int));
  printf ("dopo:  %x\n", y);
…
```

69.14.1.4 Funzione «strcpy()» (string copy)

La funzione ***strcpy()*** copia la stringa ***org*** nell'array a cui punta ***dst***, includendo anche il carattere zero di conclusione delle stringhe, alla condizione che le due stringhe non si sovrappongano. La funzione restituisce ***dst***.

```
#include <string.h>
char *
strcpy (char *restrict dst, const char *restrict org)
{
  size_t i;
  for (i = 0; org[i] != 0; i++)
    {
      dst[i] = org[i];
    }
  dst[i] = 0;
  return dst;
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, dove l'array ***y*** viene sovrascritto dal contenuto di ***x***.

```
#include <stdio.h>
#include <string.h>
int main (void)
{
  char x[] = "abcdefghijklmnopqrstuvwxyz";
  char y[50] = "ciao";
  printf ("prima: %s\n", y);
  strcpy (y, x);
  printf ("dopo:  %s\n", y);
  return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
prima: ciao
dopo:  abcdefghijklmnopqrstuvwxyz
```

69.14.1.5 Funzione «strncpy()»

«

La funzione ***strncpy()*** agisce in modo analogo a quello di '**`strcpy`**', con la differenza che la copia riguarda al massimo i primi ***n*** caratteri, includendo in questo anche il carattere nullo di terminazione delle stringhe. Se però la stringa ***org*** è più breve (in quanto si incontra il carattere di terminazione prima di ***n*** caratteri), i caratteri rimasti vengono copiati con un valore a zero nella destinazione. La funzione restituisce ***dst***.

```
#include <string.h>
char *
strncpy (char *restrict dst, const char *restrict org,
         size_t n)
{
  size_t i;
```

```
  for (i = 0; n > 0 && i < n && org[i] != 0; i++)
    {
      dst[i] = org[i];
    }
  for ( ; n > 0 && i < n; i++)
    {
      dst[i] = 0;
    }
  return dst;
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, dove l'array ***y*** viene sovrascritto dal contenuto di ***x***.

```
#include <stdio.h>
#include <string.h>
int main (void)
{
  char x[] = "abcdefghijklmnopqrstuvwxyz";
  char y[50] = "ciaociaociaociaociaociaociaociao";
  printf ("prima:   %s\n", y);
  strncpy (y, x, 10);
  printf ("durante: %s\n", y);
  strncpy (y, x, 27);
  printf ("dopo:    %s\n", y);
  return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
prima:   ciaociaociaociaociaociaociaociao
durante: abcdefghijaociaociaociaociaociao
```

```
dopo:    abcdefghijklmnopqrstuvwxyz
```

69.14.1.6 Funzione «strdup()»

La funzione ***strdup()***, richiesta dallo standard POSIX, è simile a ***strcpy()***, con la differenza che richiede solo l'indicazione della stringa da duplicare, mentre alloca autonomamente la memoria per produrne una copia. Pertanto, la funzione restituisce il puntatore alla stringa duplicata (e allocata) ed è poi possibile liberare la memoria attraverso la funzione ***free()***.

```
#include <string.h>
#include <stdlib.h>
#include <stdint.h>

char *
strdup (const char *org)
{
  char  *d;
  int    i;
  size_t size;

  for (i = 0; i < (SIZE_MAX - 1); i++)
    {
      if (org[i] == '\0')
        {
          break;
        }
    }

  if (i == (SIZE_MAX - 1))
    {
      return NULL;
    }
```

```
  size = i + 1;

  d = malloc (size);

  if (d == NULL)
    {
      return NULL;
    }

  strcpy (d, org);

  return d;
}
```

Nell'esempio proposto, si riutilizza la funzione ***strcpy()*** e la funzione ***malloc()*** per allocare la memoria necessaria. Viene anche usata la macro-variabile ***SIZE_MAX***, per dare un limite massimo alla scansione, nel caso la stringa di origine non contenga il carattere nullo di terminazione. Per questa ragione, diventa necessario includere i file 'stdint.h' e 'stdlib.h'.

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, dove l'array ***o*** viene copiato altrove, associandogli il puntatore ***d***: se l'operazione ha successo, viene visualizzata la stringa a cui punta ***d***, altrimenti si ottiene un messaggio di errore. Dovendo usare la funzione ***free()*** per liberare la memoria, si include anche il file 'stdlib.h'.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

```
int
main (void)
{
  char *d;
  char  o[] = "abcdefghijklmnopqrstuvwxyz";

  d = strdup (o);

  if (d == NULL)
    {
      printf ("Non è possibile duplicare la stringa!\n");
    }
  else
    {
      printf ("%s\n", d);
      free (d);
    }
  return 0;
}
```

69.14.2 Concatenamento

«

Seguono i prototipi delle funzioni per il concatenamento:

```
char *strcat  (char *restrict dst,
               const char *restrict org);
char *strncat (char *restrict dst,
               const char *restrict org, size_t n);
```

69.14.2.1 Funzione «strcat()» (string cat)

La funzione ***strcat()*** copia la stringa ***org*** a partire dalla fine della stringa ***dst*** (sovrascrivendo il carattere nullo preesistente), alla condizione che le due stringhe non siano sovrapposte. La funzione restituisce ***dst***.

```
#include <string.h>
char *
strcat (char *restrict dst, const char *restrict org)
{
  size_t i;
  size_t j;
  for (i = 0; dst[i] != 0; i++)
    {
      ; // Si limita a cercare il carattere nullo.
    }
  for (j = 0; org[j] != 0; i++, j++)
    {
      dst[i] = org[j];
    }
  dst[i] = 0;
  return dst;
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, dove la stringa ***y*** viene estesa con il contenuto di ***x***.

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  char x[] = "abcdefghijklmnopqrstuvwxyz";
```

```
  char y[50] = "ciao";
  printf ("prima: %s\n", y);
  strcat (y, x);
  printf ("dopo: %s\n", y);
  return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
prima: ciao
dopo: ciaoabcdefghijklmnopqrstuvwxyz
```

69.14.2.2 Funzione «strncat()»

La funzione ***strncat()*** si comporta in modo analogo a ***strcat()***, con la differenza che copia al massimo ***n*** caratteri, ammesso che la stringa ***org*** ne contenga abbastanza. In ogni caso, la stringa ***dst*** viene completata con il carattere nullo di terminazione.

```
#include <string.h>
char *
strncat (char *restrict dst, const char *restrict org,
         size_t n)
{
  size_t i;
  size_t j;
  for (i = 0; n > 0 && dst[i] != 0; i++)
    {
      ; // Si limita a cercare il carattere nullo.
    }
  for (j = 0; n > 0 && j < n && org[j] != 0; i++, j++)
    {
      dst[i] = org[j];
```

```
    }
  dst[i] = 0;
  return dst;
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, dove la stringa ***y*** viene estesa con il contenuto di ***x***, in due fasi.

```
#include <stdio.h>
#include <string.h>
int main (void)
{
  char x[] = "abcdefghijklmnopqrstuvwxyz";
  char y[50] = "ciao";
  printf ("prima: %s\n", y);
  strncat (y, x, 10);
  printf ("durante: %s\n", y);
  strncat (y, x, 40);
  printf ("dopo: %s\n", y);
  return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
prima: ciao
durante: ciaoabcdefghij
dopo: ciaoabcdefghijabcdefghijklmnopqrstuvwxyz
```

69.14.3 Comparazione

«

Le funzioni di comparazione ***memcmp()***, ***strcmp()*** e ***strncmp()*** confrontano due sequenze di caratteri, determinando se la prima sia maggiore, minore o uguale rispetto alla seconda, scandendo i caratteri progressivamente e arrestando l'analisi appena si incontra una differenza. Pertanto, il carattere che differisce è quello che determina l'ordine tra le due sequenze.

```
int    memcmp  (const void *s1, const void *s2, size_t n);
int    strcmp  (const char *s1, const char *s2);
int    strcoll (const char *s1, const char *s2);
int    strncmp (const char *s1, const char *s2, size_t n);
size_t strxfrm (char *restrict dst,
                const char *restrict org, size_t n);
```

69.14.3.1 Funzione «memcmp()» (memory compare)

«

La funzione ***memcmp()*** confronta i primi ***n*** caratteri delle aree di memoria a cui puntano ***s1*** e ***s2***, restituendo: un valore pari a zero se le due sequenze si equivalgono; un valore maggiore di zero se la sequenza di ***s1*** è maggiore di ***s2***; un valore minore di zero se la sequenza di ***s1*** è minore di ***s2***.

```
#include <string.h>
int
memcmp (const void *s1, const void *s2, size_t n)
{
  unsigned char *a = (unsigned char *) s1;
  unsigned char *b = (unsigned char *) s2;
  size_t i;
  for (i = 0; n > 0 && i < n; i++)
    {
      if (a[i] > b[i])
```

```
        {
          return 1;
        }
      else if (a[i] < b[i])
        {
          return -1;
        }
    }
  return 0;
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, dove le variabili ***x*** e ***y*** sono interi (che si presume siano a 32 bit) rappresentati in memoria invertendo l'ordine dei byte (*little endian*), pertanto il confronto avviene in modo inverso all'apparenza dei simboli.

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  unsigned int  x = 0x123456FF;
  unsigned int  y = 0xEEEEEEEE;
  int  r;
  r = memcmp (&x, &y, sizeof (int));
  printf ("memcmp: %x %i %x\n", x, r, y);
  r = memcmp (&x, &x, sizeof (int));
  printf ("memcmp: %x %i %x\n", x, r, x);
  r = memcmp (&y, &x, sizeof (int));
  printf ("memcmp: %x %i %x\n", y, r, x);
  return 0;
}
```

Avviando questo programma nelle condizioni descritte, si deve ottenere un risultato come quello seguente:

```
memcmp: 123456ff 1 eeeeeeee
memcmp: 123456ff 0 123456ff
memcmp: eeeeeeee -1 123456ff
```

69.14.3.2 Funzione «strcmp()» (string compare)

«

La funzione ***strcmp()*** confronta due stringhe restituendo: un valore pari a zero se sono uguali; un valore maggiore di zero se la stringa ***s1*** è maggiore di ***s2***; un valore minore di zero se la stringa ***s1*** è minore di ***s2***.

```
#include <string.h>
int
strcmp (const char *s1, const char *s2)
{
  unsigned char *a = (unsigned char *) s1;
  unsigned char *b = (unsigned char *) s2;
  size_t i;
  for (i = 0; ; i++)
    {
      if (a[i] > b[i])
        {
          return 1;
        }
      else if (a[i] < b[i])
        {
          return -1;
        }
      else if (a[i] == 0 && b[i] == 0)
        {
          return 0;
        }
```

```
    }
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  char x[] = "ciao";
  char y[] = "ciao amore";
  int  r;
  r = strcmp (x, y);
  printf ("strcmp: %s %i %s\n", x, r, y);
  r = strcmp (x, x);
  printf ("strcmp: %s %i %s\n", x, r, x);
  r = strcmp (y, x);
  printf ("strcmp: %s %i %s\n", y, r, x);
  return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
strcmp: ciao -1 ciao amore
strcmp: ciao 0 ciao
strcmp: ciao amore 1 ciao
```

69.14.3.3 Funzione «strcoll()» (string collate compare)

«

La funzione ***strcoll()*** è analoga a ***strcmp()***, con la differenza che la comparazione avviene sulla base della configurazione locale (la categoria '**LC_COLLATE**'). Nel caso della configurazione locale '**C**' la funzione si comporta esattamente come ***strcmp()***.

69.14.3.4 Funzione «strncmp()»

«

La funzione ***strncmp()*** si comporta in modo analogo a ***strcmp()***, con la differenza che la comparazione si arresta al massimo dopo ***n*** caratteri.

```
#include <string.h>
int
strncmp (const char *s1, const char *s2, size_t n)
{
  unsigned char *a = (unsigned char *) s1;
  unsigned char *b = (unsigned char *) s2;
  size_t i;
  for (i = 0; i < n ; i++)
    {
      if (a[i] > b[i])
        {
          return 1;
        }
      else if (a[i] < b[i])
        {
          return -1;
        }
      else if (a[i] == 0 && b[i] == 0)
        {
          return 0;
        }
    }
```

```
  return 0;
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  char x[] = "CIao";
  char y[] = "CIAO";
  int  r;
  r = strncmp (x, y, 4);
  printf ("strcmp: %i %s %i %s\n", 4, x, r, y);
  r = strncmp (x, y, 2);
  printf ("strcmp: %i %s %i %s\n", 2, x, r, x);
  r = strncmp (y, x, 4);
  printf ("strcmp: %i %s %i %s\n", 4, y, r, x);
  return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
strncmp: 4 CIao 1 CIAO
strncmp: 2 CIao 0 CIao
strncmp: 4 CIAO -1 CIao
```

69.14.3.5 Funzione «strxfrm()» (string transform)

«

La funzione ***strxfrm()*** trasforma la stringa ***org*** sovrascrivendo la stringa ***dst*** in modo relativo alla configurazione locale. In pratica, la stringa trasformata che si ottiene può essere comparata con un'altra stringa trasformata nello stesso modo attraverso la funzione ***strcmp()*** ottenendo lo stesso esito che si avrebbe confrontando le stringhe originali con la funzione ***strcoll()***.

Nella stringa di destinazione vengono messi non più di ***n*** caratteri, incluso il carattere nullo di terminazione. Se ***n*** è pari a zero, ***dst*** può essere un puntatore nullo. Le due stringhe non devono sovrapporsi.

La funzione ***strxfrm()*** restituisce la quantità di caratteri necessari a contenere la stringa ***org*** trasformata, senza però contare il carattere nullo di terminazione. Se ***n*** è zero e ***dst*** corrisponde al puntatore nullo, restituisce il valore che sarebbe necessario per trasformare la stringa ***org*** in tutta la sua lunghezza.

L'esempio seguente di tale funzione è valido solo per la configurazione locale '`C`':

```
#include <string.h>
size_t
strxfrm (char *restrict dst, const char *restrict org,
         size_t n)
{
  size_t i;
  if (n == 0 && dst == NULL)
    {
      return strlen (org);
    }
  else
    {
```

```
      for (i = 0; i < n ; i++)
        {
          dst[i] = org[i];
          if (org[i] == 0)
            {
              break;
            }
        }
      return i;
    }
}
```

69.14.4 Ricerca

Seguono i prototipi delle funzioni utili per la ricerca all'interno di sequenze di byte, secondo lo standard C:

```
void  *memchr   (const void *s, int c, size_t n);
char  *strchr   (const char *s, int c);
char  *strrchr  (const char *s, int c);
size_t strspn   (const char *s, const char *accept);
size_t strcspn  (const char *s, const char *reject);
char  *strpbrk  (const char *s, const char *accept);
char  *strstr   (const char *string, const char *substring);
char  *strtok   (char *restrict string,
                 const char *restrict delim);
```

Lo standard POSIX aggiunge anche il prototipo seguente:

```
char  *strtok_r (char *restrict string,
                 const char *restrict delim,
                 char **saveptr);
```

69.14.4.1 Funzione «memchr()» (memory character)

«

La funzione ***memchr()*** cerca un carattere a partire da una certa posizione in memoria, scandendo al massimo una quantità determinata di caratteri, restituendo il puntatore al carattere trovato. Se nell'ambito specificato non trova il carattere, restituisce il puntatore nullo.

```
#include <string.h>
void *
memchr (const void *s, int c, size_t n)
{
  unsigned char *a = (unsigned char *) s;
  unsigned char x  = (unsigned char) c;
  size_t i;
  for (i = 0; n > 0 && i < n; i++)
    {
      if (a[i] == x)
        {
          return (void *) (s + i);
        }
    }
  return NULL;
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente, in cui si scandisce il contenuto di una variabile di tipo '**int**', intendendo che questa debba occupare uno spazio di 32 bit:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
```

```
  int x = 0x13579BDF;
  void *p;
  p = memchr (&x, 0xDF, 4);
  printf ("contenuto della variabile: %x\n", x);
  printf ("indirizzo iniziale della variabile: 0x%x\n",
          (unsigned int) &x);
  printf ("indirizzo di 0x%x all'interno della "
          "variabile: 0x%x\n",
          0xDF,
          (unsigned int) p);
  return 0;
}
```

Avviando questo programma in un'architettura che inverte l'ordine dei byte (*little endian*) si deve ottenere un risultato simile a quello seguente:

```
contenuto della variabile: 13579bdf
indirizzo iniziale della variabile: 0xbff7f3bc
indirizzo di 0xdf all'interno della variabile: 0xbff7f3bc
```

69.14.4.2 Funzione «strchr()» (string character)

La funzione ***strchr()*** cerca un carattere all'interno di una stringa, restituendo il puntatore al carattere trovato, oppure il puntatore nullo se la ricerca fallisce. Nella scansione viene preso in considerazione anche il carattere nullo di terminazione della stringa.

```
#include <string.h>
char *
strchr (const char *s, int c)
{
  unsigned char *a = (unsigned char *) s;
  unsigned char x  = (unsigned char) c;
```

```
  size_t i;
  for (i = 0; ; i++)
    {
      if (a[i] == x)
        {
          return (char *) (s + i);
        }
      else if (a[i] == 0)
        {
          return NULL;
        }
    }
}
```

Per verificare sommariamente il comportamento della funzione si può realizzare un programma molto semplice come quello seguente:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  char *x = "ciao amore mio";
  char *p;

  p = strchr (x, 'a');
  printf ("La stringa \"%s\", collocata a partire ", x);
  printf ("dall'indirizzo %u, contiene il carattere '%c' ",
          (unsigned int) x,  'a');
  printf ("all'indirizzo %u.\n", (unsigned int) p);

  p = strchr (x, 0);
  printf ("La stringa \"%s\", collocata a partire ", x);
  printf ("dall'indirizzo %u, contiene il carattere 0x%x ",
          (unsigned int) x, 0);
```

```
  printf ("all'indirizzo %u.\n", (unsigned int) p);

  return 0;
}
```

Avviando questo programma si deve ottenere un risultato simile a quello seguente:

```
La stringa "ciao amore mio", collocata a partire ↩
↪dall'indirizzo 134516936, contiene il carattere ↩
↪'a' all'indirizzo 134516938.
La stringa "ciao amore mio", collocata a partire ↩
↪dall'indirizzo 134516936, contiene il carattere ↩
↪0x0 all'indirizzo 134516950.
```

69.14.4.3 Funzione «strrchr()» (string character)

La funzione ***strrchr()*** cerca un carattere all'interno di una stringa, restituendo il puntatore all'ultimo carattere corrispondente trovato, oppure il puntatore nullo se la ricerca fallisce. Nella scansione viene preso in considerazione anche il carattere nullo di terminazione della stringa.

```
#include <string.h>
char *
strrchr (const char *string, int c)
{
  int i;
  //
  for (i = strlen (string); i >= 0; i--)
    {
      if (string[i] == (char) c)
        {
          break;
        }
```

```
    }
  //
  if (i < 0)
    {
      return NULL;
    }
  else
    {
      return (string + i);
    }
}
```

Per verificare sommariamente il comportamento della funzione si può modificare leggermente l'esempio già apparso a proposito della funzione ***strchr()***:

```
…
  p = strrchr (x, 'a');
…
  p = strrchr (x, 0);
…
```

Avviando questo programma si deve ottenere un risultato simile a quello seguente:

```
La stringa "ciao amore mio", collocata a partire ↩
↪dall'indirizzo 134514088, contiene il carattere ↩
↪'a' all'indirizzo 134514093.
La stringa "ciao amore mio", collocata a partire ↩
↪dall'indirizzo 134514088, contiene il carattere ↩
↪0x0 all'indirizzo 134514102.
```

69.14.4.4 Funzione «strspn()» (string span)

La funzione ***strspn()*** calcola la lunghezza massima iniziale della stringa ***s***, composta esclusivamente da caratteri contenuti nella stringa ***accept***, restituendo tale valore.

```
#include <string.h>
size_t
strspn (const char *s, const char *accept)
{
  size_t i;
  size_t j;
  int found;
  for (i = 0; s[i] != 0; i++)
    {
      for (j = 0, found = 0; accept[j] != 0; j++)
        {
          if (s[i] == accept[j])
            {
              found = 1;
              break;
            }
        }
      if (!found)
        {
          break;
        }
    }
  return i;
}
```

Per verificare sommariamente il comportamento della funzione si può utilizzare l'esempio seguente:

```
#include <stdio.h>
```

```
#include <string.h>
int
main (void)
{
  char *x = "ciao amore mio";
  char *y = "abcdefghi";
  size_t n;

  n = strspn (x, y);
  printf ("La parte iniziale di \"%s\" ", x);
  printf ("che contiene i caratteri \"%s\" ", y);
  printf ("si compone di %i caratteri.\n", n);

  n = strspn (x, x);
  printf ("La parte iniziale di \"%s\" ", x);
  printf ("che contiene i caratteri \"%s\" ", x);
  printf ("si compone di %i caratteri.\n", n);

  return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
La parte iniziale di "ciao amore mio" che contiene i ↩
↪caratteri "abcdefghi" si compone di 3 caratteri.
La parte iniziale di "ciao amore mio" che contiene i ↩
↪caratteri "ciao amore mio" si compone di 14 caratteri.
```

69.14.4.5 Funzione «strcspn()»

«

La funzione ***strcspn()*** si comporta in modo analogo a ***strspn()***, con la differenza che l'insieme di caratteri contenuto nella stringa '**`reject`**' non deve costituire l'insieme iniziale della stringa ***s*** che

si va a contare. Pertanto, il valore restituito è la quantità di caratteri iniziali della stringa ***s*** che non si trovano anche nell'insieme ***reject***.

```
#include <string.h>
size_t
strcspn (const char *s, const char *reject)
{
  size_t i;
  size_t j;
  int found;
  for (i = 0; s[i] != 0; i++)
    {
      for (j = 0, found = 0; reject[j] != 0 || found; j++)
        {
          if (s[i] == reject[j])
            {
              found = 1;
              break;
            }
        }
      if (found)
        {
          break;
        }
    }
  return i;
}
```

Per verificare sommariamente il comportamento della funzione si può utilizzare l'esempio seguente:

```
#include <stdio.h>
#include <string.h>
int
main (void)
```

```
{
  char *x = "ciao amore mio";
  char *y = "mnopqrstuvwxyz";
  size_t n;

  n = strcspn (x, y);
  printf ("La parte iniziale di \"%s\" ", x);
  printf ("che non contiene i caratteri \"%s\" ", y);
  printf ("si compone di %i caratteri.\n", n);

  n = strcspn (x, x);
  printf ("La parte iniziale di \"%s\" ", x);
  printf ("che non contiene i caratteri \"%s\" ", x);
  printf ("si compone di %i caratteri.\n", n);

  return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
La parte iniziale di "ciao amore mio" che non contiene i ↩
↪caratteri "mnopqrstuvwxyz" si compone di 3 caratteri.
La parte iniziale di "ciao amore mio" che non contiene i ↩
↪caratteri "ciao amore mio" si compone di 0 caratteri.
```

69.14.4.6 Funzione «strpbrk()» (string point break)

«

La funzione ***strpbrk()*** scandisce la stringa ***s*** alla ricerca del primo carattere che risulti contenuto nella stringa ***accept***, restituendo il puntatore al carattere trovato, oppure, in mancanza di alcuna corrispondenza, il puntatore nullo.

```
#include <string.h>
char *
```

```
strpbrk (const char *s, const char *accept)
{
  size_t i;
  size_t j;
  for (i = 0; s[i] != 0; i++)
    {
      for (j = 0; accept[j] != 0; j++)
        {
          if (s[i] == accept[j])
            {
              return (char *) (s + i);
            }
        }
    }
  return NULL;
}
```

Per verificare sommariamente il comportamento della funzione si può utilizzare l'esempio seguente:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  char *x = "ciao amore mio";
  char *y = "mnopqrstuvwxyz";
  char *p;

  p = strpbrk (x, y);
  printf ("La stringa \"%s\" che inizia all'indirizzo %u, ",
          x, (unsigned int) x);
  printf ("trova la prima corrispondenza con la "
          "stringa \"%s\" ", y);
  printf ("all'indirizzo %u.\n", (unsigned int) p);
```

```
  return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
La stringa "ciao amore mio" che inizia all'indirizzo ↩
↪134516840, trova la prima corrispondenza con la ↩
↪stringa "mnopqrstuvwxyz" all'indirizzo 134516843.
```

69.14.4.7 Funzione «strstr()»

«

La funzione ***strstr()*** cerca la stringa ***substring*** nella stringa ***string*** restituendo il puntatore alla prima corrispondenza trovata (nella stringa ***string***). Se la corrispondenza non c'è, la funzione restituisce il puntatore nullo.

```
#include <string.h>
char *
strstr (const char *string, const char *substring)
{
  size_t i;
  size_t j;
  size_t k;
  int found;
  if (substring[0] == 0)
    {
      return (char *) string;
    }
  for (i = 0, j = 0, found = 0; string[i] != 0; i++)
    {
      if (string[i] == substring[0])
        {
```

```
          for (k = i, j = 0;
               string[k] == substring[j] && string[k] != 0
                && substring[j] != 0;
               j++, k++)
            {
              ;
            }
          if (substring[j] == 0)
            {
              found = 1;
            }
        }
      if (found)
        {
          return (char *) (string + i);
        }
    }
  return NULL;
}
```

Per verificare sommariamente il comportamento della funzione si può utilizzare l'esempio seguente:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  char *x = "ciao amore mio";
  char *y = "amore";
  char *p;

  p = strstr (x, y);
  printf ("La stringa \"%s\" che inizia all'indirizzo %u, ",
          x, (unsigned int) x);
```

```
  printf ("contiene la stringa \"%s\" ", y);
  printf ("all'indirizzo %u.\n", (unsigned int) p);

  p = strstr (x, "");
  printf ("La stringa \"%s\" che inizia all'indirizzo %u, ",
          x, (unsigned int) x);
  printf ("contiene la stringa \"%s\" ", "");
  printf ("all'indirizzo %u.\n", (unsigned int) p);

  p = strstr (x, "baba");
  printf ("La stringa \"%s\" che inizia all'indirizzo %u, ",
          x, (unsigned int) x);
  printf ("contiene la stringa \"%s\" ", "baba");
  printf ("all'indirizzo %u.\n", (unsigned int) p);

  return 0;
}
```

Avviando questo programma si deve ottenere un risultato come quello seguente:

```
La stringa "ciao amore mio" che inizia all'indirizzo ↩
↪134517000, contiene la stringa "amore" all'indirizzo ↩
↪134517005.
La stringa "ciao amore mio" che inizia all'indirizzo ↩
↪134517000, contiene la stringa "" all'indirizzo 134517000.
La stringa "ciao amore mio" che inizia all'indirizzo ↩
↪134517000, contiene la stringa "baba" all'indirizzo 0.
```

69.14.4.8 Funzione «strtok()» (string token)

«

La funzione ***strtok()*** serve a suddividere una stringa in unità, definite *token*, specificando un elenco di caratteri da intendere come delimitatori, in una seconda stringa. La funzione va usata in fasi successive, fornendo solo inizialmente la stringa da suddividere che continua poi

a essere utilizzata se al suo posto viene fornito il puntatore nullo. La funzione restituisce, di volta in volta, il puntatore alla sottostringa contenente l'unità individuata, oppure il puntatore nullo, se non può trovarla.

La funzione deve tenere memoria di un puntatore in un'area di memoria persistente (quello che nei commenti viene definito «puntatore statico») e deve isolare le unità modificando la stringa originale, inserendo il carattere nullo di terminazione alla fine delle unità individuate.

Quando la funzione viene chiamata indicando al posto della stringa da scandire il puntatore nullo, l'insieme dei delimitatori può essere diverso da quello usato nelle fasi precedenti.

```
#include <string.h>
char *
strtok (char *restrict string, const char *restrict delim)
{
  static char *next = NULL;
  size_t i = 0;
  size_t j;
  int found_token;
  int found_delim;
  //
  // Se la stringa fornita come argomento è un puntatore
  // nullo, occorre avvalersi del puntatore statico. Se
  // però questo è nullo a sua volta, la scansione non può
  // avvenire.
  //
  if (string == NULL)
    {
      if (next == NULL)
        {
          return NULL;
```

```
        }
      else
        {
          string = next;
        }
    }
  //
  // Se la stringa fornita come argomento è vuota, la
  // scansione non può avvenire.
  //
  if (string[0] == 0)
    {
      next = NULL;
      return NULL;
    }
  else
    {
      if (delim[0] == 0)
        {
          return string;
        }
    }
  //
  // Trova la prossima unità (token).
  //
  for (i = 0, found_token = 0, j = 0;
       string[i] != 0 && (!found_token);
       i++)
    {
      //
      // Cerca tra i delimitatori.
      //
      for (j = 0, found_delim = 0; delim[j] != 0; j++)
        {
```

```
            if (string[i] == delim[j])
              {
                found_delim = 1;
              }
          }
        //
        // Se il carattere attuale della stringa non è
        // un delimitatore, si tratta dell'inizio di una
        // nuova unità (token).
        //
        if (!found_delim)
          {
            found_token = 1;
            break;
          }
      }
    //
    // Se è stata trovata una unità (token) viene aggiustato
    // il puntatore che rappresenta la stringa. Se invece
    // non è stata trovata l'unità, vuol dire che non ce ne
    // possono essere altre.
    //
    if (found_token)
      {
        string += i;
      }
    else
      {
        next = NULL;
        return NULL;
      }
    //
    // Cerca la fine dell'unità trovata.
    //
```

```
  for (i = 0, found_delim = 0; string[i] != 0; i++)
    {
      for (j = 0; delim[j] != 0; j++)
        {
          if (string[i] == delim[j])
            {
              found_delim = 1;
              break;
            }
        }
      if (found_delim)
        {
          break;
        }
    }
  //
  // Se è stato trovato un delimitatore, allora il carattere
  // corrispondente nella stringa deve essere azzerato.
  // Se invece la stringa originale è terminata per conto
  // proprio, allora non è possibile continuare la ricerca
  // in una fase successiva, perché non ci possono essere
  // altre unità.
  //
  if (found_delim)
    {
      string[i] = 0;
      next = &string[i+1];
    }
  else
    {
      next = NULL;
    }
  //
  // A questo punto, la stringa attuale rappresenta
```

```
  // l'unità trovata.
  //
  return string;
}
```

Per comprendere lo scopo della funzione viene utilizzato lo stesso esempio che appare nel documento *ISO/IEC 9899:TC2*, al paragrafo 7.21.5.7, con qualche piccola modifica per poterlo rendere un programma autonomo:

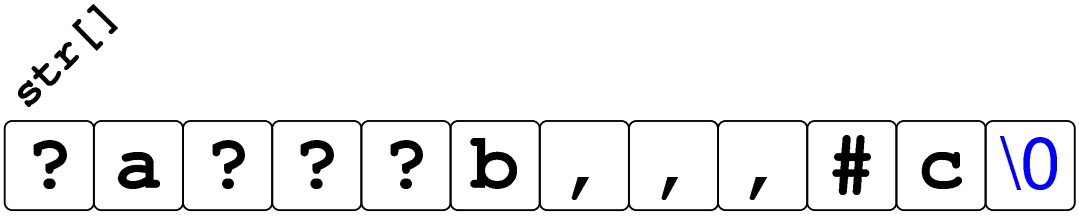
```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  char str[] = "?a???b,,,#c";
  char *t;

  t = strtok (str, "?");           // t punta all'unità "a"
  printf ("strtok: \"%s\"\n", t);
  t = strtok (NULL, ",");          // t punta all'unità "??b"
  printf ("strtok: \"%s\"\n", t);
  t = strtok (NULL, "#,");         // t punta all'unità "c"
  printf ("strtok: \"%s\"\n", t);
  t = strtok (NULL, "?");          // t è un puntatore nullo
  printf ("strtok: \"%s\"\n", t);

  return 0;
}
```

Avviando il programma si ottiene quanto già descritto dai commenti inseriti nel codice:

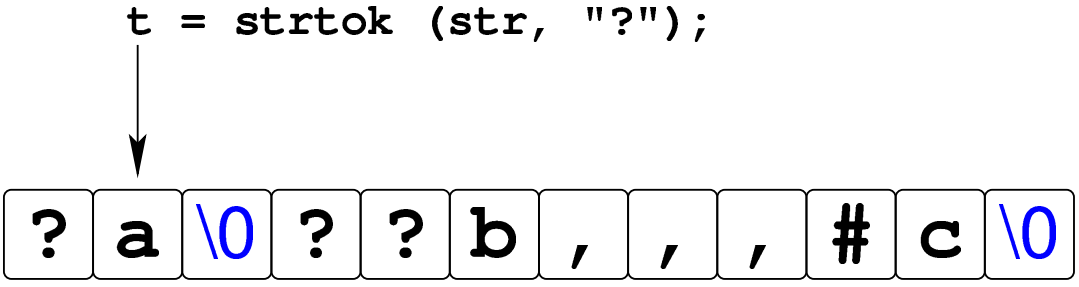
```
strtok: "a"
```

```
strtok: "??b"
strtok: "c"
strtok: "(null)"
```

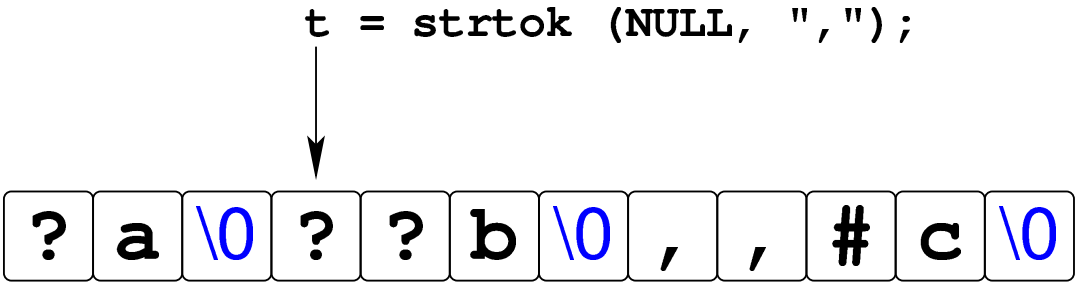
Ciò che avviene nell'esempio può essere schematizzato dalle figure successive. Inizialmente la stringa '**str**' ha in memoria l'aspetto seguente:



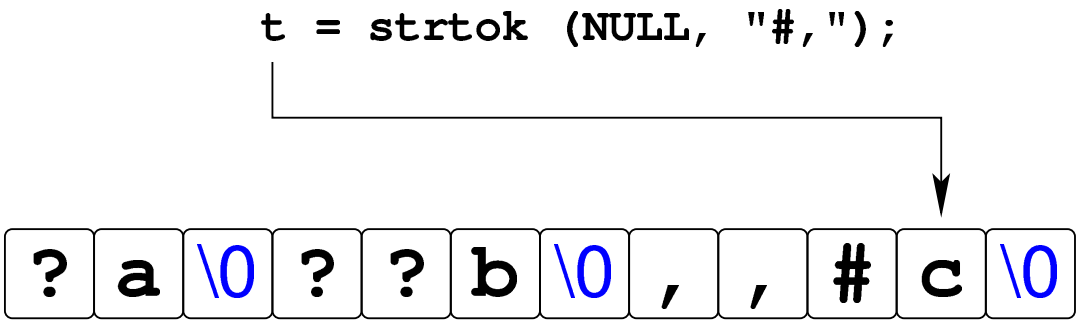
Dopo la prima chiamata della funzione ***strtok()*** la stringa risulta alterata e il puntatore ottenuto raggiunge la lettera '**a**':



Dopo la seconda chiamata della funzione, in cui si usa il puntatore nullo per richiedere una scansione ulteriore della stringa originale, si ottiene un nuovo puntatore che, questa volta, inizia a partire dal quarto carattere, rispetto alla stringa originale, dal momento che il terzo è già stato sovrascritto da un carattere nullo:



La penultima chiamata della funzione ***strtok()*** raggiunge la lettera '**c**' che è anche alla fine della stringa originale:



L'ultimo tentativo di chiamata della funzione non può dare alcun esito, perché la stringa originale si è già conclusa.

Va tenuto in considerazione che la funzione ***strtok()***, dovendo mantenere in memoria la posizione trovata dell'ultima scansione eseguita, da una chiamata a quella successiva, non è «rientrante», pertanto non si presta per i programmi che si suddividono in più thread.

69.14.4.9 Funzione «strtok_r()»

La funzione ***strtok_r()***, richiesta dallo standard POSIX, salva il puntatore interno alla stringa che viene scandita, esternamente, in modo da poter essere usata in un contesto in cui più thread operano simultaneamente. In pratica si aggiunge un terzo parametro, costituito da un puntatore a puntatore a carattere.

Il puntatore a carattere ('**char ***'), il cui puntatore viene fornito come terzo argomento della funzione, deve essere dichiarato prima della chiamata della funzione. La prima volta che viene chiamata la funzione ***strtok_r()*** non conta quale valore abbia effettivamente tale

variabile di tipo '**char ***', perché è la funzione stessa che lo inizializza, ma nelle chiamate successive (quando al posto della stringa si dà alla funzione il valore '**NULL**'). Quando termina la scansione della stringa con le chiamate di ***strtok_r()***, la variabile '**char ***' di cui si passa il puntatore, può essere utilizzata per altri scopi, o per altre scansioni.

La soluzione seguente, per la realizzazione della funzione ***strtok_r()***, può essere confrontato con quella relativa alla funzione ***strtok()***, per comprendere il senso e l'utilizzo dell'ultimo parametro.

```
char *
strtok_r (char *restrict string, const char *restrict delim,
          char **saveptr)
{
  size_t i = 0;
  size_t j;
  int found_token;
  int found_delim;
  //
  // Se la stringa fornita come argomento è un puntatore
  // nullo, occorre avvalersi del puntatore "*saveptr".
  // Se però questo è nullo a sua volta, la scansione non
  // può avvenire.
  //
  if (string == NULL)
    {
      if (*saveptr == NULL)
        {
          return NULL;
        }
      else
        {
          string = *saveptr;
        }
```

```
    }
  //
  // Se la stringa fornita come argomento è vuota, la
  // scansione non può avvenire.
  //
  if (string[0] == 0)
    {
      *saveptr = NULL;
      return NULL;
    }
  else
    {
      if (delim[0] == 0)
        {
          return string;
        }
    }
  //
  // Trova la prossima unità (token).
  //
  for (i = 0, found_token = 0, j = 0;
       string[i] != 0 && (!found_token);
       i++)
    {
      //
      // Cerca tra i delimitatori.
      //
      for (j = 0, found_delim = 0; delim[j] != 0; j++)
        {
          if (string[i] == delim[j])
            {
              found_delim = 1;
            }
        }
```

```
      //
      // Se il carattere attuale della stringa non è
      // un delimitatore, si tratta dell'inizio di una
      // nuova unità (token).
      //
      if (!found_delim)
        {
          found_token = 1;
          break;
        }
    }
  //
  // Se è stata trovata una unità (token) viene aggiustato
  // il puntatore che rappresenta la stringa. Se invece
  // non è stata trovata l'unità, vuol dire che non ce ne
  // possono essere altre.
  //
  if (found_token)
    {
      string += i;
    }
  else
    {
      *saveptr = NULL;
      return NULL;
    }
  //
  // Cerca la fine dell'unità trovata.
  //
  for (i = 0, found_delim = 0; string[i] != 0; i++)
    {
      for (j = 0; delim[j] != 0; j++)
        {
          if (string[i] == delim[j])
```

```
            {
              found_delim = 1;
              break;
            }
        }
      if (found_delim)
        {
          break;
        }
    }
  //
  // Se è stato trovato un delimitatore, allora il carattere
  // corrispondente nella stringa deve essere azzerato.
  // Se invece la stringa originale è terminata per conto
  // proprio, allora non è possibile continuare la ricerca
  // in una fase successiva, perché non ci possono essere
  // altre unità.
  //
  if (found_delim)
    {
      string[i] = 0;
      *saveptr = &string[i+1];
    }
  else
    {
      *saveptr = NULL;
    }
  //
  // A questo punto, la stringa attuale rappresenta
  // l'unità trovata.
  //
  return string;
}
```

Per dimostrare il lavoro della funzione, viene utilizzato lo stesso esempio già usato a proposito di ***strtok()***, con poche piccole modifiche:

```
int
main (void)
{
  char str[] = "?a???b,,,#c";
  char *t;
  char *save;

  t = strtok_r (str, "?", &save);   // t punta all'unità "a"
  printf ("strtok: \"%s\"\n", t);
  t = strtok_r (NULL, ",", &save);  // t punta all'unità "??b"
  printf ("strtok: \"%s\"\n", t);
  t = strtok_r (NULL, "#,", &save); // t punta all'unità "c"
  printf ("strtok: \"%s\"\n", t);
  t = strtok_r (NULL, "?", &save);  // t è un puntatore nullo
  printf ("strtok: \"%s\"\n", t);

  return 0;
}
```

Avviando il programma si ottiene esattamente la stessa cosa dell'esempio già visto:

```
strtok: "a"
strtok: "??b"
strtok: "c"
strtok: "(null)"
```

69.14.5 Funzioni varie

Seguono i prototipi delle funzioni descritte nelle sezioni successive. Questi appartengono allo standard C:

```
void  *memset     (void *s, int c, size_t n);
char  *strerror   (int errnum);
size_t strlen     (const char *s);
```

Il prototipo successivo viene aggiunto dallo standard POSIX:

POSIX

```
int    strerror_r (int errnum, char *s, size_t n);
```

69.14.5.1 Funzione «memset()» (memory set)

La funzione ***memset()*** consente di inizializzare una certa area di memoria con la ripetizione di un certo carattere. Per la precisione, viene usato il valore del parametro ***c***, tradotto in un carattere senza segno, copiandolo per ***n*** volte a partire dall'indirizzo a cui punta ***s***. La funzione restituisce ***s***.

```
#include <string.h>
void *
memset (void *s, int c, size_t n)
{
  unsigned char *a = (unsigned char *) s;
  unsigned char x  = (unsigned char) c;
  size_t i;
  for (i = 0; n > 0 && i < n; i++)
    {
      a[i] = x;
    }
  return s;
}
```

Per verificare sommariamente il comportamento della funzione si può utilizzare l'esempio seguente:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  int x = 0x12345678;
  printf ("prima: 0x%x\n", x);
  memset (&x, 0xFF, 2);
  printf ("dopo: 0x%x\n", x);

  char X[] = "ciao amore mio";
  printf ("prima: \"%s\"\n", X);
  memset (X, 'Q', 5);
  printf ("dopo: \"%s\"\n", X);

  return 0;
}
```

Avviando questo programma in un elaboratore con architettura a 32 bit e inversione dei byte (*little endian*) si deve ottenere il risultato seguente:

```
prima: 0x12345678
dopo: 0x1234ffff
prima: "ciao amore mio"
dopo: "QQQQQamore mio"
```

69.14.5.2 Funzione «strerror()» (string error)

La funzione ***strerror()*** serve a tradurre il numero fornito come argomento in un puntatore da cui inizia una stringa contenente una spiegazione. In altri termini, serve a trasformare un numero in una descrizione di un tipo di errore. Qui viene mostrata una soluzione priva di utilità, anche se risponde alle richieste delle specifiche:

```
#include <string.h>
char *
strerror (int errnum)
{
  static char answare[] = "Unknown error";
  return answare;
}
```

L'esempio successivo può servire a dimostrare il senso di questa funzione:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  printf ("%s\n", strerror (0));
  printf ("%s\n", strerror (1));
  printf ("%s\n", strerror (2));
  printf ("%s\n", strerror (3));
  printf ("%s\n", strerror (4));
  return 0;
}
```

Utilizzando questo programma compilato con le librerie di un sistema GNU si potrebbero vedere i messaggi seguenti:

```
Success
Operation not permitted
No such file or directory
No such process
Interrupted system call
```

La stringa a cui punta la funzione può essere condivisa da altre chiamate successive della stessa, pertanto, in un programma con thread multipli, è possibile che avvenga la sovrascrittura, a meno di disporre di un elenco separato di tutti i tipi di messaggio di errore.

69.14.5.3 Funzione «strerror_r()» (string error)

«

La funzione ***strerror_r()*** viene aggiunta dallo standard POSIX e consente di tradurre un errore numerico in stringa, fornendo alla funzione il puntatore iniziale della stringa da produrre e la lunghezza massima che questa può raggiungere, garantendo l'indipendenza tra thread multipli.

POSIX

Per quanto riguarda l'utilizzo, la differenza fondamentale rispetto a ***strerror()*** sta nel fatto che restituisce un valore intero, pari a zero, se l'operazione ha avuto successo, oppure −1 in caso di problemi, aggiornando di conseguenza anche la variabile ***errno***. L'esempio successivo può servire a dimostrare il senso di questa funzione:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  char msg[100];

  if (!strerror_r (1, msg, 100))
```

```
    {
      printf ("%s\n", msg);
    }
  if (!strerror_r (2, msg, 100))
    {
      printf ("%s\n", msg);
    }
  if (!strerror_r (3, msg, 100))
    {
      printf ("%s\n", msg);
    }
  if (!strerror_r (4, msg, 100))
    {
      printf ("%s\n", msg);
    }
  return 0;
}
```

Utilizzando questo programma compilato con le librerie di un sistema GNU si potrebbero vedere i messaggi seguenti:

```
Success
Operation not permitted
No such file or directory
No such process
Interrupted system call
```

69.14.5.4 Funzione «strlen()» (string length)

La funzione ***strlen()*** calcola la lunghezza di una stringa, escludendo dal conteggio il carattere nullo di terminazione:

```
#include <string.h>
size_t
```

```
strlen (const char *s)
{
  size_t i;
  for (i = 0; s[i] != 0 ; i++)
    {
      ; // Esegue solo il conteggio.
    }
  return i;
}
```

Per verificare sommariamente il comportamento della funzione si può utilizzare l'esempio seguente:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
  size_t lunghezza;
  char stringa[] = "ciao amore";
  lunghezza = strlen (stringa);
  printf ("la frase \"%s\" si compone di %i caratteri\n",
          stringa,
          lunghezza);
  return 0;
}
```

Avviando il programma si deve vedere il risultato seguente:

```
la frase "ciao amore" si compone di 10 caratteri
```

69.15 File «signal.h»

Il file ‘`signal.h`’ della libreria standard definisce principalmente delle funzioni per la gestione dei segnali che riguardano il programma. Assieme alle funzioni definisce anche delle macro-variabili per classificare i segnali e per fare riferimento a delle funzioni predefinite, destinate astrattamente al trattamento dei segnali (si veda eventualmente la realizzazione del file ‘`signal.h`’ e di alcune delle sue funzioni nei sorgenti di os32, sezione 95.17).

Dal punto di vista del programmatore, l'uso delle funzioni di questo file di intestazione può essere abbastanza semplice, ma la comprensione di come siano organizzate nel file ‘`signal.h`’ diventa invece difficile.

Nello standard POSIX, la questione dei segnali è particolarmente complessa. Nel capitolo viene considerato solo il fatto che i segnali standard sono in numero maggiore, tralasciando sostanzialmente il resto.

Qui vengono proposti due modi alternativi di scrivere il file ‘`signal.h`’ che dovrebbero essere disponibili presso *allegati/c/include/signal.h* e *allegati/c/include/signal-bis.h* .

69.15.1 Dichiarazione contorta

Per la gestione dei segnali ci sono due funzioni che vengono dichiarate nel file ‘`signal.h`’: ***signal()*** e ***raise()***. La funzione ***raise()*** serve ad azionare un segnale, come dire che serve ad attivare manualmente un allarme interno al programma, specificato da un numero particolare che ne definisce il tipo. Il programma contiene sempre una procedura predefinita che stabilisce ciò che deve essere fatto in

presenza di un certo allarme, ma il programmatore può ridefinire la procedura attraverso l'uso della funzione ***signal()***, con la quale si associa l'avvio di una funzione particolare in presenza di un certo segnale. Il modello sintattico seguente rappresenta, in modo estremamente semplificato, l'uso della funzione ***signal()***:

```
signal (n_segnale, funzione_da_associare)
```

Logicamente la funzione che si associa a un certo numero di segnale viene indicata negli argomenti della chiamata come puntatore a funzione. La funzione che viene passata come argomento è un gestore di segnale e deve avere una certa forma:

```
void gestore (n_segnale)
```

In pratica, quando viene creata l'associazione tra segnale e funzione che deve gestirlo, la funzione in questione deve avere un parametro tale da poter rappresentare il numero del segnale che la riguarda e non restituisce alcun valore (pertanto è di tipo '**void**').

Avendo determinato questo, il modello della funzione ***signal()*** può essere precisato un po' di più:

```
signal (n_segnale, void (*gestore)(int))
```

Ciò significa che il secondo argomento della funzione ***signal()*** è un puntatore a una funzione ('*gestore* **()**') con un parametro di tipo '**int**', la quale non restituisce alcunché ('**void**').

Ma non è ancora stato specificato cosa deve restituire la funzione

signal(): un puntatore a una funzione che ha un parametro di tipo '**int**' e che a sua volta non restituisce alcunché. In pratica, ***signal()*** deve restituire il puntatore a una funzione che ha le stesse caratteristiche di quella del proprio secondo parametro. A questo punto, si arriva al prototipo completo, ma molto difficile da interpretare a prima vista:

```
void (*signal (n_segnale, void (*gestore)(int)))(int);
```

Per ovviare a questo problema di comprensibilità, anche se lo standard non lo prescrive, di norma, nel file '`signal.h`' si dichiara un tipo speciale, in qualità di puntatore a funzione con le caratteristiche del gestore di segnale:

```
...
typedef void (*sighandler_t) (int);
...
```

Così facendo, la funzione ***signal()*** può essere dichiarata in modo più gradevole:

```
sighandler_t signal (n_segnale, sighandler_t gestore);
```

69.15.2 Tipo speciale

A parte il caso di '**sighandler_t**' che non fa parte dello standard del linguaggio, il file '`include.h`' definisce il tipo '**sig_atomic_t**', il cui uso non viene precisato dai documenti ufficiali. Si chiarisce solo che deve trattarsi di un valore intero, possibilmente di tipo volatile, a cui si possa accedere attraverso una sola

istruzione elementare del linguaggio macchina (in modo tale che la lettura o la modifica del suo contenuto non possa essere sospesa a metà da un'interruzione di qualunque genere).

```
typedef int sig_atomic_t;
```

Nell'esempio, il tipo '**sig_atomic_t**' viene dichiarato come equivalente al tipo '**int**', supponendo che l'accesso alla memoria per un tipo intero normale corrisponda a un'operazione «atomica» nel linguaggio macchina. A ogni modo, il tipo a cui corrisponde '**sig_atomic_t**' può dipendere da altri fattori, mentre l'unico vincolo nel rango è quello di poter contenere i valori rappresentati dalle macro-variabili ***SIG***..., le quali individuano mnemonicamente i segnali.

Il programmatore che deve memorizzare un segnale in una variabile, potrebbe usare per questo il tipo '**sig_atomic_t**'.

69.15.3 Denominazione dei segnali

«

Un gruppo di macro-variabili definisce l'elenco dei segnali gestibili. Lo standard del linguaggio ne prescrive solo una quantità minima, mentre il sistema operativo può richiederne degli altri. Teoricamente l'associazione del numero al nome simbolico del segnale è libera, ma in pratica la concordanza con altri standard prescrive il rispetto di un minimo di uniformità.

```
#define SIGINT           2
#define SIGILL           4
#define SIGABRT          6
#define SIGFPE           8
#define SIGSEGV         11
#define SIGTERM         15
```

Tabella 69.208. Denominazione dei segnali indispensabili al linguaggio C.

Denominazione	Significato mnemonico	Descrizione
`SIGABRT`	*abort*	Deriva da una terminazione anomala che può essere causata espressamente dall'uso della funzione ***abort()***.
`SIGFPE`	*floating point exception*	Viene provocato da un'operazione aritmetica errata, come la divisione per zero o uno straripamento del risultato.
`SIGILL`	*illegal*	Istruzione «illegale».
`SIGINT`	*interrupt*	Deriva dalla ricezione di una richiesta interattiva di attenzione, quale può essere quella di un'interruzione.
`SIGSEGV`	*segmentation violation*	Deriva da un accesso alla memoria non valido, per esempio oltre i limiti fissati.
`SIGTERM`	*termination*	Indica la ricezione di una richiesta di terminazione del funzionamento del programma.

69.15.4 Segnali secondo POSIX

Lo standard POSIX prescrive un insieme minimo di segnali più numeroso, attribuendo anche un'azione predefinita a carico del sistema operativo.

Tabella 69.209. Denominazione dei segnali indispensabili allo standard POSIX.

Denominazione	Azione predefinita	Descrizione
`SIGABRT`	terminazione anomala	Conclusione prematura del processo elaborativo.
`SIGALRM`	terminazione normale	Segnale di allarme da un orologio programmabile.
`SIGBUS`	terminazione anomala	Accesso errato alla memoria.
`SIGCHLD`	ignorato	Conclusione, sospensione o continuazione di un processo figlio.
`SIGCONT`	continuazione	Ripresa dell'esecuzione di un processo, se risulta sospeso.
`SIGFPE`	terminazione anomala	Operazione aritmetica errata.
`SIGHUP`	terminazione normale	Aggancio (interruzione del collegamento con il terminale).
`SIGILL`	terminazione anomala	Istruzione illegale.
`SIGINT`	terminazione normale	Segnale di interruzione dal terminale.
`SIGKILL`	terminazione normale	Eliminazione del processo elaborativo (senza la possibilità che il segnale sia catturato o ignorato).

Denominazione	Azione predefinita	Descrizione
SIGPIPE	terminazione normale	Scrittura verso un condotto (*pipe*) senza che alcun processo stia leggendo da lì.
SIGQUIT	terminazione anomala	Segnale di abbandono (*quit*) dal terminale.
SIGSEGV	terminazione anomala	Riferimento errato alla memoria.
SIGSTOP	sospensione	Sospensione dell'esecuzione (senza la possibilità che il segnale sia catturato o ignorato).
SIGTERM	terminazione normale	Richiesta di conclusione del funzionamento.
SIGSTOP	sospensione	Segnale di stop dal terminale.
SIGTTIN	sospensione	Processo sullo sfondo che attende di poter leggere (dalla tastiera).
SIGTTOU	sospensione	Processo sullo sfondo che attende di poter scrivere (sullo schermo del terminale).
SIGUSR1	terminazione normale	Segnale 1, il cui scopo è definibile dall'utente.
SIGUSR2	terminazione normale	Segnale 2, il cui scopo è definibile dall'utente.
SIGPOLL	terminazione normale	*Pollable event*.

Denominazione	Azione predefinita	Descrizione
`SIGPROF`	terminazione normale	*Profiling timer expired*.
`SIGSYS`	terminazione anomala	Chiamata di sistema errata.
`SIGTRAP`	terminazione anomala	Trappola scattata per il tracciamento del codice.
`SIGURG`	ignorato	Informazione urgente disponibile da un socket.
`SIGVTALRM`	terminazione normale	*Virtual timer expired*.
`SIGXCPU`	terminazione anomala	Limite del tempo di CPU superato.
`SIGXFSZ`	terminazione anomala	Limite della dimensione dei file superato.

L'azione predefinita è quella che deve essere svolta dal sistema operativo se il segnale non viene catturato o non viene ignorato dal programma che lo riceve (tenendo conto che per '**`SIGKILL`**' e '**`SIGSTOP`**' i programmi non possono intervenire). Una «terminazione normale» implica la conclusione normale del processo elaborativo, a parte il fatto che il valore restituito dal processo dipende dal segnale stesso; una «terminazione anomala» implica di solito qualcosa di più, di solito si tratta dello scarico della memoria del processo in un file (*core dump*), per consentire un'analisi di quanto accaduto; la

«sospensione» rappresenta un arresto temporaneo, in attesa di un segnale di «continuazione»; un segnale «ignorato» indica che lo stato del processo non viene cambiato, ma ciò non significa che il segnale in sé sia privo di conseguenze.

Il segnale '**SIGCHLD**' che formalmente viene indicato come privo di effetti, nella tradizione Unix ha un ruolo molto importante e relativamente complesso, per la gestione della dipendenza dei processi. In un sistema Unix i processi hanno una dipendenza gerarchica, trattata secondo un albero genealogico, dove ogni processo ha un genitore. Dato che la conclusione di un processo produce un valore che dovrebbe essere raccolto dal genitore che lo ha avviato (oppure che lo ha adottato, nel caso il genitore vero sia defunto nel frattempo), quando un processo muore (termina di funzionare per qualunque motivo), il genitore riceve un segnale '**SIGCHLD**': se il genitore è in attesa del valore di uscita del processo defunto, lo raccoglie e le tracce residue di tale processo possono essere distrutte definitivamente; altrimenti, se il segnale non viene catturato il processo defunto viene eliminato senza comunicare tale valore al genitore.

69.15.5 Gestori fittizi di segnali

Lo standard prescrive di definire tre macro-variabili che devono espandersi in un puntatore a quel tipo di funzione che deve essere in grado di gestire le azioni da compiere in relazione alla ricezione di un certo segnale. Tuttavia, questo puntatore non deve essere rivolto a una funzione vera, ma averne solo la forma. In pratica, si usano dei valori interi con un valore assoluto molto piccolo e si esegue un cast per trasformarli in puntatori a funzione, come già accennato.

Per ottenere questo risultato, si possono dichiarare le macro-

variabili in due modi equivalenti, con la differenza che il secondo è probabilmente più difficile da interpretare:

```
typedef void (*sighandler_t) (int); // Il tipo
                                    // «sighandler_t» è un
                                    // puntatore a funzione
                                    // per la gestione dei
                                    // segnali con parametro
                                    // «int» che
                                    // restituisce «void».
//
// Funzioni non dichiarabili
//
#define SIG_ERR ((sighandler_t) -1) // Trasforma un numero
#define SIG_DFL ((sighandler_t) 0)  // intero  in un tipo
#define SIG_IGN ((sighandler_t) 1)  // «sighandler_t»,
                                    // ovvero un puntatore
                                    // a funzione che però
                                    // non esiste realmente.
```

```
//
// Funzioni non dichiarabili
//
#define SIG_ERR ((void (*) (int)) -1) // Trasforma un numero
#define SIG_DFL ((void (*) (int)) 0)  // intero in un
#define SIG_IGN ((void (*) (int)) 1)  // puntatore a una
                                      // funzione che ha un
                                      // parametro «int» e
                                      // restituisce «void».
```

Lo standard sottolinea il fatto che il numero trasformato in puntatore non deve poter corrispondere all'indirizzo di alcuna funzione reale; pertanto i valori usati possono essere solo molto bassi o molto alti (in termini di valore assoluto), contando sul fatto che a tali indirizzi

non ci possano essere funzioni reali. In pratica, non deve succedere che venga dichiarata una funzione per la gestione di un segnale che finisca per avere proprio tali indirizzi, perché se così fosse, non verrebbe avviata, ma al suo posto verrebbe considerata l'azione che una di queste macro-variabili simboleggia.

Tabella 69.212. Macro-variabili per la gestione predefinita dei segnali.

Denominazione	Significato mnemonico	Descrizione
`SIG_DFL`	*default*	Indica simbolicamente che l'azione da compiere alla ricezione del segnale deve essere quella predefinita.
`SIG_IGN`	*ignore*	Indica simbolicamente che alla ricezione del segnale si procede come se nulla fosse accaduto.
`SIG_ERR`	*error*	Rappresenta un risultato errato nell'uso della funzione ***signal()***.

69.15.6 Funzioni

La funzione ***signal()*** viene usata per associare un «gestore di segnale», costituito dal puntatore a una funzione, a un certo segnale; tutto questo allo scopo di attivare automaticamente quella tale funzione al verificarsi di un certo evento che si manifesta tramite un certo segnale.

La funzione ***signal()*** restituisce un puntatore alla funzione che precedentemente si doveva occupare di quel segnale. Se invece l'operazione fallisce, ***signal()*** esprime questo errore restituendo il valo-

re ***SIG_ERR***, spiegando così il motivo per cui questo debba avere l'apparenza di un puntatore a funzione.

Per la stessa ragione per cui esiste ***SIG_ERR***, le macro-variabili ***SIG_DFL*** e ***SIG_IGN*** vanno usate come gestori di segnali, rispettivamente, per ottenere il comportamento predefinito o per far sì che i segnali siano ignorati semplicemente.

In linea di principio si può ritenere che nel proprio programma esista una serie iniziale di dichiarazioni implicite per cui si associano tutti i segnali gestibili a ***SIG_DFL***:

```
…
signal (segnale, SIG_DFL);
…
```

In base al fatto che sia stata dichiarato o meno il tipo '**sighandler_t**', la funzione potrebbe avere i prototipi seguenti:

```
sighandler_t signal (int sig, sighandler_t handler);
```

```
void (*signal (int sig, void (*handler) (int))) (int);
```

L'altra funzione da considerare è ***raise()***, con la quale si attiva volontariamente un segnale, dal quale poi dovrebbero o potrebbero sortire delle conseguenze, come stabilito in una fase precedente attraverso ***signal()***. La funzione ***raise()*** è molto semplice:

```
int raise (int sig);
```

La funzione richiede come argomento il numero del segnale da attivare e restituisce un valore pari a zero in caso di successo, altrimenti restituisce un valore diverso da zero. Naturalmente, a seconda del-

l'azione che viene intrapresa all'interno del programma, a seguito della ricezione del segnale, può darsi che dopo questa funzione non venga eseguito altro, pertanto non è detto che possa essere letto il valore che la funzione potrebbe restituire.

69.15.7 Esempio

Viene proposto un esempio che serve a dimostrare il meccanismo di provocazione e intercettazione dei segnali:

```
#include <stdio.h>
#include <signal.h>

void sig_generic_handler (int sig)
{
  printf ("Ho intercettato il segnale n. %i.\n", sig);
}

void sigfpe_handler (int sig)
{
  printf ("Attenzione: ho intercettato il segnale "
          "SIGFPE (%i)\n"
          "            e devo concludere il "
          "funzionamento!\n", sig);
  exit (sig);
}

void sigterm_handler (int sig)
{
  printf ("Attenzione: ho intercettato il segnale "
          "SIGTERM (%i),\n"
          "            però non intendo rispettarlo.\n",
          sig);
}
```

```
void sigint_handler (int sig)
{
  printf ("Attenzione: ho intercettato il segnale "
          "SIGINT (%i),\n"
          "            però non intendo rispettarlo.\n",
          sig);
}

int main (void)
{
  signal (SIGFPE,  sigfpe_handler);
  signal (SIGTERM, sigterm_handler);
  signal (SIGINT,  sigint_handler);
  signal (SIGILL,  sig_generic_handler);
  signal (SIGSEGV, sig_generic_handler);

  int c;
  int x;

  printf ("[0][Invio] divisione per zero\n");
  printf ("[c][Invio] provoca un segnale SIGINT\n");
  printf ("[t][Invio] provoca un segnale SIGTERM\n");
  printf ("[q][Invio] conclude il funzionamento\n");
  while (1)
    {
      c = getchar();
      if (c == '0')
        {
          printf ("Sto per eseguire una divisione per "
                  "zero:\n");
          x = x / 0;
        }
      else if (c == 'c')
        {
```

```
            raise (SIGINT);
          }
        else if (c == 't')
          {
            raise (SIGTERM);
          }
        else if (c == 'q')
          {
            return 0;
          }
      }
    return 0;
}
```

All'inizio del programma vengono definite delle funzioni per il trattamento delle situazioni che hanno provocato un certo segnale. Nella funzione ***main()***, prima di ogni altra cosa, si associano tali funzioni ai segnali principali, quindi si passa a un ciclo senza fine, nel quale possono essere provocati dei segnali premendo un certo tasto, come suggerito da un breve menù. Per esempio è possibile provocare la condizione che si verifica tentando di dividere un numero per zero:

```
[0][Invio] divisione per zero
[c][Invio] provoca un segnale SIGINT
[t][Invio] provoca un segnale SIGTERM
[q][Invio] conclude il funzionamento
```

0 [*Invio*]

```
Sto per eseguire una divisione per zero:
Attenzione: ho intercettato il segnale SIGFPE (8)
            e devo concludere il funzionamento!
```

La divisione per zero fa scattare il segnale '**SIGFPE**' che viene in-

tercettato dalla funzione ***sigfpe_handler()***, la quale però non può far molto e così conclude anche il funzionamento del programma.

Attraverso il menù è possibile provocare anche un segnale '**SIGINT**' e un segnale '**SIGTERM**', ma per questo è più interessante provare con i mezzi che dovrebbe offrire il sistema operativo:

```
[0][Invio] divisione per zero
[c][Invio] provoca un segnale SIGINT
[t][Invio] provoca un segnale SIGTERM
[q][Invio] conclude il funzionamento
```

[*Ctrl c*][*Invio*]

```
Attenzione: ho intercettato il segnale SIGINT (2),
            però non intendo rispettarlo.
```

Utilizzando un sistema operativo Unix o simile, da un altro terminale, o da un'altra console, è possibile inviare un segnale specifico al programma:

```
$ kill n_processo [Invio]

Attenzione: ho intercettato il segnale SIGTERM (15),
            però non intendo rispettarlo.

$ kill -s 4 n_processo [Invio]

Ho intercettato il segnale n. 4.

$ kill -s 11 n_processo [Invio]

Ho intercettato il segnale n. 11.
```

Secondo l'esempio, i segnali 4 e 11 sono, rispettivamente, '**SIGILL**' e '**SIGSEGV**'.

[q] [*Invio*]

69.16 File «time.h»

Il file ‘`time.h`’ della libreria standard definisce principalmente delle funzioni per il trattamento delle informazioni data-orario. Non è stabilito in che modo venga rappresentato il tempo internamente alle funzioni, anche se di norma si tratta di un valore intero che esprime una quantità di secondi o di frazioni di secondo (si veda eventualmente la realizzazione del file ‘`time.h`’ e di alcune delle sue funzioni nei sorgenti di os32, sezione 95.29).

69.16.1 Il tempo di CPU

La funzione ***clock()*** consente di ottenere il tempo di utilizzo del microprocessore (CPU), espresso virtualmente in cicli di CPU. In pratica, viene definita la macro-variabile ***CLOCKS_PER_SEC***, contenente il valore che esprime convenzionalmente la quantità di cicli di CPU per secondo; quindi, il valore restituito dalla funzione ***clock()*** si traduce in secondi dividendolo per ***CLOCKS_PER_SEC***. Il valore restituito dalla funzione ***clock()*** e l’espressione in cui si traduce la macro-variabile ***CLOCKS_PER_SEC*** sono di tipo ‘**`clock_t`**’:

```
typedef long int clock_t;    // Unità di tempo convenzionale
                             // che rappresenta un ciclo
                             // virtuale di CPU.

#define CLOCKS_PER_SEC  1000000L  // Valore convenzionale di
                                  // 1 s, in termini di
                                  // cicli virtuali di CPU.

clock_t clock (void);             // Tempo di utilizzo della
                                  // CPU.
```

La funzione ***clock()*** restituisce il tempo di CPU espresso in unità '**`clock_t`**', utilizzato dal processo elaborativo a partire dall'avvio del programma. Se la funzione non è in grado di dare questa indicazione, allora restituisce il valore −1, o più precisamente '**`(clock_t) (-1)`**'.

Per valutare l'intervallo di tempo di utilizzo della CPU, da una certa posizione del programma, a un'altra, occorre memorizzare i valori ottenuti dalla funzione e poi procedere a una sottrazione.

Per comprendere il significato della funzione ***clock()***, del tipo '**`clock_t`**' e della macro-variabile ***CLOCKS_PER_SEC***, viene proposto un esempio molto semplice, ma completo, dove si intende che il tipo '**`clock_t`**' sia intero e sia contenibile in una variabile di tipo '**`long int`**':

```
#include <stdio.h>
#include <time.h>

int
main (int argc, char *argv[])
{
  clock_t t0;
  clock_t t1;
  long int i;
  long int x;

  t0 = clock ();
  printf ("Tempo iniziale: %li/%li\n",
          (long int) t0, (long int) CLOCKS_PER_SEC);

  for (i = 0; i < 10000000; i++)
    {
      x = i * 123;
```

```
    }

  t1 = clock ();
  printf ("Tempo finale: %li/%li\n",
          (long int) t1, (long int) CLOCKS_PER_SEC);

  return 0;
}
```

Avviando questo programma si potrebbe leggere un risultato simile al testo seguente, dove si vede un valore di '**CLOCKS_PER_SEC**' pari a 1000000:

```
Tempo iniziale: 0/1000000
Tempo finale: 20000/1000000
```

69.16.2 Rappresentazione interna del tempo

Generalmente, nei sistemi Unix si tratta il tempo come una quantità di secondi trascorsi a partire da un'epoca di riferimento, che tradizionalmente coincide con l'ora zero del giorno 1 gennaio 1970. Da questo concetto deriva il tipo '**time_t**' della libreria, che, secondo lo standard, rappresenta la quantità di unità di tempo trascorsa a partire da un'epoca di riferimento.

```
typedef long int time_t; // Unità di tempo convenzionale
                         // per le informazioni data-orario.
```

Ammesso che si tratti di un numero intero, così come viene ipotizzato dall'esempio proposto, il rango costituisce il limite alle date rappresentabili. Pertanto, se il tipo '**time_t**' viene dichiarato come numero intero con segno, a 32 bit, per rappresentare una quantità di secondi (come nella tradizione Unix), significa che si possono rap-

presentare al massimo 24855 giorni, pari a circa 68 anni.[8] Se l'epoca di riferimento è il 1970, si può arrivare al massimo al 2038.

69.16.3 Rappresentazione strutturata del tempo

«

La libreria standard prescrive che sia definito il tipo '**`struct tm`**', con il quale è possibile rappresentare tutte le informazioni relative a un certo tempo, secondo le convenzioni umane. Lo standard prescrive con precisione i membri minimi della struttura e l'intervallo di valori che possono contenere:

```
struct tm {
    int tm_sec;     // Secondi:                    da 0 a 60.
    int tm_min;     // Minuti:                     da 0 a 59.
    int tm_hour;    // Ora:                        da 0 a 23.
    int tm_mday;    // Giorno del mese:            da 1 a 31.
    int tm_mon;     // Mese dell'anno:             da 0 a 11.
    int tm_year;    // Anno dal 1900.
    int tm_wday;    // Giorno della settimana: da 0 a 6
                    // con lo zero corrispondente alla
                    // domenica.
    int tm_yday;    // Giorno dell'anno:           da 0 a 365.
    int tm_isdst;   // Ora estiva. Contiene un valore
                    // positivo se è in vigore l'ora estiva;
                    // zero se l'ora è quella «normale»
                    // ovvero quella invernale;
                    // un valore negativo se l'informazione
                    // non è disponibile.
};
```

Si può osservare che il mese viene rappresentato con valori che vanno da 0 a 11, pertanto gennaio si indica con lo zero e dicembre con il numero 11; inoltre, l'intervallo ammesso per i secondi consente di rappresentare un secondo in più, dato che l'intervallo corretto sa-

rebbe da 0 a 59; infine, il fatto che i giorni dell'anno vadano da 0 (il primo) a 365 (l'ultimo), significa che negli anni normali i valori vanno da 0 a 364, mentre negli anni bisestili si arriva a contare fino a 365.

69.16.4 Funzioni per l'elaborazione di valori legati al tempo

Un gruppo di funzioni dichiarate nel file '`time.h`' ha lo scopo di elaborare in qualche modo le informazioni legate al tempo ed eventualmente di convertirle in formati diversi. Queste funzioni trattano il tempo in forma di variabili di tipo '**`time_t`**' o di tipo '**`struct tm`**'.[9]

La variabile di tipo '**`time_t`**' che viene usata in queste funzioni potrebbe esprimere un valore riferito al tempo universale (UT), mentre le funzioni che la utilizzano dovrebbero tenere conto del fuso orario, in base alle informazioni che può offrire il sistema operativo.

69.16.4.1 Funzione «time()»

La funzione ***time()*** determina il tempo attuale secondo il calendario del sistema operativo, restituendolo nella forma del tipo '**`time_t`**'. La funzione richiede un parametro, costituito da un puntatore di tipo '**`time_t *`**': se questo puntatore è valido, la stessa informazione che viene restituita viene anche memorizzata nell'indirizzo indicato da tale puntatore.

```
time_t  time (time_t *timer);
```

In pratica, se è possibile, l'informazione data-orario raccolta dalla funzione, viene anche memorizzata in ****timer***.

Se la funzione non può fornire l'informazione richiesta, allora restituisce il valore −1, o più precisamente: '**`(time_t) (-1)`**'.

69.16.4.2 Funzione «difftime()»

La funzione ***difftime()*** calcola la differenza tra due date, espresse in forma '**`time_t`**' e restituisce l'intervallo in secondi, in una variabile in virgola mobile, di tipo '**`double`**':

```
double  difftime (time_t time1, time_t time0);
```

Per la precisione, viene eseguito ***time1−time0*** e di conseguenza va il segno del risultato.

69.16.4.3 Funzione «mktime()»

La funzione ***mktime()*** riceve come argomento il puntatore a una variabile strutturata di tipo '**`struct tm`**', contenente le informazioni sull'ora locale, e determina il valore di quella data secondo la rappresentazione interna, di tipo '**`time_t`**':

```
time_t  mktime (struct tm *timeptr);
```

La funzione tiene in considerazione solo alcuni membri della struttura; per la precisione, non considera il giorno della settimana e il giorno dell'anno; inoltre, ammette anche valori al di fuori degli intervalli stabiliti per i vari membri della struttura; infine, considera un valore negativo per il membro ***timeptr->tm_isdst*** come la richiesta di determinare se sia o meno in vigore l'ora estiva per la data indicata.

Se la funzione non è in grado di restituire un valore rappresentabile nel tipo '**`time_t`**', o comunque se non può eseguire il suo compito, restituisce il valore −1, o più precisamente '**`(time_t) (-1)`**'.

Se invece tutto procede regolarmente, la funzione provvede anche a correggere i valori dei vari membri della struttura e a ricalcolare il giorno della settimana e dell'anno.

L'esempio successivo mostra la dichiarazione di una variabile strutturata di tipo '**`struct tm`**', assegnando ai suoi membri dei valori non corretti. Con l'aiuto della funzione ***mktime()*** si ricostruisce la data secondo le convenzioni comuni:

```
#include <stdio.h>
#include <time.h>

int
main (int argc, char *argv[])
{
  struct tm t;
  time_t tx;

  t.tm_year = 107;     // 2007 - 1900
  t.tm_mon  = 5;
  t.tm_mday = 33;
  t.tm_hour = 0;
  t.tm_min  = 0;
  t.tm_sec  = 60;
  t.tm_isdst = -1;

  printf ("%i/%i/%i %i:%i:%i\n",
          t.tm_year + 1900, t.tm_mon + 1, t.tm_mday,
          t.tm_hour, t.tm_min, t.tm_sec);

  tx = mktime (&t);

  if (tx == (time_t) (-1))
    {
      printf ("Errore! %li\n", (long int) tx);
```

```
    }
  else
    {
      printf ("%i/%i/%i %i:%i:%i\n",
              t.tm_year + 1900, t.tm_mon + 1, t.tm_mday,
              t.tm_hour, t.tm_min, t.tm_sec);

      printf ("giorno della settimana: %i\n", t.tm_wday);
      printf ("giorno dell'anno: %i\n", t.tm_yday + 1);
      printf ("ora estiva: %i\n", t.tm_isdst);
    }
  return 0;
}
```

Eseguendo questo programma di esempio si dovrebbe ottenere il testo seguente:

```
2007/6/33 0:0:60
2007/7/3 0:1:0
giorno della settimana: 2
giorno dell'anno: 184
ora estiva: 1
```

69.16.4.4 Funzioni «gmtime()» e «localtime()»

«

Le funzioni ***gmtime()*** e ***localtime()*** hanno in comune il fatto di ricevere come argomento il puntatore di tipo '**`time_t *`**', a un'informazione data-orario, per restituire il puntatore a una variabile strutturata di tipo '**`struct tm *`**'. In altri termini, le due funzioni convertono una data espressa nella forma del tipo '**`time_t`**', in una data suddivisa nella struttura '**`struct tm`**':

```
struct tm *gmtime    (const time_t *timer);
struct tm *localtime (const time_t *timer);
```

Nell'ambito di queste funzioni, è ragionevole supporre che l'informazione di tipo '**time_t**' a cui fanno riferimento, sia espressa in termini di tempo universale e che le funzioni stesse abbiano la possibilità di stabilire il fuso orario e la modalità di regolazione dell'ora estiva.

In ogni caso, la differenza tra le due funzioni sta nel fatto che ***gmtime()*** traduce il tempo a cui punta il suo argomento in una struttura contenente la data tradotta secondo il tempo coordinato universale, mentre ***localtime()*** la traduce secondo l'ora locale.

Va osservato che queste funzioni restituiscono un puntatore a un'area di memoria che può essere sovrascritta da altre chiamate alle stesse funzioni o a funzioni simili.

69.16.5 Conversione in stringa

Un piccolo gruppo di funzioni del file '`time.h`' è destinato alla conversione dei valori data-orario in stringhe, per l'interpretazione umana.

69.16.5.1 Funzione «asctime()»

La funzione ***asctime()*** converte un'informazione data-orario, espressa nella forma di una struttura '**struct tm**', in una stringa che esprime l'ora locale, usando però una rappresentazione fissa in lingua inglese:

```
char *asctime (const struct tm *timeptr);
```

In pratica, dal momento che la data e l'orario vanno espressi secondo le convenzioni della lingua inglese, lo standard stesso descrive completamente questa funzione e il listato seguente è tratto letteralmente da tale definizione:

```
#include <time.h>
char *asctime(const struct tm *timeptr)
{
  static const char wday_name[7][3] = {
      "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
  };
  static const char mon_name[12][3] = {
      "Jan", "Feb", "Mar", "Apr", "May", "Jun",
      "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
  };
  static char result[26];
  sprintf(result, "%.3s %.3s%3d %.2d:%.2d:%.2d %i\n",
      wday_name[timeptr->tm_wday],
      mon_name[timeptr->tm_mon],
      timeptr->tm_mday, timeptr->tm_hour,
      timeptr->tm_min, timeptr->tm_sec,
      1900 + timeptr->tm_year);
  return result;
}
```

69.16.5.2 Funzione «ctime()»

«

La funzione ***ctime()*** converte un'informazione data-orario, espressa nella forma del tipo '**`time_t`**' in una stringa che esprime l'ora locale, usando però una rappresentazione fissa in lingua inglese:

```
char *ctime (const time_t *timer);
```

Il comportamento di questa funzione è tale da generare una stringa analoga a quella della funzione ***asctime()***, tanto che la si potrebbe esprimere così:

```
char *ctime (const time_t *timer)
{
  return asctime (localtime (timer));
}
```

Oppure, come macroistruzione, così:

```
#define ctime(t) (asctime (localtime (t)));
```

69.16.5.3 Funzione «strftime()»

La funzione ***strftime()*** si occupa di interpretare il contenuto di una struttura di tipo '**`struct tm`**' e di tradurlo in un testo, secondo una stringa di composizione libera. In altri termini, questa funzione si comporta in modo simile a ***printf()***, dove l'input è costituito dalla struttura contenente le informazioni data-orario.

```
size_t strftime (char * restrict s,
                 size_t maxsize,
                 const char * restrict format,
                 const struct tm * restrict timeptr);
```

Dal modello del prototipo della funzione, si vede che questa restituisce un valore numerico di tipo '**`size_t`**'. Questo valore rappresenta la quantità di elementi[10] che sono stati scritti nella stringa di destinazione, rappresentata dal primo parametro. Dal computo di questi elementi è escluso il carattere nullo di terminazione, benché venga comunque aggiunto dalla funzione.

La funzione richiede, nell'ordine: un array di caratteri da utilizzare per comporre il testo; la dimensione massima di questo array; la stringa di composizione, contenente del testo costante e degli specificatori di conversione; il puntatore alla struttura contenente le informazioni data-orario da usare nella conversione.

La funzione termina il proprio lavoro con successo solo se può scrivere nell'array di destinazione il testo composto secondo le indicazioni della stringa di composizione, includendo anche il carattere nullo di terminazione. Se ciò non avviene, il valore restituito dalla funzione è zero e il contenuto dell'array di destinazione è imprecisato.

Il listato successivo mostra un programma completo che dimostra il funzionamento di ***strftime()***. Va osservato che la conversione eseguita da tale funzione è sensibile alla configurazione locale; precisamente dipende dalla categoria '**`LC_TIME`**':

```
#include <stdio.h>
#include <locale.h>
#include <time.h>

int
main (int argc, char *argv[])
{
  char s[100];
  time_t t      = time (NULL);
  struct tm *tp = localtime (&t);
  int dim;
  setlocale (LC_ALL, "it_IT.UTF-8");
  dim = strftime (s, 100, "Ciao amore: sono "
                          "le %H:%M del %d %B %Y.", tp);
  printf ("%d: %s\n", dim, s);
  return 0;
}
```

Ecco cosa si potrebbe ottenere eseguendo questo programma:

```
45: Ciao amore: sono le 09:32 del 27 giugno 2012.
```

Nella tabella successiva vengono elencati gli specificatori di conversione principali. Sono ammissibili delle varianti, con l'aggiunta di modificatori, che però non vengono descritte. Per esempio è ammissibile l'uso degli specificatori '**`%Ec`**' e '**`%Od`**', per indicare rispettivamente una variante di '**`%c`**' e '**`%d`**'.

Tabella 69.244. Specificatori di conversione usati dalla funzione ***strftime()***.

Specificatore	Corrispondenza
`%C`	*century* Il secolo, ottenuto dividendo l'anno per 100 e ignorando i decimali.
`%y` `%Y`	*year* L'anno: nel primo caso si mostrano solo le ultime due cifre, mentre nel secondo si mostrano tutte.
`%b` | `%h` `%B`	Rispettivamente, il nome abbreviato e il nome per esteso del mese.
`%m`	*month* Il numero del mese, da 01 a 12.
`%d` `%e`	*day* Il giorno del mese, in forma numerica, da 1 a 31, utilizzando sempre due cifre: nel primo caso si aggiunge eventualmente uno zero; nel secondo si aggiunge eventualmente uno spazio.
`%a` `%A`	Rispettivamente, il nome abbreviato e il nome per esteso del giorno della settimana.

Specificatore	Corrispondenza
`%H` `%L`	*hour* L'ora, espressa rispettivamente a 24 ore e a 12 ore.
`%p`	La sigla da usare, secondo la configurazione locale, per specificare che si tratta di un'ora antimeridiana o pomeridiana. Nella convenzione inglese si ottengono, per esempio, le sigle «AM» e «PM».
`%r`	L'ora espressa a 12 ore, completa dell'indicazione se trattasi di ora antimeridiana o pomeridiana, secondo le convenzioni locali.
`%R`	L'ora e i minuti, equivalente a '**`%H:%M`**'.
`%M`	*minute* I minuti, da 00 a 59.
`%S`	*second* I secondi, espresso con valori da 00 a 60.
`%T`	*time* L'ora, i minuti e i secondi, equivalente a '**`%H:%M:%S`**'.
`%z` `%Z`	*time zone* La rappresentazione del fuso orario, nel primo caso come distanza dal tempo coordinato universale (UTC), mentre nel secondo si usa una rappresentazione conforme alla configurazione locale.
`%j`	*julian* Il giorno dell'anno, usando sempre tre cifre numeriche: da 001 a 366.

Specificatore	Corrispondenza
`%g` `%G`	L'anno a cui appartiene la settimana secondo lo standard ISO 8601: nel primo caso si mostrano solo le ultime due cifre, mentre nel secondo si ha l'anno per esteso. Secondo lo standard ISO 8601 la settimana inizia con lunedì e la prima settimana dell'anno è quella che include il 4 gennaio.
`%V`	Il numero della settimana secondo lo standard ISO 8601. I valori vanno da 01 a 53. Secondo lo standard ISO 8601 la settimana inizia con lunedì e la prima settimana dell'anno è quella che include il 4 gennaio.
`%u` `%w`	Il giorno della settimana, espresso in forma numerica, dove, rispettivamente, si conta da 1 a 7, oppure da 0 a 6. Zero e sette rappresentano la domenica; uno è il lunedì.
`%U` `%W`	Il numero della settimana, contando, rispettivamente, dalla prima domenica o dal primo lunedì di gennaio. Si ottengono cifre da 00 a 53.
`%x`	La data, rappresentata secondo le convenzioni locali.
`%X`	L'ora, rappresentata secondo le convenzioni locali.
`%c`	La data e l'ora, rappresentate secondo le convenzioni locali.
`%D`	*date* La data, rappresentata come '**`%m/%d/%Y`**'.
`%F`	La data, rappresentata come '**`%Y-%m-%d`**'.
`%n`	Viene rimpiazzato dal codice di interruzione di riga.
`%t`	Viene rimpiazzato da una tabulazione orizzontale.

Specificatore	Corrispondenza
`%%`	Viene rimpiazzato da un carattere di percentuale.

69.17 File «stdio.h»

«

Il file ‘`stdio.h`’ della libreria standard è quello che fornisce le funzioni più importanti e in generale è il più complesso da realizzare, in quanto dipende strettamente dal meccanismo di gestione dei file del sistema operativo (si veda eventualmente la realizzazione del file ‘`stdio.h`’ e di alcune delle sue funzioni nei sorgenti di os32, sezione 95.18). L’elemento più delicato che viene definito qui è il tipo di dati ‘**`FILE`**’, da cui dipende quasi tutto il resto.

Alle complicazioni che esistevano già alla nascita del linguaggio, nei primi sistemi Unix, si aggiungono attualmente quelle relative alla distinzione tra file di testo e file binari, oltre che quelle relative alla gestione dei caratteri multibyte, per cui la lettura o la scrittura attraverso un flusso di dati deve tenere conto dello stato di completamento di tali informazioni.

Il file ‘`stdio.h`’ definisce le funzioni principali per l’accesso ai file e una serie di funzioni per la lettura e scrittura di dati formattati (si vedano ***print()***, ***scanf()*** e altre analoghe), ma altre funzioni realizzate espressamente per caratteri e stringhe estese (formate da elementi ‘**`wchar_t`**’) si trovano nel file ‘`wchar.h`’.

I file proposti che si basano sugli esempi del capitolo sono incompleti, in quanto manca la dichiarazione del tipo ‘**`FILE`**’ e del tipo ‘**`fpos_t`**’.

69.17.1 Tipi

Il file ‘`stdio.h`’, oltre a ‘**`size_t`**’ che fa già parte del file ‘`stddef.h`’, e di ‘**`va_list`**’ che fa già parte del file ‘`stdarg.h`’, dichiara due tipi di dati a uso specifico per la gestione dei file: ‘**`FILE`**’ e ‘**`fpos_t`**’, realizzati normalmente attraverso delle strutture.

Il tipo ‘**`fpos_t`**’ serve a rappresentare tutte le informazioni necessarie a specificare univocamente le posizioni interne a un file, per gli scopi delle funzioni ***fgetpos()*** e ***fsetpos()***. Il tipo ‘**`FILE`**’ deve poter esprimere tutte le informazioni necessarie a controllare un flusso di file (ovvero le operazioni su un file aperto), in particolare le posizioni correnti, il puntatore alla memoria tampone (*buffer*), l’indicatore di errore e di fine file.

```
typedef struct { /* omissis */ } fpos_t;

typedef struct { /* omissis */ } FILE;
```

L’organizzazione effettiva delle strutture che costituiscono i tipi ‘**`fpos_t`**’ e ‘**`FILE`**’ dipende strettamente dal sistema operativo (nel contesto particolare della propria architettura); pertanto, per poterne approfondire le caratteristiche, occorre prima uno studio dettagliato delle funzionalità del sistema operativo stesso.

Alcune funzioni aggiunte dallo standard POSIX utilizzano anche il tipo ‘**`off_t`**’, che è descritto nel file di intestazione ‘`sys/types.h`’.

69.17.2 Macro-variabili varie

Il file ‘`stdio.h`’ dichiara la macro-variabile ***NULL***, come già avviene nel file ‘`stddef.h`’, assieme ad altre macro-variabili a uso delle funzioni dichiarate al proprio interno. Quelle più semplici sono descritte nella tabella 69.248. L’esempio proposto della dichiarazione di tali macro-variabili è molto approssimativo:

```
#define _IOFBF          0 // Input-output fully buffered.
#define _IOLBF          1 // Input-output line buffered.
#define _IONBF          2 // Input-output with no buffering.

#define EOF             (-1)

#define FOPEN_MAX       10
#define FILENAME_MAX    254
#define L_tmpnam        FILENAME_MAX

#define SEEK_SET        0       // Dall'inizio.
#define SEEK_CUR        1       // Dalla posizione corrente.
#define SEEK_END        2       // Dalla fine del file.

#define TMP_MAX         100000  // Si ipotizza di usare nomi
                                // da «TMP00000.tmp» a
                                // «TMP99999.tmp».
```

POSIX La porzione successiva riguarda le estensioni POSIX:

```
#define L_ctermid       14
#define P_tmpdir        "/tmp"
```

Tabella 69.248. Macro-variabili comuni per le funzioni di ‘stdio.h’.

Denominazione	Significato mnemonico	Descrizione
_IOFBF	*input output fully buffered*	Indica simbolicamente la richiesta di utilizzo di una memoria tampone a blocchi.
_IOLBF	*input output line buffered*	Indica simbolicamente la richiesta di utilizzo di una memoria tampone gestita a righe di testo.
_IONBF	*input output with no buffering*	Indica simbolicamente la richiesta di non utilizzare alcuna memoria tampone.
BUFSIZE	*buffer size*	Rappresenta la dimensione predefinita della memoria tampone.
EOF	*end of file*	È un numero intero di tipo ‘**int**’, negativo, che rappresenta il raggiungimento della fine del file. È in pratica ciò che si ottiene leggendo oltre la fine del file.
FOPEN_MAX	*file open max*	Il numero di file che un processo elaborativo può aprire simultaneamente, in base alle limitazioni poste dal sistema operativo.

Denominazione	Significato mnemonico	Descrizione
`FILENAME_MAX`		La dimensione di un array di elementi '**`char`**', tale da essere abbastanza grande da contenere il nome del file più lungo (incluse le eventuali sequenze multibyte) che il sistema consenta di gestire.
`L_tmpnam`	*temporary name*	La dimensione di un array di elementi '**`char`**', tale da essere abbastanza grande da contenere il nome di un file temporaneo generato dalla funzione ***tmpnam()***.
`SEEK_CUR`	*seek current*	Indica di eseguire un posizionamento a partire dalla posizione corrente del file.
`SEEK_END`		Indica di eseguire un posizionamento a partire dalla fine di un file.
`SEEK_SET`		Indica di eseguire un posizionamento a partire dall'inizio di un file.
`TMP_MAX`		Rappresenta la quantità massima di nomi di file differenti che possono essere generati dalla funzione ***tmpnam()***.

Tabella 69.249. Macro-variabili aggiunte dalle estensioni POSIX.

Denominazione	Significato mnemonico	Descrizione
`L_ctermid`	*character terminal identity*	La dimensione di un array di elementi '**`char`**', tale da essere abbastanza grande da contenere il nome del file di dispositivo restituito dalla funzione ***ctermid()***.
`P_tmpdir`	*temporary directory*	Definisce il percorso di una directory temporanea, da scegliere quando ciò che viene specificato con la funzione ***tempnam()*** non è appropriato.

69.17.3 Ipotesi di gestione del tipo «FILE»

Pur non essendo necessario che sia così, si può ipotizzare che per ogni file che possa essere aperto simultaneamente, sia disponibile un elemento di tipo '**`FILE`**' organizzato in un array. In tal caso, potrebbe essere dichiarato come nell'esempio seguente, già nel file '`stdio.h`', anche se il nome usato per l'array è puramente indicativo:

```
...
FILE _stream[FOPEN_MAX];
...
```

L'uso della macro-variabile ***FOPEN_MAX*** garantisce che siano predisposti esattamente tutti gli elementi necessari alla gestione simultanea del limite di file previsti.

69.17.4 Flussi standard

«

Lo standard del linguaggio C prescrive che i nomi dei flussi standard previsti siano delle macro-variabili, tali da espandersi in espressioni che rappresentino puntatori di tipo '**FILE ***', diretti ai flussi standard rispettivi. Nel caso del compilatore GNU C i puntatori sono già definiti con lo stesso nome dei flussi e, nel file 'stdio.h' vi si fa riferimento in qualità di variabili esterne (in quanto dichiarate nella libreria precompilata):

```
extern FILE *stdin;      // Si ipotizza che la libreria C
                         // definisca già i puntatori ai
extern FILE *stdout;     // flussi standard, usando
extern FILE *stderr;     // i nomi predefiniti.
                         //
#define stdin  stdin     // In questo caso, è facile definire
#define stdout stdout    // le macro che fanno riferimento
#define stderr stderr    // ai flussi standard.
```

Diversamente, nell'ipotesi in cui si gestisca un array di elementi '**FILE**', si potrebbe supporre che i primi tre elementi siano usati per i flussi standard e in tal caso le dichiarazioni delle macro-variabili potrebbero essere fatte così:

```
...
#define stdin  (&_stream[0])
#define stdout (&_stream[1])
#define stderr (&_stream[2])
...
```

69.17.5 Funzioni per la rimozione e la ridenominazione dei file

Le funzioni ***remove()*** e ***rename()*** consentono, rispettivamente di eliminare o di rinominare un file. Il file in questione viene individuato da una stringa, il cui contenuto deve conformarsi alle caratteristiche del sistema operativo. Le due funzioni hanno in comune il fatto di restituire un valore intero (di tipo '**int**'), dove il valore zero rappresenta il completamento con successo dell'operazione, mentre un valore differente indica un fallimento.

```
int remove (const char *filename);
int rename (const char *old, const char *new);
```

La sintassi per l'uso della funzione ***remove()*** è evidente dal suo prototipo, in quanto si attende un solo argomento che è costituito dal nome del file da eliminare; nel caso della funzione ***rename()***, invece, il primo argomento è il nome del file preesistente e il secondo è quello che si vuole attribuirgli.

È importante ribadire che il comportamento delle due funzioni dipende dal sistema operativo. Per esempio, la ridenominazione può provocare la cancellazione di un file preesistente con lo stesso nome che si vorrebbe attribuire a un altro, oppure potrebbe limitarsi a fallire. In un sistema Unix o simile, molto dipende dalla configurazione dei permessi.

69.17.6 Funzioni per la gestione dei file temporanei

Le funzioni ***tmpfile()*** e ***tmpnam()*** servono per facilitare la creazione di file temporanei. La prima crea automaticamente un file di cui non si conosce il nome e la collocazione, aprendolo in aggiorna-

mento (modalità '**wb+**'); la seconda si limita a generare un nome che potrebbe essere usato per creare un file temporaneo:

```
FILE *tmpfile (void);
char *tmpnam  (char *s);
```

L'uso della funzione ***tmpfile()*** è evidente, in quanto non richiede argomenti e restituisce il puntatore al file creato; la seconda richiede l'indicazione di un array di caratteri da poter modificare, restituendo comunque il puntatore all'inizio dello stesso array. In ogni caso va chiarito che il file creato con la funzione '**tmpfile**', una volta chiuso, viene rimosso automaticamente.

Le due funzioni devono essere in grado di poter generare un numero di nomi differente pari almeno al valore rappresentato da '**TMP_MAX**', rimanendo il fatto che non possano essere aperti più di '**FOPEN_MAX**' file e che non possono essere generati file con nomi già esistenti.

Se si utilizza la funzione ***tmpnam()***, l'array di caratteri che costituisce il primo argomento (***s***), viene usato dalla funzione per scriverci il nome del file temporaneo, restituendone poi il puntatore; tale array deve avere una dimensione di almeno '**L_tmpnam**' elementi, come si vede nell'esempio seguente:

```
#include <stdio.h>
int main (void)
{
  char t[L_tmpnam];
  char *p;
  p = tmpnam (t);
  printf ("%s %s\n", t, p);
  return 0;
}
```

Se la funzione ***tmpnam()*** riceve come argomento il puntatore nullo, il nome del file temporaneo viene scritto in un'area di memoria statica che viene sovrascritta a ogni chiamata successiva della funzione stessa.

Entrambe le funzioni, se non possono eseguire il loro compito, restituiscono un puntatore nullo.

Le estensioni POSIX aggiungono anche la funzione ***tempnam()***, la quale ha un comportamento simile a quello di ***tmpnam()***, in quanto non crea il file, ma restituisce il percorso del file che potrebbe essere creato:

```
char *tempnam (const char *dir, const char *prefix);
```

La funzione ***tempnam()*** richiede l'indicazione di una stringa contenente il percorso di una directory, in cui si vuole sia creato un file temporaneo. Se la stringa indica una directory inadatta (perché non esiste o non è accessibile o non gli si può scrivere) oppure si indica il puntatore nullo, si fa riferimento a quanto descritto dalla macro-variabile ***P_tmpdir***. Se anche la directory a cui si riferisce la macro-variabile ***P_tmpdir*** dà dei problemi, è possibile che la funzione decida in qualche modo dove sia possibile collocare un file temporaneo.

Il secondo argomento della funzione è una stringa che rappresenta un prefisso da usare per il nome del file temporaneo. Tale prefisso può essere lungo al massimo cinque caratteri. Se si vuole omettere tale prefisso, basta indicare il puntatore nullo.

Se tutto va bene, la funzione alloca dello spazio in memoria per la stringa che deve contenere il percorso di un file temporaneo che si potrebbe creare (ma senza crearlo) e ne restituisce il puntatore.

Quando tale informazione non serve più, la memoria allocata può essere liberata con la funzione ***free()***. Se invece la funzione fallisce nel suo compito, restituisce il puntatore nullo e aggiorna la variabile ***errno***.

69.17.7 Funzioni per l'apertura e la chiusura dei flussi di file

«

Le funzioni ***fopen()***, ***freopen()*** e ***fclose()***, consentono di aprire e chiudere i file, gestendoli attraverso un puntatore al ***flusso di file*** loro associato (*stream*). Il puntatore in questione è di tipo '**`FILE *`**'.

```
FILE *fopen   (const char *restrict filename,
               const char *restrict io_mode);

FILE *freopen (const char *restrict filename,
               const char *restrict io_mode,
               FILE *restrict stream);

int   fclose  (FILE *stream);
```

Quando viene aperto un file, gli si associa una variabile strutturata di tipo '**`FILE`**', contenente tutte le informazioni che servono a gestirne l'accesso. Questa variabile deve rimanere univoca e vi si accede normalmente attraverso un puntatore ('**`FILE *`**'). Dal momento che per il linguaggio C un file aperto è un flusso, la variabile strutturata che contiene le informazioni necessarie a gestirne l'accesso viene identificata come il flusso stesso, pertanto nei prototipi la variabile che contiene il puntatore di tipo '**`FILE *`**' viene denominata generalmente ***stream***.

Dal momento che non è compito del programmatore dichiarare la variabile di tipo '**FILE**', in pratica ci si riferisce al flusso di file sempre solo attraverso un puntatore a quella variabile. Pertanto, è più propriamente il puntatore a tale variabile che rappresenta il flusso di file.

L'apertura di un file, oltre che l'indicazione del nome del file, richiede di specificare la modalità, ovvero il tipo di accesso che si intende gestire. Sono previste le modalità elencate nella tabella 69.258.

Tabella 69.258. Modalità di accesso ai file.

Sigla	Mnemonico	Descrizione
`r`	*read*	Accesso in sola lettura di un file di testo.
`w`	*write*	Accesso a un file di testo in scrittura, che implica la creazione del file all'apertura, ovvero il suo troncamento a zero, se esiste già.
`a`	*append*	Accesso a un file di testo in aggiunta, che implica la creazione del file all'apertura, ovvero la sua estensione se esiste già.
`rb` `wb` `ab`	*binary*	Accesso in lettura, scrittura o aggiunta, ma di tipo binario.

Sigla	Mnemonico	Descrizione
`r+` `w+` `a+`	*update*	Accesso a un file di testo in lettura, scrittura o aggiunta, assieme alla modalità di aggiornamento. In pratica, con la lettura è consentita anche la scrittura; con la scrittura e l'aggiunta è consentita anche la rilettura.
`rb+` | `r+b` `wb+` | `w+b` `ab+` | `a+b`		Accesso a un file binario in lettura, scrittura o aggiunta, assieme alla modalità di aggiornamento. In pratica, con la lettura è consentita anche la scrittura; con la scrittura e l'aggiunta è consentita anche la rilettura. Si può osservare che il segno '**+**' può essere messo indifferentemente in mezzo o alla fine.

La funzione ***fopen()*** apre il file indicato come primo argomento (una stringa), con la modalità specificata nel secondo (un'altra stringa), restituendo il puntatore al flusso che consente di accedervi (se l'operazione fallisce, la funzione restituisce il puntatore nullo). La modalità di accesso viene espressa attraverso le sigle elencate nella tabella 69.258.

La funzione ***freopen()*** consente di associare un file differente a un flusso già esistente, cambiando anche la modalità di accesso, cosa che viene fatta normalmente per ridirigere i flussi standard. I primi due argomenti della funzione sono gli stessi di ***fopen()***, con l'aggiunta alla fine del puntatore al flusso che si vuole ridirigere. La

funzione restituisce il puntatore al flusso ridiretto se l'operazione ha successo, altrimenti produce soltanto il puntatore nullo. Se nel primo argomento, al posto di indicare il nome del file, si mette un puntatore nullo, la chiamata della funzione serve solo per modificare la modalità di accesso a un file già aperto, senza ridirigerne il flusso. Va osservato che il cambiamento della modalità di accesso, in ogni caso, dipende dal sistema operativo e non è detto che si possano applicare tutte le combinazioni.

La funzione ***fclose()*** permette di chiudere il flusso indicato come argomento, restituendo un valore numerico pari a zero se l'operazione ha successo, oppure il valore corrispondente alla macro-variabile ***EOF*** in caso contrario. La chiusura di un flusso implica la scrittura di dati rimasti in sospeso (in una memoria tampone). Un flusso già chiuso non deve essere chiuso nuovamente.

Dal momento che lo standard POSIX introduce il concetto di descrittore di file, per poter associare un flusso di file a un file già aperto come descrittore, si usa la funzione ***fdopen()***, mentre per fare il contrario, si usa la funzione ***fileno()***: POSIX

```
FILE *fdopen  (int fdn, const char *io_mode);
int   fileno  (FILE *stream);
```

La funzione ***fdopen()*** richiede l'indicazione del numero del descrittore e della modalità di accesso, la quale deve essere compatibile con quanto già definito a proposito del descrittore stesso. L'associazione tra flusso di file e descrittore comporta inizialmente l'azzeramento dell'indicatore di errore e di quello di fine file; inoltre, se viene chiuso il flusso di file, si ottiene automaticamente la chiusura del descrittore relativo.

La funzione ***fileno()*** restituisce il numero di descrittore associato a un flusso di file già aperto. Se però l'operazione fallisce, restituisce il valore −1 e aggiorna la variabile ***errno***.

69.17.8 Funzioni per la gestione della memoria tampone

Le funzioni ***setvbuf()*** e ***setbuf()*** consentono di attribuire una memoria tampone (*buffer*) a un certo flusso di dati (un file già aperto), mentre ***fflush()*** consente di richiedere espressamente lo scarico della memoria in modo che le operazioni sospese di scrittura siano portate a termine completamente.

```
int  setvbuf (FILE *restrict stream, char *restrict buffer,
              int buf_mode, size_t size);

void setbuf  (FILE *restrict stream, char *restrict buffer);

int  fflush  (FILE *stream);
```

La funzione ***setvbuf()*** permette di attribuire una memoria tampone a un file che è appena stato aperto e per il quale non è ancora stato eseguito alcun accesso. Il primo argomento della funzione è il puntatore al flusso relativo e il secondo è il puntatore all'inizio dell'array di caratteri da usare come memoria tampone. Se al posto del riferimento alla memoria tampone si indica un puntatore nullo, si intende che la funzione debba allocare automaticamente lo spazio necessario; se invece l'array viene fornito, è evidente che deve rimanere disponibile per tutto il tempo in cui il flusso rimane aperto.

Il terzo argomento atteso dalla funzione ***setvbuf()*** è un numero che esprime la modalità di funzionamento della memoria tampone. Questo numero viene fornito attraverso l'indicazione di una tra le macrovariabili ***_IOFBF***, ***_IOLBF*** e ***_IONBF***. Il quarto argomento indica

la dimensione dell'array da usare come memoria tampone: se l'array viene fornito effettivamente, si tratta della dimensione che può essere utilizzata; altrimenti è la dimensione richiesta per l'allocazione automatica.

La funzione ***setvbuf()*** restituisce zero se l'operazione richiesta è eseguita con successo; in caso contrario restituisce un valore differente.

La funzione ***setbuf()*** è una semplificazione di ***setvbuf()*** che non restituisce alcun valore, dove al posto di indicare la modalità di gestione della memoria tampone, si intende implicitamente quella corrispondente alla macro-variabile ***_IOFBF*** (pertanto si tratta di una gestione completa della memoria tampone), mentre al posto di indicare la dimensione dell'array che costituisce la memoria tampone si intende il valore corrispondente alla macro-variabile ***BUFSIZ***. In pratica, è come utilizzare la funzione ***setvbuf()*** così:

```
(void) setvbuf (stream, buffer, _IOFBF, BUFSIZ);
```

La funzione '**fflush**' si usa per i file aperti in scrittura, allo scopo di aggiornare i file se ci sono dati sospesi nella memoria tampone che devono ancora essere trasferiti effettivamente. La funzione si attende come argomento il puntatore al flusso per il quale eseguire questo aggiornamento, ma se si fornisce il puntatore nullo (la macro-variabile ***NULL***), si ottiene l'aggiornamento di tutti i file aperti in scrittura. A parte questo, la funzione non altera lo stato del flusso.

La funzione ***fflush()*** restituisce zero se riesce a completare con successo il proprio compito, altrimenti restituisce il valore corrispondente a ***EOF*** e aggiorna la variabile individuata dall'espressione

errno in modo da poter risalire al tipo di errore che si è presentato.

La funzione ***fflush()*** interviene solo nella memoria tampone gestita internamente al programma, ma bisogna tenere presente che il sistema operativo potrebbe gestire un'altra memoria del genere, per il cui scarico occorre eventualmente intervenire con funzioni specifiche del sistema stesso.

69.17.9 Funzioni per la composizione dell'output

«

Alcune funzioni del file '`stdio.h`' sono realizzate con lo scopo principale di comporre una stringa attraverso l'inserzione di componenti di vario genere, convertendo i dati in modo da poterli rappresentare in forma «tipografica», nel senso di sequenza di caratteri che hanno una rappresentazione grafica.

Queste funzioni hanno in comune una stringa contenente degli ***specificatori di conversione***, caratterizzati dal fatto che iniziano con il simbolo di percentuale ('`%`') e dalla presenza di un elenco indefinito di argomenti, il cui valore viene utilizzato in sostituzione degli specificatori di conversione. Il modo in cui si esprime uno specificatore di conversione può essere complesso, pertanto viene mostrato un modello sintattico che descrive la sua struttura:

`%` [***simbolo***] [***n_ampiezza***] [**.** ***n_precisione***] [`hh` | `h` | `l` | `ll` | `j` | `z` | `t` | `L`] ***tipo***

La prima cosa da individuare in uno specificatore di conversione è il tipo di argomento che viene interpretato e, di conseguenza, il genere di rappresentazione che se ne vuole produrre. Il tipo viene espresso

da una lettera alfabetica, alla fine dello specificatore di conversione. La tabella successiva riepiloga i tipi principali.

Tabella 69.261. Tipi di conversione principali.

Simbolo	Tipo di argomento	Conversione applicata
`%…d` `%…i`	`int`	Numero intero con segno da rappresentare in base dieci.
`%…u`	`unsigned int`	Numero intero senza segno da rappresentare in base dieci.
`%…o`	`unsigned int`	Numero intero senza segno da rappresentare in ottale (senza lo zero iniziale che viene usato spesso per caratterizzare un tale tipo di rappresentazione).
`%…x` `%…X`	`unsigned int`	Numero intero senza segno da rappresentare in esadecimale (senza il prefisso '**`0x`**' o '**`0X`**' che viene usato spesso per caratterizzare un tale tipo di rappresentazione).
`%…c`	`int`	Un carattere singolo, dopo la conversione in '**`unsigned char`**'.
`%…s`	`char *`	Una stringa.
`%…f`	`double`	Un numero a virgola mobile, da rappresentare in notazione decimale fissa: **[**`-`**]** ***iii . dddddd***

Simbolo	Tipo di argomento	Conversione applicata
`%...e` `%...E`	`double`	Un numero a virgola mobile, da rappresentare in notazione esponenziale: **[**−**]** *i* **.** *dddddd* **e**±*xx* **[**−**]** *i* **.** *dddddd* **E**±*xx*
`%...g` `%...G`	`double`	Un numero a virgola mobile, rappresentato in notazione decimale fissa o in notazione esponenziale, a seconda di quale si presti meglio in base ai vincoli posti da altri componenti dello specificatore di conversione.
`%p`	`void *`	Un puntatore generico rappresentato in qualche modo in forma grafica.
`%n`	`int *`	Questo specificatore non esegue alcuna conversione e si limita a memorizzare un valore intero (di tipo '**`int`**') nella variabile a cui punta l'argomento. Per la precisione, viene memorizzata la quantità di caratteri generati fino a quel punto dalla conversione.
`%%`		Questo specificatore si limita a produrre un carattere di percentuale ('**`%`**') che altrimenti non sarebbe rappresentabile.

Nel modello sintattico che descrive lo specificatore di conversione, si vede che subito dopo il segno di percentuale può apparire un simbolo

(*flag*). I simboli principali che possono essere utilizzati sono descritti nella tabella successiva.

Tabella 69.262. Alcuni simboli, o *flag*.

Simbolo	Corrispondenza
%+... %#+... %+0***ampiezza***... %#+0***ampiezza***...	Il segno «+» fa sì che i numeri con segno lo mostrino anche se è positivo. Può combinarsi con lo zero e il cancelletto.
%0***ampiezza***... %+0***ampiezza***... %#0***ampiezza***... %#+0***ampiezza***...	Lo zero fa sì che siano inseriti degli zeri a sinistra per allineare a destra il valore, nell'ambito dell'ampiezza specificata. Può combinarsi con il segno «+» e il cancelletto.
%***ampiezza***... % ***ampiezza***...	In mancanza di uno zero iniziale, in presenza dell'indicazione dell'ampiezza, il valore viene allineato a destra usando degli spazi. È possibile esprimere esplicitamente l'intenzione di usare gli spazi mettendo proprio uno spazio, ma in generale non è richiesto. Se si mette lo spazio letteralmente, questo non è poi compatibile con lo zero, mentre le combinazioni con gli altri simboli sono ammissibili.

Simbolo	Corrispondenza
%−*ampiezza*... %−+*ampiezza*... %#−*ampiezza*... %#−+*ampiezza*...	Il segno meno, usato quando la conversione prevede l'uso di una quantità fissa di caratteri con un valore che appare di norma allineato a destra, fa sì che il risultato sia allineato a sinistra. Il segno meno si può combinare il segno «+» e il cancelletto.
%#...	Il cancelletto richiede una modalità di rappresentazione alternativa, ammesso che questa sia prevista per il tipo di conversione specificato. È compatibili con gli altri simboli, ammesso che il suo utilizzo serva effettivamente per ottenere una rappresentazione alternativa.

Subito prima della lettera che definisce il tipo di conversione, possono apparire una o due lettere che modificano la lunghezza del valore da interpretare (per lunghezza si intende qui la quantità di byte usati per rappresentarlo). Per esempio, '`%...Lf`' indica che la conversione riguarda un valore di tipo '**`long double`**'. Tra questi specificatori della lunghezza del dato in ingresso ce ne sono alcuni che indicano un rango inferiore a quello di '**`int`**', come per esempio '`%...hhd`' che si riferisce a un numero intero della dimensione di un '**`signed char`**'; in questi casi occorre comunque considerare che nella trasmissione degli argomenti alle funzioni interviene sempre la promozione a intero, pertanto viene letto il dato della dimensione specificata, ma viene «consumato» il risultato ottenuto dalla promozione. La tabella successiva riepiloga i modificatori di lunghezza principali.

Tabella 69.263. Alcuni modificatori della lunghezza del dato in ingresso.

Simbolo	Tipo	Simbolo	Tipo
%…hhd %…hhi	signed char	%…hhu %…hho %…hhx| %…hhX	unsigned char
%…hd %…hi	short int	%…hu %…ho %…hx|%…hX	unsigned short int
%…ld %…li	long int	%…lu %…lo %…lx|%…lX	unsigned long int
%…lc	wint_t	%…ls	wchar_t *
%…lld %…lli	long long int	%…llu %…llo %…llx| %…llX	unsigned long long int

Simbolo	Tipo	Simbolo	Tipo
`%…jd` `%…ji`	`intmax_t`	`%…ju` `%…jo` `%…jx` | `%…jX`	`uintmax_t`
`%…zd` `%…zi`	`size_t`	`%…zu` `%…zo` `%…zx` | `%…zX`	`size_t`
`%…td` `%…ti`	`ptrdiff_t`	`%…tu` `%…to` `%…tx` | `%…tX`	`ptrdiff_t`
`%…Le` | `%…LE` `%…Lf` | `%…LF` `%…Lg` | `%…LG`	`long double`		

I modificatori di lunghezza si possono utilizzare anche con il tipo ‘**`%…n`**’. In tal caso, si intende che il puntatore sia del tipo specificato dalla lunghezza. Per esempio, ‘**`%tn`**’ richiede di memorizzare la quantità di byte composta fino a quel punto in una variabile di tipo ‘**`ptrdiff_t`**’, a cui si accede tramite il puntatore fornito.

Tra il simbolo (*flag*) e il modificatore di lunghezza può apparire un numero che rappresenta l’ampiezza da usare nella trasformazione

ed eventualmente la precisione: '***ampiezza*** **[** **.** ***precisione*** **]** '. Il concetto parte dalla rappresentazione dei valori in virgola mobile, dove l'ampiezza indica la quantità complessiva di caratteri da usare e la precisione indica quanti di quei caratteri usare per il punto decimale e le cifre successive, ma si applica anche alle stringhe.

In generale, per quanto riguarda la rappresentazione di valori numerici, la parte intera viene sempre espressa in modo completo, anche se l'ampiezza indicata è inferiore; ai numeri interi la precisione non si applica; per i numeri in virgola mobile con rappresentazione esponenziale, la precisione riguarda le cifre decimali che precedono l'esponente; per le stringhe la precisione specifica la quantità di caratteri da considerare, troncando il resto.

In un altro capitolo, la tabella 67.26 riporta un elenco di esempi di utilizzo della funzione ***printf()*** dove si può valutare l'effetto dell'indicazione dell'ampiezza e della precisione.

L'ampiezza, o la precisione, o entrambe, potrebbero essere indicate da un asterisco, come per esempio '**`%*.*f`**'. L'asterisco usato in questo modo indica che il valore corrispondente (ampiezza, precisione o entrambe) viene tratto dagli argomenti come intero ('**`int`**'). Pertanto, per tornare all'esempio composto come '**`%*.*f`**', dagli argomenti viene prelevato un intero che rappresenta l'ampiezza, un altro intero che rappresenta la precisione, quindi si preleva un valore '**`double`**' che è quanto va rappresentato secondo l'ampiezza e la precisione richieste.

69.17.9.1 Funzioni che ricevono gli argomenti direttamente

«

Un gruppo di funzioni per la composizione dell'output riceve direttamente gli argomenti variabili che servono agli specificatori di conversione:

```
int sprintf  (char *restrict s, const char *restrict format,
              ...);
int snprintf (char *restrict s, size_t n,
              const char *restrict format, ...);
int fprintf  (FILE *restrict stream,
              const char *restrict format, ...);
int printf   (const char *restrict format, ...);
```

Tutte le funzioni di questo gruppo hanno in comune la stringa di composizione, costituita dal parametro ***format***, e gli argomenti successivi che sono in quantità e qualità indeterminata, in quanto per la loro interpretazione contano gli specificatori di conversione inseriti nella stringa di composizione. Inoltre, tutte queste funzioni restituiscono la quantità di caratteri prodotti dall'elaborazione della stringa di composizione. Va osservato che il conteggio riguarda solo i caratteri e non include, eventualmente, il carattere nullo di terminazione di stringa che viene usato per le funzioni ***sprintf()*** e ***snprintf()***. Se durante il procedimento di composizione si verifica un errore, queste funzioni possono restituire un valore negativo.

La funzione ***sprintf()*** produce il risultato della composizione memorizzandolo a partire dal puntatore indicato come primo parametro (***s***) e aggiungendo il carattere nullo di terminazione. La funzione ***snprintf()***, invece, produce al massimo $n-1$ caratteri, aggiungendo sempre il carattere nullo di terminazione.

La funzione ***fprintf()*** scrive il risultato della composizione attraverso il flusso di file ***stream***, mentre ***printf()*** lo scrive attraverso lo standard output.

69.17.9.2 Funzioni che ricevono gli argomenti da un'altra funzione

A fianco delle funzioni descritte nella sezione precedente, un gruppo analogo svolge le stesse operazioni, ma ricevendo gli argomenti variabili per riferimento. In pratica si tratta di ciò che serve quando gli argomenti variabili sono stati ottenuti da un'altra funzione e non da una chiamata diretta.

```
int vsprintf  (char *restrict s,
               const char *restrict format, va_list arg);
int vsnprintf (char *restrict s, size_t n,
               const char *restrict format, va_list arg);
int vfprintf  (FILE *restrict stream,
               const char *restrict format, va_list arg);
int vprintf   (const char *restrict format, va_list arg);
```

Il funzionamento è conforme a quello delle funzioni che non hanno la lettera '**v**' iniziale; per esempio, ***vsprintf()*** si comporta conformemente a ***sprintf()***. Per comprendere la differenza si potrebbe dimostrare la realizzazione ipotetica della funzione ***printf()*** avvalendosi di ***vprintf()***:

```
int printf   (const char *restrict format, ...)
{
  va_list arg;
  va_start (arg, format);
  int count;
  count = vprintf (format, arg);
  va_end (arg);
  return count;
}
```

69.17.10 Funzioni per l'interpretazione dell'input

«

Un piccolo gruppo di funzioni del file 'stdio.h' è specializzato nell'interpretazione di una stringa, dalla quale si vanno a estrapolare dei componenti da collocare in variabili di tipo opportuno. In altri termini, da una stringa che rappresenta un valore espresso attraverso caratteri grafici, si vuole estrarre il valore e assegnare a una certa variabile.

Il meccanismo è opposto a quello usato dalle funzioni del tipo '...**printf()**' e anche in questo caso si parte da una stringa contenente principalmente degli specificatori di conversione, seguita da un numero indefinito di argomenti. Gli specificatori delle funzioni che interpretano l'input sono simili a quelli usati per la composizione dell'output, ma non possono essere equivalenti in tutto. Sinteticamente si possono descrivere così:

```
%[*] [n_ampiezza] [hh|h|l|ll|j|z|t|L] tipo
```

Come si può vedere, all'inizio può apparire un asterisco, il cui scopo è quello di annullare l'assegnamento del valore a una variabile. In

pratica, con l'asterisco il dato corrispondente allo specificatore viene interpretato, ma poi non viene salvato.

Successivamente può apparire un numero che rappresenta l'ampiezza del dato da interpretare, in byte, il cui scopo è quello di limitare la lettura fino a un certo carattere (inteso come '**`char`**', pertanto le sequenze multibyte contano per più di una unità singola).

Dopo può apparire una sigla, composta da una o più lettere, il cui scopo è quello di modificare la dimensione predefinita della variabile di destinazione. In altri termini, senza questo modificatore si intende che la variabile ricevente debba essere di una certa grandezza, ma con l'aggiunta del «modificatore di lunghezza» si precisa invece qualcosa di diverso. In pratica, il modificatore di lunghezza usato da queste funzioni è equivalente a quello delle funzioni di composizione dell'output.

Al termine dello specificatore di conversione appare una lettera che dichiara come deve essere interpretato il dato in ingresso e, in mancanza del modificatore di lunghezza, indica anche la dimensione della variabile ricevente.

Tabella 69.267. Tipi di conversione principali.

Simbolo	Tipo di argomento	Conversione applicata
`%…d`	`int *`	Numero intero con segno rappresentato in base dieci.
`%…i`	`int *`	Numero intero con segno rappresentare in base dieci o in base otto, avendo come prefisso uno zero, oppure in base sedici, avendo come prefisso '**`0x`**' o '**`0X`**'.

Simbolo	Tipo di argomento	Conversione applicata
`%…u`	`unsigned int *`	Numero intero senza segno rappresentato in base dieci.
`%…o`	`unsigned int *`	Numero intero senza segno rappresentato in ottale (con o senza lo zero iniziale).
`%…x`	`unsigned int *`	Numero intero senza segno rappresentato in esadecimale (con o senza il prefisso '**`0x`**' o '**`0X`**').
`%…c`	`char *`	Interpreta un solo carattere, o più caratteri se si specifica l'ampiezza. Nella lettura contano anche gli spazi o qualunque altro carattere e non viene aggiunto il carattere nullo di terminazione.
`%…s`	`char *`	Interpreta una sequenza di caratteri che non siano spazi, aggiungendo alla fine il carattere nullo di terminazione.
`%…a` `%…e` `%…f` `%…g`	`double *`	Un numero a virgola mobile rappresentato in notazione decimale fissa o in notazione esponenziale: **[−]** ***iii . dddddd*** **[−]** ***i . dddddd*`e`±*xx*** **[−]** ***i . dddddd*`E`±*xx***
`%p`	`void *`	Interpreta il valore di un puntatore che sia rappresentato nello stesso modo in cui farebbe la funzione '**`printf("%p",` *puntatore*`)`**'.

Simbolo	Tipo di argomento	Conversione applicata
`%n`	`int *`	Questo specificatore non esegue alcuna conversione e si limita a memorizzare la quantità di caratteri (‘**`char`**’) letti fino a quel punto.
`%…[…]`	`char *`	Interpreta una stringa non vuota contenente solo i caratteri elencati tra parentesi quadre, aggiungendo alla fine il carattere nullo di terminazione. Se tra i caratteri si cerca anche la parentesi quadra chiusa, questa va messa all’inizio dell’elenco: ‘**`%…[]…]`**’.
`%…[^…]`	`char *`	Interpreta una stringa non vuota contenente solo caratteri diversi da quelli elencati tra parentesi quadre, aggiungendo alla fine il carattere nullo di terminazione. Se tra i caratteri da escludere si vuole indicare anche la parentesi quadra chiusa, questa va messa all’inizio dell’elenco: ‘**`%…[^]…]`**’.
`%%`		Interpreta un carattere di percentuale tra i dati in ingresso, ma senza memorizzare alcunché.

Tabella 69.268. Alcuni modificatori della lunghezza del dato in uscita.

Simbolo	Tipo	Simbolo	Tipo
`%…hhd` `%…hhi`	`signed char *`	`%…hhu` `%…hho` `%…hhx` `%…hhn`	`unsigned char *`
`%…hd` `%…hi`	`short int *`	`%…hu` `%…ho` `%…hx` `%…hn`	`unsigned short int *`
`%…ld` `%…li`	`long int *`	`%…lu` `%…lo` `%…lx` `%…ln`	`unsigned long int *`
		`%…lc` `%…ls` `%…lc` `%…l[…]`	`wchar_t *`

Simbolo	Tipo	Simbolo	Tipo
%…lld %…lli	long long int *	%…llu %…llo %…llx %…lln	unsigned long long int *
%…jd %…ji	intmax_t *	%…ju %…jo %…jx %…jn	uintmax_t *
%…zd %…zi	size_t *	%…zu %…zo %…zx %…zn	size_t *
%…td %…ti	ptrdiff_t *	%…tu %…to %…tx %…tn	ptrdiff_t *

Simbolo	Tipo	Simbolo	Tipo
`%...Le`			
`%...Lf`	`long double *`		
`%...Lg`			

A proposito dell'interpretazione di caratteri e di stringhe, va precisato cosa accade quando si usa il modificatore '**`l`**' (elle). Se nello specificatore di conversione appare un valore numerico che esprime un'ampiezza, questa indica una quantità di caratteri, in ingresso, da intendersi come byte. Utilizzando gli specificatori '**`%...lc`**' e '**`%...ls`**', la quantità di questi caratteri continua a riferirsi a byte, ma si interpretano le sequenze multibyte in ingresso per generare caratteri di tipo '**`wchar_t`**'.

Il documento che descrive lo standard del linguaggio afferma che la stringa di conversione è composta da direttive, ognuna delle quali è formata da: uno o più spazi (spazi veri e propri o caratteri di tabulazione orizzontale); un carattere multibyte diverso da '`%`' e diverso dai caratteri che rappresentano spazi, oppure uno specificatore di conversione.

[*spazi*] *carattere_multibyte* | `%...`

Dalla sequenza multibyte che costituisce i dati in ingresso da interpretare, vengono eliminati automaticamente gli spazi iniziali e finali (tutto ciò che si può considerare spazio, anche il codice di interruzione di riga), quando all'inizio o alla fine non ci sono corrispondenze con specificatori di conversione che possono interpretarli.

Quando la direttiva di interpretazione inizia con uno o più spazi orizzontali, significa che si vogliono ignorare gli spazi a partire dalla posizione corrente nella lettura dei dati in ingresso; inoltre, la presenza di un carattere che non fa parte di uno specificatore di conversione indica che quello stesso carattere deve essere incontrato nell'interpretazione dei dati in ingresso, altrimenti il procedimento di lettura e valutazione si deve interrompere. Se due specificatori di conversione appaiono adiacenti, i dati in ingresso corrispondenti possono essere separati da spazi orizzontali o da spazi verticali (il codice di interruzione di riga).

Purtroppo, la sintassi per la scrittura delle stringhe di conversione non è molto soddisfacente e diventa difficile spiegarne il comportamento, a meno di rimanere fermi su esempi molto semplici.

69.17.10.1 Funzioni che ricevono gli argomenti direttamente

Un gruppo di funzioni per l'interpretazione dell'input riceve direttamente gli argomenti variabili che servono agli specificatori di conversione:

```
int fscanf (FILE *restrict stream,
            const char *restrict format, ...);
int sscanf (const char *restrict s,
            const char *restrict format, ...);
int scanf  (const char *restrict format, ...);
```

Tutte le funzioni di questo gruppo hanno in comune la stringa di conversione, costituita dal parametro ***format***, e gli argomenti suc-

cessivi che sono puntatori di tipo indeterminato, in quanto per la loro interpretazione contano gli specificatori di conversione inseriti nella stringa. Inoltre, tutte queste funzioni restituiscono la quantità di valori assegnati alle variabili rispettive, oppure il valore corrispondente alla macro-variabile ***EOF*** nel caso si verifichi un errore prima di qualunque conversione.

La funzione ***sscanf()*** scandisce il contenuto della stringa indicata come primo parametro (***s***); la funzione ***fscanf()*** scandisce l'input proveniente dal flusso di file indicato come primo argomento (***stream***), mentre la funzione ***scanf()*** scandisce direttamente lo standard input.

69.17.10.2 Funzioni che ricevono gli argomenti da un'altra funzione

«

A fianco delle funzioni descritte nella sezione precedente, un gruppo analogo svolge le stesse operazioni, ma ricevendo gli argomenti variabili per riferimento. In pratica si tratta di ciò che serve quando gli argomenti variabili sono stati ottenuti da un'altra funzione e non da una chiamata diretta.

```
int vfscanf (FILE *restrict stream,
             const char *restrict format,
             va_list arg);
int vsscanf (const char *restrict s,
             const char *restrict format,
             va_list arg);
int vscanf  (const char *restrict format,
             va_list arg);
```

Il funzionamento è conforme a quello delle funzioni che non hanno la lettera '**v**' iniziale; per esempio, ***vscanf()*** si comporta conformemente a ***scanf()***. Per comprendere la differenza si potrebbe dimo-

strare la realizzazione ipotetica della funzione ***scanf()*** avvalendosi di ***vscanf()***:

```
int scanf (const char *restrict format, ...);
{
  va_list arg;
  va_start (arg, format);
  int count;
  count = vscanf (format, arg);
  va_end (arg);
  return count;
}
```

69.17.11 Funzioni per la lettura e la scrittura di un carattere alla volta

Le funzioni ***fgetc()*** e ***getc()*** leggono un carattere (*char*) attraverso il flusso di file indicato come argomento:

```
int fgetc (FILE *stream);

#define getc(STREAM) (fgetc (STREAM))
```

Lo standard prescrive che la funzione ***getc()*** sia in realtà una macroistruzione, così come si ipotizza nella dichiarazione appena mostrata. A questo proposito occorre tenere presente che, se si usa ***getc()***, l'espressione usata per individuare il flusso di file potrebbe essere valutata più di una volta.

Il carattere letto da ***fgetc()*** viene interpretato senza segno e trasformato in un intero (pertanto deve risultare essere di segno positivo). Se viene tentata la lettura oltre la fine del file, la funzione restituisce il valore rappresentato da ***EOF*** e memorizza questa condizione nella variabile strutturata che rappresenta il flusso di file. Se invece si

verifica un errore di lettura, viene impostato il contenuto dell'indicatore di errore relativo al flusso di file e la funzione restituisce sempre il valore ***EOF***.

Secondo lo standard, la funzione ***getchar()*** è equivalente a '**`getc (stdin)`**', senza specificare altro. Ciò può significare ragionevolmente che se ***getc()*** è una macroistruzione, anche ***getchar()*** dovrebbe esserlo, altrimenti potrebbe trattarsi di una funzione vera e propria:

```
#define getchar (getc (stdin))
```

La funzione ***ungetc()*** ha lo scopo di annullare l'effetto della lettura dell'ultimo carattere, ma il modo in cui viene gestita la cosa rende la questione molto delicata:

```
int ungetc (int c, FILE *stream);
```

Semplificando il problema, la funzione ***ungetc()*** rimanda indietro il carattere ***c*** nel flusso di file ***stream*** dal quale è appena stata eseguita una lettura. Tuttavia, non è garantito che il carattere in questione sia effettivamente quello che è stato letto per ultimo, ma la fase successiva di lettura deve fornire per primo tale carattere.

Si comprende intuitivamente che, se si eseguono operazioni di spostamento della posizione corrente relativa al flusso di file in questione, il carattere rimandato indietro con la funzione ***ungetc()*** debba essere dimenticato, soprattutto se questo non corrispondeva a quello che effettivamente era stato letto per ultimo in quel momento.

L'uso della funzione ***ungetc()*** implica un aggiornamento della posizione corrente relativa al flusso di file, ma questa modifica, in presenza di file di testo che non siano realizzati secondo lo standard tradizionale dei sistemi Unix, implica che l'entità di questa modifica

non possa essere predeterminabile.[11]

La funzione ***ungetc()*** può fallire nel suo intento e lo standard prescrive che sia «garantita» la possibilità di rimandare indietro almeno un carattere. Se la funzione riesce a eseguire l'operazione, restituisce il valore positivo corrispondente al carattere rinviato; altrimenti restituisce il valore della macro-variabile ***EOF***.

Le funzioni ***fputc()***, ***putc()*** e ***putchar()*** eseguono l'operazione inversa, rispettivamente, di ***fgetc()***, ***getc()*** e ***getchar()***; anche in questo caso vale il fatto che ***putc()*** possa essere realizzata come macroistruzione:

```
int fputc (int c, FILE *stream);

#define putc(CHAR, STREAM) (fputc ((CHAR), (STREAM)))

#define putchar(CHAR) (putc ((CHAR), stdout))
```

La funzione ***fputc()*** scrive un carattere (fornito come numero intero positivo) attraverso il flusso di file indicato; ***putc()*** fa lo stesso, ma potrebbe essere una macroistruzione; ***putchar()*** scrive attraverso lo standard output.

Se la scrittura fallisce, le funzioni (o le macroistruzioni) restituiscono il valore ***EOF***; diversamente restituiscono il valore positivo corrispondente al carattere scritto.

Per come sono state proposte queste funzioni, non c'è differenza nell'uso di ***getc()*** al posto di ***fgetc()***, così come tra ***putc()*** e ***fputc()***. Evidentemente, se la propria libreria può esprimere le macroistruzioni ***getc()*** e ***putc()*** richiamando funzioni del sistema operativo (funzioni che dovrebbero essere richiamate anche da ***fgetc()*** e ***fputc()***), si può risparmiare un livello di chiamate per accelerare leggermente l'esecuzione del programma.

69.17.12 Funzioni per l'input e l'output di file di testo

Le funzioni ***fgets()*** e ***fputs()*** sono utili per l'accesso a file di testo, quando si vuole indicare il flusso di file:

```
char *fgets (char *restrict s, int n,
             FILE *restrict stream);

int   fputs (const char *restrict s, FILE *restrict stream);
```

La funzione ***fgets()*** legge al massimo $n-1$ caratteri (nel senso di elementi '**char**') attraverso il flusso di file ***stream***, copiandoli in memoria a partire dall'indirizzo ***s*** e aggiungendo alla fine il carattere nullo di terminazione delle stringhe. La lettura si esaurisce prima di $n-1$ caratteri se viene incontrato il codice di interruzione di riga, il quale viene rappresentato nella stringa a cui punta ***s***, ovvero se si raggiunge la fine del file. In ogni caso, la stringa ***s*** viene terminata correttamente con il carattere nullo.

La funzione ***fgets()*** restituisce la stringa ***s*** se la lettura avviene con successo, ovvero se ha prodotto almeno un carattere; altrimenti, il contenuto dell'array a cui punta ***s*** non viene modificato e la funzione restituisce il puntatore nullo. Se si creano errori imprevisti, la

funzione potrebbe restituire il puntatore nullo, ma senza garantire che l'array ***s*** sia rimasto intatto.

La funzione ***fputs()*** serve a copiare la stringa a cui punta ***s*** nel file rappresentato dal flusso di file ***stream***. La copia della stringa avviene escludendo però il carattere nullo di terminazione. Va osservato che questa funzione, pur essendo contrapposta evidentemente a ***fgets()***, **non conclude la riga** del file, ovvero, non aggiunge il codice di interruzione di riga. Per ottenere la conclusione della riga di un file di testo, occorre inserire nella stringa, espressamente, il carattere '**`\n`**'.

La funzione ***fputs()*** restituisce il valore rappresentato da ***EOF*** se l'operazione di scrittura produce un errore; altrimenti restituisce un valore positivo qualunque.

Le funzioni ***gets()*** e ***puts()*** sono utili per l'accesso a file di testo, quando si vogliono utilizzare i flussi standard. In linea di massima, assomigliano a ***fgets()*** e ***fputs()***, ma il funzionamento non è perfettamente conforme a quelle:

```
char *gets (char *s);

int   puts (const char *s);
```

Il funzionamento di ***gets()*** è perfettamente conforme a quello di ***fgets()***, con la sola differenza che il flusso di file da cui si leggono i caratteri è lo standard input. Nel caso di ***puts()***, a parte il fatto che si usa lo standard output per la scrittura, occorre sottolineare che alla fine della stringa **viene accodata la scrittura del codice di interruzione di riga**.

69.17.13 Funzioni per l'input e output diretto

«

Le funzioni ***fread()*** e ***fwrite()*** consentono di leggere e scrivere attraverso un flusso di file aperto, il quale deve essere specificato espressamente tra gli argomenti. Lo standard prescrive che queste funzioni si avvalgano rispettivamente di ***fgetc()*** e di ***fputc()***.

```
size_t fread  (void *restrict ptr,
               size_t size,
               size_t nmemb,
               FILE *restrict stream);

size_t fwrite (const void *restrict ptr,
               size_t size,
               size_t nmemb,
               FILE *restrict stream);
```

Le due funzioni (***fread()*** e ***fwrite()***) hanno praticamente gli stessi argomenti, usati in modo analogo. La lettura e la scrittura avviene a blocchi da ***size*** byte, ripetuta per ***nmemb*** volte, attraverso il flusso di file specificato come ***stream***. La lettura implica la memorizzazione dei caratteri in forma di elementi '**`unsigned char`**', a partire dall'indirizzo indicato dal puntatore ***ptr***; la scrittura copia nello stesso modo i caratteri a partire dal puntatore ***ptr***, verso il flusso di file.

L'aggiornamento della posizione corrente interna al file a cui si riferisce il flusso avviene esattamente come per le funzioni ***fgetc()*** e ***fputc()***.

Il valore restituito dalle funzioni ***fread()*** e ***fwrite()*** rappresenta la quantità di blocchi, ovvero la quantità di elementi ***nmemb*** che sono stati copiati con successo. Pertanto, se si ottiene un valore inferiore

a ***nmemb***, significa che l'operazione è stata interrotta a causa di un errore.

69.17.14 Funzioni per il posizionamento

Sono previste diverse funzioni per modificare la posizione corrente dei flussi di file. Le funzioni più semplici per iniziare sono ***fseek()***, ***ftell()*** e ***rewind()***:

```
int      fseek  (FILE *stream, long int offset, int whence);
long int ftell  (FILE *stream);
void     rewind (FILE *stream);
```

Lo standard POSIX prevede anche le funzioni ***fseeko()*** e ***ftello()***, equivalenti alle funzioni ***fseek()*** e ***ftell()*** dello standard C, con la differenza che lo scostamento, fornito come argomento o restituito dalla funzione, è di tipo '**`off_t`**', al posto di essere di tipo '**`long int`**':

```
int      fseeko (FILE *stream, off_t offset, int whence);
off_t    ftello (FILE *stream);
```

In generale, le funzioni ***fseek()*** e ***fseeko()*** spostano la posizione corrente relativa al flusso di file ***stream***, nella nuova posizione determinata dai parametri ***whence*** e ***offset***. Il parametro ***whence*** viene fornito attraverso una macro-variabile che può essere ***SEEK_SET***, ***SEEK_CUR*** o ***SEEK_END***, indicando rispettivamente l'inizio del file, la posizione corrente o la fine del file. Dalla posizione indicata dal parametro ***whence*** viene aggiunta, algebricamente, la quantità di byte indicata dal parametro ***offset***.

Quanto descritto a proposito del posizionamento con le funzioni ***fseek()*** e ***fseeko()*** riguarda i file che vengono gestiti in modo binario, perché con i file di testo è opportuno avere maggiore accortezza:

il valore del parametro ***offset*** deve essere zero, oppure quanto restituito in precedenza dalle funzioni ***ftell()*** o ***ftello()*** per lo stesso flusso di file, ma in tal caso, ovviamente, il parametro ***whence*** deve corrispondere a ***SEEK_SET***.

Le funzioni ***fseek()*** e ***fseeko()*** restituiscono zero se possono eseguire l'operazione, altrimenti danno un risultato diverso; nel caso di ***fseeko()***, quando si presenta un errore, il risultato restituito è precisamente −1, e in più viene aggiornata anche la variabile ***errno***.

Le funzioni ***ftell()*** e ***ftello()*** restituiscono la posizione corrente del flusso di file indicato come argomento. Questo valore può essere usato con ***fseek()*** o ***fseeko()*** rispettivamente, al posto dello scostamento (il parametro ***offset***), indicando come posizione di riferimento l'inizio del file, ovvero ***SEEK_SET***. Se le funzioni ***ftell()*** e ***ftello()*** non riescono a fornire la posizione, restituiscono il valore −1 (tradotto rispettivamente in '**`long int`**' e '**`off_t`**') e annotano il fatto nella variabile ***errno***.

La funzione ***rewind()*** si limita a riposizionare il flusso di file all'inizio. In pratica è come utilizzare la funzione ***fseek()*** specificando uno scostamento pari a zero a partire da ***SEEK_SET***, ignorando il valore restituito:

```
(void) fseek (stream, 0L, SEEK_SET)
```

Va osservato che il riposizionamento di un flusso di file implica l'azzeramento dell'indicatore di fine file, se questo risulta impostato, e la cancellazione dei caratteri che eventualmente fossero stati rimandati indietro con la funzione ***ungetc()***.

Le funzioni ***fseek()*** e ***ftell()*** sono utili particolarmente per i file bi-

nari ed eventualmente per i file di testo con una rappresentazione dei caratteri tradizionale. Ma quando il file di testo contiene anche caratteri espressi attraverso sequenze multibyte, il posizionamento al suo interno dovrebbe tenere anche conto del progresso nell'interpretazione di queste sequenze. Pertanto, esistono altre due funzioni per leggere la posizione e ripristinarla in un secondo momento:

```
int fgetpos (FILE *restrict stream, fpos_t *restrict pos);
int fsetpos (FILE *stream, const fpos_t *pos);
```

Entrambe le funzioni che appaiono nei due prototipi restituiscono zero se l'operazione è stata compiuta con successo, altrimenti restituiscono un valore differente. Nel caso particolare di ***fsetpos()***, se si verifica un errore, questo viene annotato nella variabile ***errno***.

Le due funzioni richiedono come primo argomento il flusso di file a cui ci si riferisce; come secondo argomento richiedono il puntatore a una variabile di tipo '**`fpos_t`**'. La funzione ***fgetpos()*** memorizza nella variabile a cui punta il parametro ***pos*** le informazioni sulla posizione corrente del file, assieme allo stato di interpretazione relativo alle sequenze multibyte; la funzione ***fsetpos()***, per converso, utilizza la variabile a cui punta ***pos*** per ripristinare la posizione memorizzata, assieme allo stato di avanzamento dell'interpretazione di una sequenza multibyte.

69.17.15 Accesso esclusivo ai flussi di file

« POSIX

Lo standard POSIX, introducendo la gestione dei thread multipli, inserisce nel file '`stdio.h`' alcune funzioni per controllare l'accesso esclusivo ai flussi di file. Va tenuto a mente che si tratta di un controllo che riguarda esclusivamente il processo elaborativo in corso, e non l'accesso ai file da parte di processi differenti.

```
void flockfile   (FILE *stream);
int  trylockfile (FILE *stream);
void funlockfile (FILE *stream);
```

La funzione ***flockfile()*** cerca di ottenere un accesso esclusivo a un file, individuato da un puntatore che rappresenta il flusso di file relativo. Se il file risulta già impegnato, rimane in attesa, fino a quando si libera. La funzione ***trylockfile()***, invece, non rimane in attesa e restituisce l'esito della sua azione: zero se è riuscita a ottenere l'accesso esclusivo, un numero diverso se invece non c'è riuscita. Quando poi un thread che aveva ottenuto l'accesso esclusivo a un flusso di file, non ne ha più bisogno, lo libera con la funzione ***funlockfile()***.

Le funzioni ***getc()***, ***putc()***, ***getchar()*** e ***putchar()***, quando sono presenti le estensioni per la gestione dei thread multipli, e di conseguenza anche per l'accesso esclusivo ai flussi di file, si comportano rispettando tali vincoli. Eventualmente, se per qualche ragione si vogliono usare queste funzionalità, ignorando espressamente tali vincoli, sono disponibili funzioni equivalenti, il cui nome termina per '**`_unlocked`**':

```
int getc_unlocked (FILE *stream);
int putc_unlocked (int c, FILE *stream);
int getchar_unlocked (void);
int putchar_unlocked (int c);
```

Lo standard POSIX prescrive comunque che queste siano usate solo da thread che hanno già ottenuto un accesso esclusivo al flusso relativo, in modo da non compromettere la gestione controllata di tali accessi. Pertanto, in tal modo queste funzioni consentono semplicemente un'esecuzione più rapida, dal momento che non vengono eseguiti tutti i controlli necessari.

69.17.16 Condotti

«

Lo standard POSIX introduce il concetto di «condotto», ovvero di *pipe*, attraverso il quale è possibile inviare lo standard output di un processo elaborativo, verso lo standard input di un altro. Per attuare questo meccanismo, è necessario che un processo sia in grado di avviare un altro processo, attraverso un comando da dare alla shell, e da questo processo viene poi letto lo standard output o scritto lo standard input.

Dal punto di vista del processo che crea il condotto, il processo secondario avviato è trattato come se fosse un file, dove però si può solo leggere o scrivere, ma non si possono fare entrambe le cose.

```
FILE *popen  (const char *command, const char *type);
int   pclose (FILE *stream);
```

Il condotto viene aperto con la funzione ***popen()***, la quale assomiglia a ***fopen()***, ma invece dell'indicazione del percorso del file da aprire, richiede il comando da eseguire attraverso la shell '`/bin/sh`' (nota come la shell POSIX). Per quanto riguarda il tipo di accesso, va osservato che può trattarsi soltanto di lettura o scrittura, pertanto si può scegliere solo tra la stringa '`"r"`' o '`"w"`'.

La lettura o la scrittura in un flusso di file associato a un condotto avviene nel modo consueto, ma la chiusura richiede l'uso della funzione ***pclose()***.

69.17.17 Informazioni sul terminale

«

Lo standard POSIX prevede che il sistema operativo abbia una gestione dei file di dispositivo per rappresentare i vari componenti fisici dell'elaboratore. Nel file '`stdio.h`' inserisce la funzione ***ctermid()***,

con lo scopo di conoscere il percorso del file di dispositivo del terminale associato al processo elaborativo.

POSIX

```
char *ctermid (char *s);
```

La funzione si aspetta di ricevere come argomento il puntatore a una stringa modificabile, in cui scrivere il percorso del terminale. Se però viene fornito un puntatore nullo, l'informazione viene annotata in un'area di memoria che può essere statica (e quindi riutilizzata a ogni chiamata della funzione). Il percorso del terminale, annotato dalla funzione, può utilizzare al massimo la quantità di caratteri definita dalla macro-variabile ***L_ctermid***; pertanto, se si definisce un array di caratteri da usare per tale annotazione, deve essere almeno quella dimensione.

La funzione restituisce sempre il puntatore a una stringa, che può essere nulla se non è possibile determinare il terminale. Pertanto la funzione non prevede l'indicazione di errori.

69.17.18 Gestione degli errori

«

Un gruppo di funzioni di 'stdio.h' consente di verificare ed eventualmente azzerare lo stato degli indicatori di errore riferiti a un certo flusso di file:

```
void clearerr (FILE *stream);
int  feof     (FILE *stream);
int  ferror   (FILE *stream);
void perror   (const char *s);
```

La funzione ***clearerr()*** azzera gli indicatori di errore e di fine file per il flusso di file indicato come argomento, senza restituire alcunché.

La funzione ***feof()*** controlla lo stato dell'indicatore di fine file per il flusso di file indicato. Se questo non è attivo restituisce zero,

altrimenti restituisce un valore diverso da zero.

La funzione ***ferror()*** controlla lo stato dell'indicatore di errore per il flusso di file indicato. Se questo non è attivo restituisce zero, altrimenti restituisce un valore diverso da zero.

La funzione ***perror()*** prende in considerazione la variabile ***errno*** e cerca di tradurla in un messaggio testuale da emettere attraverso lo standard error (con tanto di terminazione della riga, in modo da riposizionare a capo il cursore). Se il parametro ***s*** corrisponde a una stringa non vuota, il testo di questa viene posto anteriormente al messaggio, separandolo con due punti e uno spazio (‘: ’). Il contenuto del messaggio è lo stesso che si otterrebbe con la funzione ***strerror()***, fornendo come argomento la variabile ***errno***.

69.17.19 Realizzazione di «vsnprintf()» e altre collegate

Le funzioni per la composizione dell'output che possono essere realizzate senza avere definito la gestione dei file, sono quelle che si limitano a produrre una stringa. La funzione che va realizzata per prima è ***vsnprintf()***, in quanto ***snprintf()*** si può limitare a richiamarla. Naturalmente, anche ***vsprintf()*** e ***sprintf()*** possono avvalersi della stessa ***vsnprintf()***, ponendo un limite massimo abbastanza grande alla stringa da generare. Nella sezione 95.18.42 è disponibile un esempio di realizzazione parziale di ***vsnprintf()***. L'esempio seguente mostra come si ottiene ***snprintf()***, una volta che è disponibile ***vsnprintf()***:

```
#include <stdio.h>
int
snprintf (char *restrict string, size_t n,
          const char *restrict format, ...)
{
```

```
  va_list ap;
  va_start (ap, format);
  return vsnprintf (string, n, format, ap);
}
```

Eventualmente, per realizzare le funzioni ***vsprintf()*** e ***sprintf()***, secondo le limitazioni già descritte, sono sufficienti due macroistruzioni:

```
#define vsprintf(s, format, arg) \
                 (vsnprintf (s, SIZE_MAX, format, arg))
#define sprintf(s, ...)          \
                 (snprintf (s, SIZE_MAX, __VA_ARGS__))
```

69.18 Riferimenti

«

- Wikipedia, *C standard library*, *http://en.wikipedia.org/wiki/C_standard_library*
- *ISO/IEC 9899:TC2*, *http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf*
- Wikipedia, *assert.h*, *limits.h*, *stdint.h*, *errno.h*, *locale.h*, *ctype.h*, *stdarg.h*, *stdlib.h*, *inttypes.h*, *iso646.h*, *stdbool.h*, *stddef.h*, *string.h*, *signal.h*, *time.h*, *stdio.h*

  http://en.wikipedia.org/wiki/Assert.h , *http://en.wikipedia.org/wiki/Limits.h* , *http://en.wikipedia.org/wiki/Stdint.h* , *http://en.wikipedia.org/wiki/Errno.h* , *http://en.wikipedia.org/wiki/Locale.h* , *http://en.wikipedia.org/wiki/Ctype.h* , *http://en.wikipedia.org/wiki/Stdarg.h* , *http://en.wikipedia.org/wiki/Stdlib.h* , *http://en.wikipedia.org/wiki/Inttypes.h* , *http://en.wikipedia.org/wiki/Iso646.h* , *http://en.wikipedia.org/wiki/*

Stdbool.h , *http://en.wikipedia.org/wiki/Stddef.h* , *http://en.wikipedia.org/wiki/String.h* , *http://en.wikipedia.org/wiki/Signal.h* , *http://en.wikipedia.org/wiki/Time.h* , *http://en.wikipedia.org/wiki/Stdio.h*

- The Open Group, *The Single UNIX® Specification, Version 2*, *assert.h*, *limits.h*, *stdint.h*, *errno.h*, *locale.h*, *ctype.h*, *stdarg.h*, *stdlib.h*, *inttypes.h*, *iso646.h*, *stdbool.h*, *stddef.h*, *string.h*, *signal.h*, *time.h*, *stdio.h*

  http://pubs.opengroup.org/onlinepubs/000095399/basedefs/assert.h.html ,
  http://pubs.opengroup.org/onlinepubs/000095399/basedefs/limits.h.html ,
  http://pubs.opengroup.org/onlinepubs/000095399/basedefs/stdint.h.html ,
  http://pubs.opengroup.org/onlinepubs/000095399/basedefs/errno.h.html ,
  http://pubs.opengroup.org/onlinepubs/000095399/basedefs/locale.h.html ,
  http://pubs.opengroup.org/onlinepubs/000095399/basedefs/ctype.h.html ,
  http://pubs.opengroup.org/onlinepubs/000095399/basedefs/stdarg.h.html ,
  http://pubs.opengroup.org/onlinepubs/000095399/basedefs/stdlib.h.html ,
  http://pubs.opengroup.org/onlinepubs/000095399/basedefs/inttypes.h.html ,
  http://pubs.opengroup.org/onlinepubs/000095399/basedefs/iso646.h.html ,

http://pubs.opengroup.org/onlinepubs/000095399/basedefs/stdbool.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/stddef.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/string.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/signal.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/time.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/stdio.h.html

- Steven Pemberton, *Enquire: Everything you wanted to know about your C Compiler and Machine, but didn't know who to ask*, *http://homepages.cwi.nl/~steven/enquire.html*

[1] Il carattere viene convertito da '**`unsigned char`**' a '**`int`**'.

[2] Si tratta di un puntatore di puntatore, solo perché si deve poter alterare ciò a cui punta, ma questo tipo di valore è, a sua volta, un puntatore.

[3] Il tipo '**`size_t`**', restituito dalle funzioni, è un intero senza segno; pertanto, in condizioni normali, ovvero con una rappresentazione dei valori negativi con il complemento a due, la conversione di -1 si traduce nel valore massimo rappresentabile.

[4] Lo standard POSIX non estende il contenuto del file '`inttypes.h`'.

[5] Lo standard POSIX non estende il file '`iso646.h`'.

[6] Lo standard POSIX non estende il file ‘`stdbool.h`’

[7] Lo standard POSIX non estende il file ‘`stddef.h`’.

[8] Il valore positivo massimo è $(2^{31})-1$, il quale, diviso per la quantità di secondi di un giorno (86400) dà 24855 che, diviso 365, dà circa 68 anni.

[9] Il caso della funzione ***clock()*** e del tipo ‘**`clock_t`**’ è stato considerato a parte.

[10] Si tratta di byte: se il testo copiato è costituito da sequenze multi-byte, i byte sono in quantità maggiore rispetto ai caratteri tipografici che si ottengono.

[11] L’arretramento di un carattere nella posizione corrente di un file di testo non è detto corrisponda alla sottrazione di una unità, perché bisogna tenere in considerazione il modo in cui un file di testo è strutturato nel proprio sistema operativo.

70.1 File «sys/types.h» 1312
70.2 File «sys/stat.h» 1315
70.2.1 Macro-variabili per la definizione del contenuto di un valore di tipo «mode_t» 1316
70.2.2 Struttura «stat» 1319
70.2.3 Prototipi di funzione 1320
70.3 File «strings.h» 1321
70.3.1 Funzione «ffs()» 1321
70.3.2 Funzioni «strcasecmp()» e «strncasecmp()» . . . 1322
70.4 File «fcntl.h» 1322
70.4.1 Tipi di dati derivati 1323
70.4.2 Opzioni per la funzione «fcntl()» 1323
70.4.3 Gestione del blocco dei file 1326
70.4.4 Funzioni 1327
70.5 File «unistd.h» 1329
70.5.1 Denominazione dei descrittori di file 1330
70.5.2 Verifica dei permessi di accesso 1330
70.5.3 Funzioni 1331

70.5.4 Funzioni di servizio per la gestione di file e directory 1333

70.6 File «dirent.h» 1342

70.6.1 Struttura «dirent» 1343
70.6.2 Tipo «DIR» 1343
70.6.3 Prototipi di funzioni 1344

70.7 File «termios.h» 1345

70.7.1 Tipi speciali 1345
70.7.2 Tipo «struct termios» 1346
70.7.3 Codici di controllo 1347
70.7.4 Indicatori per il membro «c_iflag» . 1347
70.7.5 Indicatori per il membro «c_oflag» 1348
70.7.6 Indicatori per il membro «c_cflag» 1348
70.7.7 Indicatori per il membro «c_lflag» . . . 1349
70.7.8 Funzioni 1349

70.8 Riferimenti 1350

access() 1331 1333 alarm() 1331 blkcnt_t 1312 blksize_t 1312 BRKINT 1347 cc_t 1345 chdir() 1331 1338 chmod() 1320 chown() 1331 clock_t 1312 close() 1331 closedir() 1344 confstr() 1331 creat() 1327 dev_t 1312 DIR 1343 dirent.h 1342 dup() 1331 dup2() 1331 ECHO 1349 ECHOE 1349 ECHOK 1349 ECHONL 1349 execl() 1331 execle() 1331 execlp() 1331 execv() 1331 execve() 1331 execvp() 1331 fchmod() 1320 fchown() 1331 fcntl() 1327 fcntl.h 1322 FD_CLOEXEC

1323 ffs() 1321 fork() 1331 fpathconf() 1331 fstat() 1320 ftruncate() 1331 F_DUPFD 1323 F_GETFD 1323 F_GETFL 1323 F_GETLK 1323 F_GETOWN 1323 F_OK 1330 F_RDLCK 1326 F_SETFD 1323 F_SETFL 1323 F_SETLK 1323 F_SETLKW 1323 F_SETOWN 1323 F_UNLCK 1326 F_WRLCK 1326 getcwd() 1331 1337 getegid() 1331 geteuid() 1331 getgid() 1331 getgroups() 1331 gethostname() 1331 getlogin() 1331 getlogin_r() 1331 getopt() 1331 getpgrp() 1331 getpid() 1331 getppid() 1331 getuid() 1331 gid_t 1312 ICANON 1349 ICRNL 1347 id_t 1312 IEXTEN 1349 IGNBRK 1347 IGNCR 1347 IGNPAR 1347 INLCR 1347 ino_t 1312 INPCK 1347 isatty() 1331 ISIG 1349 ISTRIP 1347 IXOFF 1347 IXON 1347 link() 1331 1340 lseek() 1331 lstat() 1320 mkdir() 1320 mkfifo() 1320 mknod() 1320 mode_t 1312 1316 NCCS 1346 nlink_t 1312 NOFLSH 1349 off_t 1312 open() 1327 opendir() 1344 OPOST 1348 O_ACCMODE 1323 O_APPEND 1323 O_CREAT 1323 O_DSYNC 1323 O_EXCL 1323 O_NOCTTY 1323 O_NONBLOCK 1323 O_RDONLY 1323 O_RDWR 1323 O_RSYNC 1323 O_SYNC 1323 O_TRUNC 1323 O_WRONLY 1323 PARMRK 1347 pathconf() 1331 pause() 1331 pid_t 1312 pipe() 1331 pthread_t 1312 read() 1331 readdir() 1344 readlink() 1331 rewinddir() 1344 rmdir() 1331 R_OK 1330 setegid() 1331 seteuid() 1331 setgid() 1331 setpgid() 1331 setsid() 1331 setuid() 1331 size_t 1312 sleep() 1331 speed_t 1345 ssize_t 1312 stat() 1320 stat.h 1315 STDERR_FILENO 1330 STDIN_FILENO 1330 STDOUT_FILENO 1330 strcasecmp() 1322 strings.h 1321 strncasecmp() 1322 structure stat

1319 struct dirent 1343 struct termios 1346 st_atime 1319 st_blksize 1319 st_blocks 1319 st_ctime 1319 st_dev 1319 st_gid 1319 st_ino 1319 st_mode 1319 st_mtime 1319 st_nlink 1319 st_rdev 1319 st_size 1319 st_uid 1319 symlink() 1331 sys/types.h 1312 sysconf() 1331 S_IFBLK 1316 S_IFCHR 1316 S_IFDIR 1316 S_IFIFO 1316 S_IFLNK 1316 S_IFMT 1316 S_IFREG 1316 S_IFSOCK 1316 S_IRGRP 1316 S_IROTH 1316 S_IRUSR 1316 S_IRWXG 1316 S_IRWXO 1316 S_IRWXU 1316 S_ISBLK() 1316 S_ISCHR() 1316 S_ISDIR() 1316 S_ISFIFO() 1316 S_ISGID 1316 S_ISLNK() 1316 S_ISREG() 1316 S_ISSOCK() 1316 S_ISUID 1316 S_ISVTX 1316 S_IWGRP 1316 S_IWOTH 1316 S_IWUSR 1316 S_IXGRP 1316 S_IXOTH 1316 S_IXUSR 1316 tcflag_t 1345 tcgetattr() 1349 tcgetpgrp() 1331 TCSADRAIN 1349 TCSAFLUSH 1349 TCSANOW 1349 tcsetattr() 1349 tcsetpgrp() 1331 termios.h 1345 time_t 1312 TOSTOP 1349 ttyname() 1331 ttyname_r() 1331 uid_t 1312 umask() 1320 unistd.h 1329 unlink() 1331 1340 VEOF 1347 VEOL 1347 VERASE 1347 VINTR 1347 VKILL 1347 VMIN 1347 VQUIT 1347 VSTART 1347 VSTOP 1347 VSUSP 1347 VTIME 1347 write() 1331 W_OK 1330 X_OK 1330 _exit() 1331

In generale, la libreria offerta da un compilatore del linguaggio C si può estendere in modo imprecisato verso le definizioni dello standard POSIX. Per esempio, è normale che una libreria C includa le funzionalità relative alla gestione delle espressioni regolari, definite dallo standard POSIX. Pertanto, non esiste propriamente una libreria

C e una POSIX, va quindi verificato con il proprio compilatore cosa offrono effettivamente le librerie disponibili, specificando eventualmente, in fase di compilazione, l'inclusione di questa o quella libreria precompilata per la gestione di quella certa funzionalità POSIX.

Nei capitoli successivi vengono descritti alcuni dei file di intestazione previsti dallo standard POSIX, che a loro volta non sono già presi in considerazione dallo standard del linguaggio C. In certi casi viene mostrato come potrebbero essere realizzati questi file (gli esempi dovrebbero essere disponibili a partire da *allegati/c/*).

Tabella 70.1. Alcuni file di intestazione dello standard POSIX che non si trovano già nello standard del linguaggio C.

Intestazione	Descrizione	Riferimenti
`sys/types.h`	Tipi di dati derivati.	sezione 70.1
`sys/stat.h`	Definizione dei dati restituiti dalla funzione ***stat()***, necessari alla qualificazione delle caratteristiche dei file, e di alcune funzioni relative alla questione.	sezione 70.2
`strings.h`	Funzioni per il trattamento delle stringhe e simili che non sono già incluse in '`string.h`'.	sezione 70.3
`fcntl.h`	Opzioni per il controllo dei file, gestiti in qualità di descrittori.	sezione 70.4
`unistd.h`	Macro-variabili standard e molte funzioni sulla gestione dei file.	sezione 70.5
`dirent.h`	Gestione delle directory, in qualità di flussi, attraverso puntatori di tipo '**`DIR *`**'.	sezione 70.6

Intestazione	Descrizione	Riferimenti
`termios.h`	Configurazione dei dispositivi di terminale.	sezione 70.7

70.1 File «sys/types.h»

«

Il file ‘`sys/types.h`’ viene usato dallo standard POSIX per definire tutti i tipi di dati derivati, inclusi alcuni che già fanno parte dello standard C puro e semplice (si veda eventualmente la realizzazione di questo file nei sorgenti di os32, sezione 95.26). La tabella successiva ne descrive solo una parte.

Tipo derivato	Descrizione
`blkcnt_t`	Numero intero con segno. Quantità di blocchi, riferita ai file.
`blksize_t`	Numero intero con segno. Dimensione in blocchi di un file.
`clock_t`	Numero intero. Unità di tempo che rappresenta un ciclo virtuale di CPU. È già definito nel file ‘`time.h`’.
`dev_t`	Numero intero. Numero identificativo di un dispositivo.
`gid_t`	Numero intero senza segno. Numero identificativo del gruppo a cui appartiene un file.
`ino_t`	Numero intero senza segno. Numero di inode, ovvero il numero identificativo di un file.
`mode_t`	Numero intero. Attributo di accesso di un file.
`nlink_t`	Numero intero senza segno. Quantità di collegamenti fisici riferiti a un file.

Tipo derivato	Descrizione
`off_t`	Numero intero con segno. Scostamento relativo al contenuto di un file.
`pid_t`	Numero intero con segno. Numero che identifica un processo elaborativo. Il segno è necessario perché il valore −1 rappresenta un errore nella creazione di un processo.
`pthread_t`	Numero intero. Numero identificativo di un thread.
`size_t`	Numero intero senza segno. Usato per esprimere la grandezza di oggetti rappresentati in memoria (variabili, distanza tra puntatori e simili). Nello standard C, questo tipo derivato viene definito nel file '`stddef.h`'.
`ssize_t`	Numero intero con segno (*signed size_t*). Usato per esprimere una quantità di byte, se positivo, oppure un errore se negativo.
`time_t`	Numero intero con segno. Usato per esprimere un tempo in secondi, trascorso a partire dal giorno 1 gennaio 1970. Nello standard C, questo tipo derivato viene definito nel file '`time.h`'.
`uid_t`	Numero intero senza segno. Numero identificativo dell'utente a cui appartiene un file.

Tipo derivato	Descrizione
`id_t`	Numero intero senza segno. Numero identificativo generico, in grado di ospitare il valore dei tipi '**`pid_t`**', '**`uid_t`**' e '**`gid_t`**'. Va tenuto in considerazione il fatto che un numero di identificazione di un processo deve essere positivo, salvo il caso in cui un valore negativo serva per indicare un errore; pertanto, il tipo '**`id_t`**' può essere definito come privo di segno.

L'esempio successivo mostra come potrebbero essere dichiarati questi tipi derivati, limitatamente ai casi descritti nella tabella:

```
typedef long int           blkcnt_t;
typedef long int           blksize_t;
typedef long int           clock_t;
typedef unsigned long int  dev_t;
typedef unsigned int       id_t;
typedef unsigned long int  ino_t;
typedef unsigned int       gid_t;
typedef unsigned int       mode_t;
typedef unsigned int       nlink_t;
typedef long long int      off_t;
typedef int                pid_t;
typedef unsigned long int  pthread_t;
typedef unsigned long int  size_t;
typedef long int           ssize_t;
typedef long int           time_t;
typedef unsigned int       uid_t;
```

70.2 File «sys/stat.h»

Il file ‘`sys/stat.h`’ viene usato dallo standard POSIX principalmente per definire un insieme di macro-variabili che individuano le caratteristiche fondamentali di un file (tipo di file e permessi), per definire il tipo ‘**`struct stat`**’ che serve a rappresentare lo stato di un file, per dichiarare il prototipo di alcune funzioni che hanno a che fare con queste informazioni (si veda eventualmente la realizzazione del file ‘`sys/stat.h`’ e di alcune delle sue funzioni nei sorgenti di os32, sezione 95.25).

È importante considerare il file ‘`sys/stat.h`’ assieme a ‘`fcntl.h`’.

Nel file ‘`sys/stat.h`’ si fa riferimento a un insieme di tipi derivati, dichiarati nel file ‘`sys/types.h`’. Per semplicità, l’esempio propone la sua inclusione iniziale:

```
#include <sys/types.h>  // dev_t
                        // off_t
                        // blkcnt_t
                        // blksize_t
                        // ino_t
                        // mode_t
                        // nlink_t
                        // uid_t
                        // gid_t
                        // time_t
```

70.2.1 Macro-variabili per la definizione del contenuto di un valore di tipo «mode_t»

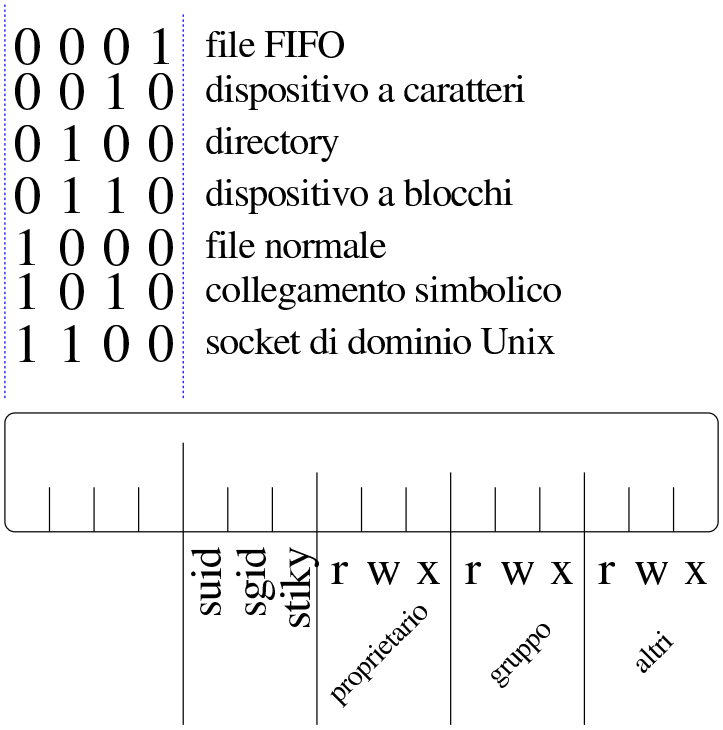
«

Il tipo derivato '**mode_t**' serve a rappresentare il tipo di un file (o di una directory) e i permessi disponibili. Questo tipo si traduce generalmente in un intero a 16 bit. Trattandosi di un valore numerico, queste informazioni sono distinte a gruppi di bit, selezionabili attraverso una maschera. Pertanto, tra le macro-variabili che distinguono le varie caratteristiche associabili a una variabile di tipo '**mode_t**', alcune vanno usate come maschera, per distinguere un certo insieme di informazioni, dalle altre.

```
//
// Tipo di file.
//
#define S_IFMT   0170000  // Maschera del tipo di file.
//
#define S_IFBLK  0060000  // File di dispositivo a blocchi.
#define S_IFCHR  0020000  // File di dispositivo a caratteri.
#define S_IFIFO  0010000  // File FIFO.
#define S_IFREG  0100000  // File puro e semplice
                          // (regular file).
#define S_IFDIR  0040000  // Directory.
#define S_IFLNK  0120000  // Collegamento simbolico.
#define S_IFSOCK 0140000  // Socket di dominio Unix.
//
// Permessi di accesso dell'utente proprietario del file.
//
#define S_IRWXU  0000700  // Maschera che rappresenta
                          // simultaneamente tutti i
                          // permessi per l'utente
                          // proprietario.
#define S_IRUSR  0000400  // Permesso di lettura.
#define S_IWUSR  0000200  // Permesso di scrittura.
```

```
#define S_IXUSR  0000100  // Permesso di esecuzione
                          // o attraversamento.
//
// Permessi di accesso del gruppo proprietario del file.
//
#define S_IRWXG  0000070  // Maschera che rappresenta
                          // simultaneamente tutti i
                          // permessi per il gruppo
                          // proprietario.
#define S_IRGRP  0000040  // Permesso di lettura.
#define S_IWGRP  0000020  // Permesso di scrittura.
#define S_IXGRP  0000010  // Permesso di esecuzione
                          // o attraversamento.
//
// Permessi di accesso degli altri utenti.
//
#define S_IRWXO  0000007  // Maschera che rappresenta
                          // simultaneamente tutti i
                          // permessi per il gruppo
                          // proprietario.
#define S_IROTH  0000004  // Permesso di lettura.
#define S_IWOTH  0000002  // Permesso di scrittura.
#define S_IXOTH  0000001  // Permesso di esecuzione
                          // o attraversamento.
//
// Permessi aggiuntivi: S-bit.
// In questo caso non c'è una maschera che li includa tutti.
//
#define S_ISUID  0004000  // S-UID.
#define S_ISGID  0002000  // S-GID.
#define S_ISVTX  0001000  // Sticky.
```

Figura 70.6. Schema delle «modalità» POSIX per descrivere le caratteristiche di un file. Questa informazione si associa all'inode.



Inoltre, vengono definite delle macroistruzioni per distinguere il tipo di file, dalle informazioni contenute in una variabile di tipo '**mode_t**', ammesso che queste informazioni siano incluse effettivamente:

```
//
// macroistruzioni per la verifica del tipo di file.
//
#define S_ISBLK(m)    (((m) & S_IFMT) == S_IFBLK)    // [1]
#define S_ISCHR(m)    (((m) & S_IFMT) == S_IFCHR)    // [2]
#define S_ISFIFO(m)   (((m) & S_IFMT) == S_IFIFO)    // [3]
#define S_ISREG(m)    (((m) & S_IFMT) == S_IFREG)    // [4]
#define S_ISDIR(m)    (((m) & S_IFMT) == S_IFDIR)    // [5]
#define S_ISLNK(m)    (((m) & S_IFMT) == S_IFLNK)    // [6]
#define S_ISSOCK(m)   (((m) & S_IFMT) == S_IFSOCK)   // [7]
// [1] Dispositivo a blocchi.
// [2] Dispositivo a caratteri.
```

```
// [3] File FIFO.
// [4] File puro e semplice.
// [5] È una directory.
// [6] Collegamento simbolico.
// [7] Socket di dominio Unix.
```

70.2.2 Struttura «stat»

Il file ‘sys/stat.h’ include la dichiarazione del tipo derivato ‘**struct stat**’, con lo scopo di contenere le informazioni disponibili su di un file. La struttura deve contenere almeno i membri che appaiono nell’esempio successivo:

```
//
// Struttura «stat».
//
struct stat {
    dev_t     st_dev;     // File di dispositivo contenente
                          // il file.
    ino_t     st_ino;     // Numero di serie del file
                          // (inode).
    mode_t    st_mode;    // Tipo e permessi del file.
    nlink_t   st_nlink;   // Collegamenti fisici associati
                          // al file.
    uid_t     st_uid;     // Numero identificativo
                          // dell'utente proprietario.
    gid_t     st_gid;     // Numero identificativo del
                          // gruppo proprietario.
    dev_t     st_rdev;    // Numero del file dispositivo, se
                          // si tratta di un tale tipo di
                          // file.
    off_t     st_size;    // Se si tratta di un file vero
                          // e proprio, misura la dimensione
                          // del file; se si tratta di un
```

```
                                // collegamento simbolico, misura
                                // la dimensione del percorso che
                                // questo rappresenta; per altri
                                // casi il significato di questo
                                // campo non è precisato.
    time_t    st_atime;         // Data dell'ultimo accesso.
    time_t    st_mtime;         // Data dell'ultima modifica del
                                // contenuto.
    time_t    st_ctime;         // Data dell'ultima modifica dello
                                // stato.
    blksize_t st_blksize;       // Dimensione ottimale del blocco
                                // per le operazioni di I/O.
    blkcnt_t  st_blocks;        // Blocchi da 512 byte allocati
                                // per il file.
};
```

70.2.3 Prototipi di funzione

«

Il file 'sys/stat.h' include la dichiarazione di alcuni prototipi di funzione, come si vede nell'esempio seguente:

```
//
// Prototipi di funzione.
//
int    chmod  (const char *path, mode_t mode);
int    fchmod (int fdn, mode_t mode);
int    fstat  (int fdn, struct stat *buf);
int    lstat  (const char *restrict file,
               struct stat *restrict buf);
int    mkdir  (const char *path, mode_t mode);
int    mkfifo (const char *path, mode_t mode);
int    mknod  (const char *path, mode_t mode, dev_t dev);
int    stat   (const char *restrict path,
               struct stat *restrict buf);
mode_t umask  (mode_t mask);
```

70.3 File «strings.h»

Il file di intestazione ‘`strings.h`’ contiene i prototipi di alcune funzioni, legate prevalentemente alla scansione delle stringhe. Dal momento che viene usato il tipo derivato ‘**`size_t`**’, questo viene definito attraverso l’inclusione del file ‘**`stddef.h`**’.

```
#include <stddef.h>

int ffs          (int i);
int strcasecmp   (const char *s1, const char *s2);
int strncasecmp  (const char *s1, const char *s2, size_t n);
```

Lo standard prevede anche altri prototipi di funzioni ormai superate, che rimangono solo per compatibilità con il passato.

70.3.1 Funzione «ffs()»

La funzione ***ffs()*** serve a scandire i bit di un valore numerico intero, alla ricerca del primo bit a uno, partendo dalla posizione meno significativa, restituendo l’indice di tale bit, considerando il bit meno significativo avente indice uno. Pertanto, se il valore da scandito è pari a zero (non ha alcun bit a uno), la funzione restituisce zero.

Al di fuori dello standard, è probabile trovare delle altre funzioni simili a questa, per la scansione degli interi di tipo ‘**`long int`**’ e di tipo ‘**`long long int`**’. In tal caso, i nomi delle funzioni ulteriori possono essere ***ffsl()*** e ***ffsll()***.

70.3.2 Funzioni «strcasecmp()» e «strncasecmp()»

«

Le funzioni ***strcasecmp()*** e ***strncasecmp()*** servono a confrontare due stringhe, ignorando la differenza tra maiuscole e minuscole. Nel caso di ***strncasecmp()*** il confronto è limitato a una certa quantità massima di caratteri.

Se la configurazione locale è quella POSIX, il confronto avviene come se le due stringhe venissero convertite preventivamente in caratteri minuscoli; ma nel caso sia attiva una configurazione locale differente, lo standard non specifica in che modo la comparazione abbia luogo.

Il valore restituito dalle due funzioni dipende dal confronto lessicografico delle due stringhe. Se sono uguali (a parte la differenza tra maiuscole e minuscole), il risultato è zero; se la prima stringa è lessicograficamente precedente rispetto alla seconda, il valore restituito è inferiore a zero; se la prima stringa è lessicograficamente successiva alla seconda, il valore ottenuto è superiore a zero.

70.4 File «fcntl.h»

«

Il file di intestazione ‘`fcntl.h`’ riguarda la parte fondamentale della gestione dei file, attraverso i descrittori; precisamente si considerano la creazione, l’apertura e l’attribuzione di opzioni di funzionamento, mentre altre questioni sono gestite attraverso le definizioni contenute nel file ‘`unistd.h`’ (si veda eventualmente la realizzazione del file ‘`fcntl.h`’ e di alcune delle sue funzioni nei sorgenti di os32, sezione 95.6).

70.4.1 Tipi di dati derivati

Il file di intestazione ‘`fcntl.h`’ utilizza alcuni tipi di dati derivati, già definiti nel file ‘`sys/types.h`’:

```
#include <sys/types.h>  // mode_t
                        // off_t
                        // pid_t
```

70.4.2 Opzioni per la funzione «fcntl()»

Nel file di intestazione ‘`fcntl.h`’ si definiscono varie macro-variabili, di cui un primo insieme riguarda, quasi in modo esclusivo, l’uso della funzione ***fcntl()***.

```
//
// Valori per il secondo argomento della funzione fcntl().
//
#define F_DUPFD         0 // Duplicate file descriptor.
#define F_GETFD         1 // Get file descriptor flags.
#define F_SETFD         2 // Set file descriptor flags.
#define F_GETFL         3 // Get file status flags.
#define F_SETFL         4 // Set file status flags.
#define F_GETLK         5 // Get record locking information.
#define F_SETLK         6 // Set record locking information.
#define F_SETLKW        7 // Set record locking information;
                          // wait if blocked.
#define F_GETOWN        8 // Set owner of socket.
#define F_SETOWN        9 // Get owner of socket.
//
// Flag da impostare con:
//   fcntl (fd, F_SETFD, ...);
//
#define FD_CLOEXEC      1 // Chiude il descrittore del file
```

```
                                    // nel momento dell'esecuzione di
                                    // una funzione del gruppo
                                    // 'exec()'.
```

Le macro-variabili ***F_DUPFD***, ***F_GETFD***, ***F_SETFD***, ***F_GETFL***, ***F_SETFL***, ***F_GETLK***, ***F_SETLK***, ***F_SETLKW***, ***F_GETOWN*** e ***F_SETOWN***, servono per dichiarare il tipo di comando da dare alla funzione ***fcntl()***, attraverso il suo secondo parametro:

```
int fcntl (int fdn, int cmd, ...);
```

La macro-variabile ***FD_CLOEXEC*** riguarderebbe un insieme di indicatori associati a un descrittore di file (***fd_flags***), di cui però ne esiste uno solo, rappresentato dalla macro-variabile stessa. Utilizzando la funzione ***fcntl()*** e specificando il comando ***F_GETFD*** è possibile ottenere lo stato di questi indicatori (ovvero solo ***FD_CLOEXEC***), mentre con il comando ***F_SETFD*** è possibile modificare questo stato. Quando l'indicatore ***FD_CLOEXEC*** risulta attivo per un certo descrittore di file, se viene eseguita la sostituzione del processo con l'ausilio di una funzione del gruppo ***exec**...()***, il descrittore in questione viene chiuso, mentre diversamente rimarrebbe aperto.

Nel file di intestazione '`fcntl.h`' vengono dichiarate anche delle macro-variabili per definire la modalità di accesso a un file, da usare prevalentemente con la funzione ***open()***:

```
//
// Indicatori per la creazione dei file, da usare nel
// parametro «oflag» della funzione open().
//
#define O_CREAT    000010 // Crea il file se non esiste già.
```

```
#define O_EXCL      000020 // Indicatore di accesso
                           // esclusivo.
#define O_NOCTTY    000040 // Non assegna un terminale di
                           // controllo.
#define O_TRUNC     000100 // Indicatore di troncamento.
//
// Indicatori dello stato dei file,
// usati nelle funzioni open() e fcntl().
//
#define O_APPEND    000200 // Scrittura in aggiunta.
#define O_DSYNC     000400 // Scrittura sincronizzata
                           // dei dati.
#define O_NONBLOCK  001000 // Modalità non bloccante.
#define O_RSYNC     002000 // Lettura sincronizzata.
#define O_SYNC      004000 // Scrittura sincronizzata
                           // dei file.
//
// Maschera per la selezione delle sole modalità principali
// di accesso ai file.
//
#define O_ACCMODE   000003 // Seleziona gli ultimi due bit,
                           // che in questo caso individuano
                           // le modalità di accesso
                           // principali (lettura, scrittura
                           // ed entrambe) dalle altre
                           // modalità che sono già state
                           // descritte sopra.
//
// Modalità principali di accesso ai file, secondo la
// tradizione.
//
//#define O_RDONLY 000000 // Apertura in sola lettura.
//#define O_WRONLY 000001 // Apertura in sola scrittura.
//#define O_RDWR   000002 // Apertura in lettura e
```

```
//                              // scrittura.
//
// Modalità principali secondo le indicazioni per le
// realizzazioni future:
//
#define O_RDONLY    000001 // Apertura in lettura.
#define O_WRONLY    000002 // Apertura in scrittura.
#define O_RDWR      (O_RDONLY | O_WRONLY) // Apertura in
                                          // lettura
                                          // e scrittura.
```

70.4.3 Gestione del blocco dei file

«

Alcune macro-variabili e il tipo derivato '**`struct flock`**' servono per la gestione del blocco dei file.[1] L'esempio seguente si limita ai soli membri indispensabili:

```
//
// Valori per il tipo "l_type" (di una variabile strutturata
// di tipo "struct flock"), usato per il blocco dei record
// attraverso la funzione fcntl().
//
#define F_RDLCK         0       // Read lock.
#define F_WRLCK         1       // Write lock.
#define F_UNLCK         2       // Remove lock.
//
// Struttura «flock».
//
struct flock {
    short int l_type;   // Tipo del blocco: F_RDLCK,
                        // F_WRLCK, o F_UNLCK.
    short int l_whence; // Punto di riferimento iniziale a
                        // cui si somma poi il valore dello
                        // scostamento, rappresentato
```

```
                        // da "l_start".
    off_t     l_start;  // Scostamento che individua
                        // l'inizio dell'area bloccata.
                        // Lo scostamento può essere
                        // positivo o negativo.
    off_t     l_len;    // Dimensione dell'area bloccata:
                        // se si indica zero, significa che
                        // questa raggiunge la fine
                        // del file.
    pid_t     l_pid;    // Il numero del processo
                        // elaborativo che blocca
                        // l'area.
};
```

70.4.4 Funzioni

Sono presenti anche i prototipi delle funzioni ***creat()***, ***fcntl()*** e ***open()***:

```
//
// Prototipi di funzione.
//
int creat (const char *file, mode_t mode);
int fcntl (int fdn, int cmd, ...);
int open  (const char *file, int oflag, ...);
```

Per l'uso delle funzioni ***open()*** e ***creat()*** si veda la sezione 68.5.

La funzione ***fcntl()*** esegue un'operazione, definita dal parametro ***cmd***, sul descrittore richiesto come primo parametro (***fdn***). A seconda del tipo di operazione richiesta, potrebbero essere necessari degli argomenti ulteriori, i quali però non possono essere formalizzati in modo esatto nel prototipo della funzione. Il valore del secondo pa-

rametro che rappresenta l'operazione richiesta, va fornito in forma di costante simbolica, come descritto nell'elenco seguente, nel quale però sono descritti solo alcuni dei comandi possibili.

Sintassi	Descrizione
`fcntl (`***fdn***`, F_DUPFD, (int)` ***fdn_min***`)`	Richiede la duplicazione del descrittore di file ***fdn***, in modo tale che la copia abbia il numero di descrittore minore possibile, ma maggiore o uguale a quello indicato come argomento ***fdn_min***.
`fcntl (`***fdn***`, F_GETFD)` `fcntl (`***fdn***`, F_SETFD, (int)` ***fd_flags***`)`	Rispettivamente, legge o imposta, gli indicatori del descrittore di file ***fdn*** (eventualmente noti come *file descriptor flags* o solo *fd flags*). Per il momento, è possibile impostare un solo indicatore, ***FD_CLOEXEC***, pertanto, al posto di ***fd_flags*** si può mettere solo la costante ***FD_CLOEXEC***.
`fcntl (`***fdn***`, F_GETFL)` `fcntl (`***fdn***`, F_SETFL, (int)` ***fl_flags***`)`	Rispettivamente, legge o imposta, gli indicatori dello stato del file, relativi al descrittore ***fdn*** (eventualmente noti come *file flags* o solo *fl flags*). Per impostare questi indicatori, vanno combinate delle costanti simboliche: ***O_RDONLY***, ***O_WRONLY***, ***O_RDWR***, ***O_CREAT***, ***O_EXCL***, ***O_NOCTTY***, ***O_TRUNC***.

Il significato del valore restituito dalla funzione dipende dal tipo di operazione richiesta, come sintetizzato dalla tabella successiva, relativa ai soli comandi già apparsi. In generale, anche per gli altri comandi, un risultato erroneo viene comunque evidenziato dalla restituzione di un valore negativo.

Operazione richiesta	Significato del valore restituito
`F_DUPFD`	Si ottiene il numero del descrittore prodotto dalla copia, oppure −1 in caso di errore.
`F_GETFD`	Si ottiene il valore degli indicatori del descrittore (*fd flags*), oppure −1 in caso di errore.
`F_GETFL`	Si ottiene il valore degli indicatori del file (*fl flags*), oppure −1 in caso di errore.

70.5 File «unistd.h»

Il file di intestazione ‘`unistd.h`’ raccoglie un po’ di tutto ciò che riguarda le estensioni POSIX, pertanto è frequente il suo utilizzo (si veda eventualmente la realizzazione del file ‘`unistd.h`’ e di alcune delle sue funzioni nei sorgenti di os32, sezione 95.30).

Nel file ‘`unistd.h`’ si distingue la presenza di un elenco numeroso di macro-variabili con prefissi ***_POSIX_***…, ***_POSIX2_***… e ***_XOPEN_***…, con lo scopo di dichiarare le caratteristiche del sistema e della libreria. Per l’interrogazione delle caratteristiche o delle limitazioni del sistema, si utilizzano però delle funzioni apposite, costituite precisamente da ***pathconf()*** e ***sysconf()***, le quali utilizzano un proprio insieme di macro-variabili per individuare le caratteristiche da interrogare. Nel caso di ***pathconf()*** si aggiungono macro-variabili con prefisso ***_PC_***…; per ***sysconf()*** le macro-variabili hanno il prefisso ***_SC_***….

Nei prototipi di funzione si utilizzano diversi tipi derivati, già dichiarati nei file ‘sys/types.h’ e ‘inttypes.h’:

```
#include <sys/types.h>   // size_t, ssize_t, uid_t, gid_t,
                         // off_t, pid_t, useconds_t
#include <inttypes.h>    // intptr_t
```

Nel file deve anche essere dichiarato il valore per la macro-variabile ***NULL***; in questo caso viene incorporato il file ‘stddef.h’:

```
#include <stddef.h>      // NULL
```

70.5.1 Denominazione dei descrittori di file

«

Per dare un nome ai descrittori dei flussi standard, nel file ‘unistd.h’ si dichiarano tre macro-variabili, il cui valore è stabilito necessariamente:

```
#define STDIN_FILENO    0
#define STDOUT_FILENO   1
#define STDERR_FILENO   2
```

70.5.2 Verifica dei permessi di accesso

«

La funzione ***access()*** consente di verificare l’accessibilità di un file. Per questo richiede l’indicazione del percorso e di un valore che rappresenta i tipi di accesso che si vogliono considerare. Questi sono rappresentati dall’unione di quattro possibili macro-variabili:

```
#define R_OK    04   // Accessibilità in lettura.
#define W_OK    02   // Accessibilità in scrittura.
#define X_OK    01   // Accessibilità in esecuzione o
                     // attraversamento.
#define F_OK    00   // Esistenza del file.
```

I valori che questa macro-variabili possono avere devono essere tali da consentire la combinazione con l'operatore '|' (OR, bit per bit), così da poter verificare simultaneamente tutti gli aspetti dell'accesso al file.

70.5.3 Funzioni

Segue l'elenco dei prototipi delle funzioni principali del file 'unistd.h':

```
int           access      (const char *path, int mode);
unsigned int  alarm       (unsigned int seconds);
int           chdir       (const char *path);
int           chown       (const char *path, uid_t owner,
                           gid_t group);
int           close       (int fdn);
size_t        confstr     (int name, char *buffer,
                           size_t length);
int           dup         (int old_fdn);
int           dup2        (int old_fdn, int new_fdn);
int           execl       (const char *path,
                           const char *arg, ...);
int           execle      (const char *path,
                           const char *arg, ...,
                           char *const envp[]);
int           execlp      (const char *path,
                           const char *arg, ...);
int           execv       (const char *path,
                           char *const argv[]);
int           execve      (const char *path,
                           char *const argv[],
                           char *const envp[]);
int           execvp      (const char *path,
                           char *const argv[]);
```

```
void          _exit        (int status);
int           fchown       (int fdn, uid_t owner,
                            gid_t group);
pid_t         fork         (void);
long          fpathconf    (int fdn, int name);
int           ftruncate    (int fdn, off_t length);
char         *getcwd       (char *buffer, size_t size);
gid_t         getegid      (void);
uid_t         geteuid      (void);
gid_t         getgid       (void);
int           getgroups    (int size, gid_t list[]);
int           gethostname  (char *name, size_t length);
char         *getlogin     (void);
int           getlogin_r   (char *buffer, size_t size);
int           getopt       (int argv, char *const argv[],
                            const char *optstring);
pid_t         getpgrp      (void);
pid_t         getpid       (void);
pid_t         getppid      (void);
uid_t         getuid       (void);
int           isatty       (int fdn);
int           link         (const char *old_path,
                            const char *new_path);
off_t         lseek        (int fdn, off_t offset,
                            int whence);
long          pathconf     (const char *path, int name);
int           pause        (void);
int           pipe         (int fdn[2]);
ssize_t       read         (int fdn, void *buffer,
                            size_t length);
ssize_t       readlink     (const char *restrict path,
                            char *restrict buffer,
                            size_t size);
int           rmdir        (const char *path);
```

```
int             setegid     (gid_t egid);
int             seteuid     (uid_t euid);
int             setgid      (gid_t gid);
int             setpgid     (pid_t pid, pid_t pgid);
pid_t           setsid      (void);
int             setuid      (uid_t uid);
unsigned        sleep       (unsigned int seconds);
int             symlink     (const char *old_path,
                             const char *new_path);
long            sysconf     (int name);
pid_t           tcgetpgrp   (int fdn);
int             tcsetpgrp   (int fdn, pid_t pgrp);
char           *ttyname     (int fdn);
int             ttyname_r   (int fdn, char *buffer,
                             size_t length);
int             unlink      (const char *path);
ssize_t         write       (int fdn, const void *buffer,
                             size_t length);
```

70.5.4 Funzioni di servizio per la gestione di file e directory

Nelle sezioni seguenti si descrivono solo alcune delle funzioni destinate alla gestione di file e directory, il cui prototipo appare nel file ‘`unistd.h`’.

70.5.4.1 Funzione «access()»

La funzione ***access()*** consente di verificare l’accessibilità di un file, il cui percorso viene fornito tramite una stringa. L’accessibilità viene valutata in base a delle opzioni, con cui è possibile specificare a cosa si è interessati in modo preciso.

```
int access (const char *percorso, int modalità);
```

Il secondo parametro della funzione è un numero intero fornito normalmente attraverso l'indicazione di una macro-variabile che rappresenta simbolicamente il tipo di accesso che si intende verificare. Si può utilizzare ***F_OK*** per verificare l'esistenza del file o della directory; in alternativa, si possono usare le macro-variabili ***R_OK***, ***W_OK*** e ***X_OK***, sommabili assieme attraverso l'operatore OR binario, per la verifica dell'accessibilità in lettura, in scrittura e in esecuzione o attraversamento. Per esempio, scrivendo '**`R_OK|W_OK|X_OK`**' si vuole verificare che il file o la directory sia accessibile con tutti i tre permessi attivi.

Tabella 70.23. Macro-variabili usate per descrivere la modalità di accesso del secondo parametro della funzione ***access()***.

Macro-variabile	Descrizione
`F_OK`	Si richiede la verifica dell'esistenza del file o della directory. Questa opzione va usata da sola e non può essere sommata alle altre.
`R_OK`	Si richiede la verifica dell'accessibilità in lettura del file o della directory.
`W_OK`	Si richiede la verifica dell'accessibilità in scrittura del file o della directory.
`X_OK`	Si richiede la verifica dell'accessibilità in esecuzione del file o in attraversamento della directory.

La funzione restituisce il valore zero se il file o la directory risultano accessibili nel modo richiesto, altrimenti restituisce il valore −1 e si può verificare il tipo di errore valutando il contenuto della variabile

errno.

L'esempio seguente descrive completamente l'uso della funzione. Si può osservare che per valutare il successo dell'operazione, l'esito restituito dalla funzione viene invertito; inoltre, il contenuto della variabile ***errno*** viene considerato con l'aiuto della funzione ***perror()***. Il file dell'esempio dovrebbe essere disponibile presso *allegati/c/esempio-posix-unistd-access.c* .

```
#include <stdio.h>
#include <unistd.h>

int
main (void)
{
    const char *file_name = "/tmp/test";

    if (! access (file_name, F_OK))
      {
        printf ("Il file o la directory "
                "\"%s\" esiste.\n", file_name);
        if (! access (file_name, R_OK))
          {
            printf ("Il file o la directory "
                    "\"%s\" ", file_name);
            printf ("è accessibile in lettura.\n");
          }
        else
          {
            printf ("Il file o la directory "
                    "\"%s\" ", file_name);
            printf ("non è accessibile in lettura.\n");
            perror (NULL);
          }
        if (! access (file_name, W_OK))
```

```
          {
            printf ("Il file o la directory "
                    "\"%s\" ", file_name);
            printf ("è accessibile in scrittura.\n");
          }
        else
          {
            printf ("Il file o la directory "
                    "\"%s\" ", file_name);
            printf ("non è accessibile in scrittura.\n");
            perror (NULL);
          }
        if (! access (file_name, X_OK))
          {
            printf ("Il file o la directory "
                    "\"%s\" ", file_name);
            printf ("è accessibile in esecuzione o "
                    "attraversamento.\n");
          }
        else
          {
            printf ("Il file o la directory "
                    "\"%s\" ", file_name);
            printf ("non è accessibile in esecuzione "
                    "o attraversamento.\n");
            perror (NULL);
          }
      }
    else
      {
        printf ("Il file o la directory "
                "\"%s\" non esiste.\n", file_name);
        perror (NULL);
      }
```

```
    return (0);
}
```

70.5.4.2 Funzione «getcwd()»

La funzione ***getcwd()*** (*get current working directory*) consente di annotare in una stringa il percorso della directory corrente.

```
char *getcwd (char *buffer, size_t max);
```

Come si può vedere dal prototipo della funzione, occorre predisporre prima un array di caratteri, da usare come stringa, in cui la funzione va a scrivere il percorso trovato (con tanto di carattere nullo di terminazione). Come secondo parametro della funzione va indicata la dimensione massima dell'array, oltre la quale la scrittura non può andare.

La funzione restituisce il puntatore alla stringa contenente il percorso, ma se si verifica un errore restituisce il puntatore nullo e aggiorna la variabile ***errno*** con la specificazione della causa di tale errore.

L'esempio seguente descrive l'uso della funzione in modo molto semplice; in particolare, in caso di errore si usa la funzione ***perror()*** per visualizzarne una descrizione. Il file dell'esempio dovrebbe essere disponibile presso *allegati/c/esempio-posix-unistd-getcwd.c* .

```
#include <stdio.h>
#include <unistd.h>

int
main (void)
```

```
{
    char cwd[500];
    char *result;

    result = getcwd (cwd, 500);

    if (result == NULL)
      {
        perror (NULL);
      }
    else
      {
        printf ("%s\n", cwd);
      }
    return (0);
}
```

70.5.4.3 Funzione «chdir()»

La funzione ***chdir()*** (*change directory*) consente di cambiare la directory corrente del processo elaborativo.

```
int chdir (const char *path);
```

La funzione richiede come unico parametro la stringa che descrive il percorso da raggiungere. La funzione restituisce zero se l'operazione si conclude con successo, oppure il valore −1 in caso di errore, ma in tal caso viene modificata anche la variabile ***errno*** con l'indicazione più precisa dell'errore verificatosi.

L'esempio seguente mostra il comportamento della funzione, ma va osservato che l'effetto riguarda esclusivamente il processo in fun-

zione e non si riflette al processo che lo genera a sua volta. Per questa ragione il programma di esempio visualizza la posizione corrente raggiunta. Il file dell'esempio dovrebbe essere disponibile presso *allegati/c/esempio-posix-unistd-chdir.c* .

```
#include <stdio.h>
#include <unistd.h>

int
main (void)
{
    const char *path = "/tmp/test";
    char cwd[500] = {'\0'};

    if (! chdir (path))
      {
        getcwd (cwd, 500);
        printf ("Il processo passa alla "
                "directory \"%s\".\n", cwd);
      }
    else
      {
        printf ("Non è possibile passare alla "
                "directory \"%s\"!\n",
                path);
        perror (NULL);
      }

    return (0);
}
```

70.5.4.4 Funzioni «link()» e «unlink()»

«

La funzione ***link()*** consente la creazione di un nuovo collegamento fisico a partire da un file o da una directory già esistente, tenendo conto che la facoltà di creare un collegamento fisico a partire da una directory è una funzione privilegiata e il sistema operativo potrebbe non ammetterla in ogni caso. Per collegamento fisico si intende il riferimento contenuto in una directory, verso un certo file o una certa sottodirectory individuati numericamente da un numero inode. La funzione ***link()*** produce una sorta di copia del file, nel senso che si predispone un riferimento aggiuntivo alla stessa cosa, senza la duplicazione dei dati relativi. Per converso, la funzione ***unlink()*** elimina il riferimento a un file o a una directory, cosa che coincide con la cancellazione del file o della directory, se si tratta dell'ultimo riferimento esistente a tale oggetto nel file system. Anche in questo caso, va considerato in modo particolare l'eliminazione del riferimento a una directory: il sistema operativo può impedirlo se si tratta di una directory non vuota.

```
int link (const char *p1, const char *p2);
```

```
int unlink (const char *p);
```

I parametri delle due funzioni sono stringhe che descrivono il percorso di un file o di una directory. Nel caso di ***unlink()*** si indica solo il percorso da rimuovere, mentre con ***link()*** se ne indicano due: l'origine e la destinazione che si vuole creare.

Le due funzioni restituiscono un valore intero pari a zero se tutto

va bene, altrimenti restituiscono il valore −1, modificando anche il contenuto della variabile ***errno*** con un'informazione più precisa sull'accaduto.

Gli esempi seguenti mostrano il comportamento delle due funzioni. I file degli esempi dovrebbero essere disponibili presso *allegati/c/esempio-posix-unistd-link.c* e *allegati/c/esempio-posix-unistd-unlink.c* .

```
#include <stdio.h>
#include <unistd.h>

int
main (void)
{
    const char *old = "/tmp/test";
    const char *new = "/tmp/test.link";

    if (! link (old, new))
      {
        printf ("Creato il collegamento \"%s\".\n", new);
      }
    else
      {
        printf ("Non è possibile creare il "
                "collegamento \"%s\"!\n",
                new);
        perror (NULL);
      }

    return (0);
}
```

```
#include <stdio.h>
#include <unistd.h>
```

```
int
main (void)
{
    const char *name = "/tmp/test";

    if (! unlink (name))
      {
        printf ("Cancellato il file o la "
                "directory \"%s\".\n", name);
      }
    else
      {
        printf ("Non è possibile cancellare il file "
                "o la directory ");
        printf ("\"%s\"!\n", name);
        perror (NULL);
      }

    return (0);
}
```

70.6 File «dirent.h»

«

Il file di intestazione ‘`dirent.h`’ raccoglie ciò che serve per la gestione delle directory, attraverso flussi individuati da puntatori di tipo ‘**`DIR *`**’ (si veda eventualmente la realizzazione del file ‘`dirent.h`’ e di alcune delle sue funzioni nei sorgenti di os32, sezione 95.4). La gestione di tali flussi può avvenire attraverso i descrittori di file, così come già avviene per i flussi individuati da puntatori di tipo ‘**`FILE *`**’, ma ciò è una facoltà, non una necessità realizzativa.

70.6.1 Struttura «dirent»

Il file ‘`dirent.h`’ prevede la dichiarazione di un tipo derivato, denominato ‘**`struct dirent`**’, da usare per rappresentare una voce singola di una directory. I membri necessari di questa struttura sono ***d_ino***, per rappresentare il numero di inode della voce, e ***d_name[]*** per contenere il nome del file relativo, completo di carattere nullo di terminazione delle stringhe. La definizione del tipo ‘**`struct dirent`**’ potrebbe essere realizzata nel modo seguente:

```
#include <sys/types.h>  // ino_t
#include <limits.h>     // NAME_MAX
//
struct dirent {
    ino_t  d_ino;                   // Numero di inode
    char   d_name[NAME_MAX+1];      // NAME_MAX + '\0'
} __attribute__ ((packed));
```

70.6.2 Tipo «DIR»

Nel file ‘`dirent.h`’ è definito il tipo derivato ‘**`DIR`**’, con il quale si intende fare riferimento a un flusso che individua una directory aperta. Se la gestione di tali flussi avviene attraverso i descrittori di file, ci deve poter essere il riferimento al numero del descrittore relativo. Quello che segue è un esempio di una tale struttura, seguita dalla dichiarazione di un array per il contenimento delle informazioni su tutte le directory aperte del processo:

```
typedef struct {
    int            fdn;     // Numero del descrittore di file.
    struct dirent dir;      // Last directory item read.
} DIR;

extern DIR _directory_stream[]; // Array di directory che
                                // però non è previsto
                                // espressamente dallo
                                // standard.
```

Nella struttura di tipo '**DIR**' di questo esempio, viene inclusa una struttura di tipo '**struct dirent**', per permettere alla funzione ***readdir()*** di annotare l'ultima voce letta da una certa directory.

70.6.3 Prototipi di funzioni

«

La gestione delle directory in forma di flussi di tipo '**DIR ***' richiede alcune funzioni specifiche, di cui si trovano i prototipi nel file 'dirent.h':

```
int            closedir  (DIR *dp);
DIR           *opendir   (const char *name);
struct dirent *readdir   (DIR *dp);
void           rewinddir (DIR *dp);
```

La funzione ***opendir()*** apre un flusso corrispondente a una directory indicata tramite il suo percorso, restituendo il puntatore relativo. Una volta aperta una directory, si possono leggere le sue voci con la funzione ***readdir()***, la quale restituisce il puntatore di una variabile strutturata di tipo '**struct dirent**', all'interno della quale è possibile trarre i dati della voce letta: negli esempi di questo capitolo, tali informazioni sono incorporate nella struttura '**DIR**' che rappresenta la directory aperta. Ogni lettura fa avanzare alla voce successiva della directory e, se necessario, si può riposizionare l'indice di lettura

alla prima voce, con la funzione ***rewinddir()***. Per chiudere un flusso aperto, si usa la funzione ***closedir()***.

70.7 File «termios.h»

La gestione essenziale del terminale a caratteri è abbastanza complessa e si sintetizza con le definizioni del file ‘`termios.h`’ (si veda eventualmente la realizzazione del file ‘`termios.h`’ e di alcune delle sue funzioni nei sorgenti di os32, sezione 95.28). Lo standard prevede due modalità di inserimento: canonica e non canonica. Negli esempi di questo capitolo ci si sofferma su ciò che serve nel file ‘`termios.h`’ per gestire la modalità canonica, ovvero quella tradizionale per cui il dispositivo del terminale fornisce dati, solo dopo l’inserimento completo di una riga, confermato con un codice di interruzione di riga o con un altro codice che concluda comunque l’inserimento.

70.7.1 Tipi speciali

Nel file ‘`termios.h`’ vengono definiti alcuni tipi speciali per variabili scalari, che potrebbero essere descritti nel modo seguente:

```
typedef unsigned int tcflag_t;
typedef unsigned int speed_t;
typedef unsigned int cc_t;
```

Il tipo ‘**`tcflag_t`**’ viene usato nelle strutture di tipo ‘**`struct termios`**’, la cui definizione viene fatta nello stesso file ‘`termios.h`’; il tipo ‘**`speed_t`**’ serve per contenere il valore di una velocità di comunicazione del terminale; il tipo ‘**`cc_t`**’ serve per rappresentare un carattere di controllo, per la gestione del terminale.

70.7.2 Tipo «struct termios»

«

Nel file ‘`termios.h`’ viene definita la struttura ‘**`struct termios`**’, allo scopo di annotare tutte le informazioni sulla modalità di funzionamento di un certo dispositivo di terminale. Nell’esempio seguente si vedono solo i membri obbligatori, ma va considerata l’aggiunta di informazioni legate alla velocità di comunicazione, se il terminale ne deve fare uso. La definizione della struttura ‘**`struct termios`**’ richiede anche la dichiarazione della macro-variabile ***NCCS***, come si vede nell’esempio.

```
#define NCCS    11      // Dimensione dell'array 'c_cc[]'.
//
struct termios {
    tcflag_t c_iflag;
    tcflag_t c_oflag;
    tcflag_t c_cflag;
    tcflag_t c_lflag;
    cc_t     c_cc[NCCS];
};
```

Il membro ***c_iflag*** contiene gli indicatori che descrivono la modalità di inserimento di dati attraverso il terminale; il membro ***c_oflag*** contiene indicatori per la modalità di elaborazione dei dati in uscita, prima della loro lettura; il membro ***c_cflag*** contiene opzioni di controllo; il membro ***lflag*** contiene opzioni «locali» (qui, in particolare, si definisce se il terminale debba funzionare in modalità canonica o meno); l’array ***c_cc[]*** contiene i codici di caratteri da interpretare in modo speciale, per il controllo del funzionamento del terminale.

70.7.3 Codici di controllo

Durante l'inserimento di dati attraverso il terminale, alcuni codici possono assumere un significato particolare. Il valore numerico di questi codici è annotato nell'array ***c_cc[]*** che è membro della struttura di tipo '**struct termios**'; per farvi riferimento, l'indice da usare nell'array ***c_cc[]*** deve essere indicato attraverso una meta-variabile:

```
#define VEOF      0     // carattere EOF
#define VEOL      1     // carattere EOL
#define VERASE    2     // carattere ERASE
#define VINTR     3     // carattere INTR
#define VKILL     4     // carattere KILL
#define VMIN      5     // valore    MIN
#define VQUIT     6     // carattere QUIT
#define VSTART    7     // carattere START
#define VSTOP     8     // carattere STOP
#define VSUSP     9     // carattere SUSP
#define VTIME    10     // valore    TIME
```

Due valori di questo elenco fanno eccezione: '**VMIN**' e '**VTIME**', in quanto rappresentano invece un'informazione di tipo differente, necessaria per la gestione non canonica del terminale.

70.7.4 Indicatori per il membro «c_iflag»

Il membro ***c_iflag*** di una struttura di tipo '**struct termios**' può contenere degli indicatori indipendenti sulla configurazione per l'inserimento. Tali indicatori vanno forniti attraverso macro-variabili definite nel file '`termios.h`', di cui segue un esempio:

```
#define BRKINT    1 // Invia un segnale di interruzione
                    // in caso di ricevimento di un
                    // carattere INTR.
#define ICRNL     2 // Converte <CR> in <NL>.
#define IGNBRK    4 // Ignora il carattere INTR.
#define IGNCR     8 // Ignora il carattere <CR>.
#define IGNPAR   16 // Ignora i caratteri con errori di
                    // parità.
#define INLCR    32 // Converte <NL> in <CR>.
#define INPCK    64 // Abilita il controllo di parità.
#define ISTRIP  128 // Azzera l'ottavo bit dei caratteri.
#define IXOFF   256 // Abilita il controllo start/stop in
                    // ingresso.
#define IXON    512 // Abilita il controllo start/stop in
                    // uscita.
#define PARMRK 1024 // Segnala gli errori di parità.
```

70.7.5 Indicatori per il membro «c_oflag»

«

Il membro ***c_oflag*** di una struttura di tipo '**struct termios**' può contenere degli indicatori indipendenti sulla configurazione per l'output. Tali indicatori vanno forniti attraverso macro-variabili definite nel file '`termios.h`'. Di questi indicatori ne è obbligatorio solo uno: ***OPOST***, che ha lo scopo di abilitare una forma imprecisata di elaborazione dell'output.

```
#define OPOST     1     // post-process output
```

70.7.6 Indicatori per il membro «c_cflag»

«

Il membro ***c_cflag*** di una struttura di tipo '**struct termios**' può contenere degli indicatori indipendenti sulla configurazione per il controllo del terminale. Tale configurazione qui viene omessa.

70.7.7 Indicatori per il membro «c_lflag»

Il membro ***c_lflag*** di una struttura di tipo '**`struct termios`**' può contenere degli indicatori indipendenti sulla configurazione «locale» del terminale. Tali indicatori vanno forniti attraverso macro-variabili definite nel file '`termios.h`', di cui segue un esempio:

```
#define ECHO      1 // Abilita l'eco del terminale.
#define ECHOE     2 // Visualizza la cancellazione di un
                    // carattere.
#define ECHOK     4 // Visualizza l'eliminazione di una
                    // riga.
#define ECHONL    8 // Visualizza l'effetto del codice di
                    // interruzione di riga.
#define ICANON   16 // Modalità di inserimento canonica.
#define IEXTEN   32 // Modalità di inserimento estesa
                    // (non canonica).
#define ISIG     64 // Abilita i segnali.
#define NOFLSH  128 // Disabilita lo scarico della memoria
                    // tampone dopo un'interruzione o una
                    // conclusione forzata.
#define TOSTOP  256 // Invia il segnale SIGTTOU per l'output
                    // sullo sfondo.
```

L'indicatore ***ICANON*** consente di ottenere un funzionamento del terminale in modalità canonica.

70.7.8 Funzioni

Quanto descritto fino a questo punto sul file '`termios.h`' è ciò che poi serve per l'uso di alcune funzioni, il cui scopo è quello di configurare o interrogare la configurazione di un terminale. Nell'esempio seguente appaiono solo alcuni prototipi, assieme alla dichiarazione di alcune macro-variabili necessarie per la funzione ***tcsetattr()***.

```
#define TCSANOW   1 // Cambia gli attributi immediatamente.
#define TCSADRAIN 2 // Cambia gli attributi quando l'output
                    // è stato utilizzato.
#define TCSAFLUSH 3 // Cambia gli attributi quando l'output
                    // è stato utilizzato e scarica l'input
                    // ancora sospeso.

int tcgetattr (int fdn, struct termios *termios_p);
int tcsetattr (int fdn, int action,
               struct termios *termios_p);
```

Le due funzioni mostrate nell'esempio richiedono l'indicazione del numero di descrittore associato a un terminale aperto. Il parametro ***termios_p*** è un puntatore a una struttura con le informazioni sulla configurazione del terminale: la funzione ***tcgetattr()*** serve a ottenere la configurazione attuale, mentre la funzione ***tcsetattr()*** serve a modificarla. Il parametro ***action*** di ***tcsetattr()*** richiede di precisare la tempestività di tale modifica attraverso una delle macro-variabili elencate poco sopra.

70.8 Riferimenti

«

- Wikipedia, *C POSIX library*, *http://en.wikipedia.org/wiki/C_POSIX_library*
- The Open Group, *The Single UNIX® Specification, Version 2, System Interface & Headers Issue 5*, *http://pubs.opengroup.org/onlinepubs/007908799/xshix.html*
- Free Software Foundation, *The GNU C Library*, *http://www.gnu.org/software/libc/manual/*
- The Open Group, *The Single UNIX® Specification, Version 2, sys/types.h, sys/stat.h strings.h fcntl.h unistd.h dirent.h termios.h*

http://pubs.opengroup.org/onlinepubs/000095399/basedefs/sys/types.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/sys/stat.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/strings.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/fcntl.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/unistd.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/dirent.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/termios.h.html

- Pagina di manuale *fcntl(2)*
- The Open Group, *The Single UNIX® Specification, Version 2*, *fcntl*, *http://pubs.opengroup.org/onlinepubs/000095399/functions/fcntl.html*
- The Open Group, *The Single UNIX® Specification, Version 2*, *General Terminal Interface*, *http://pubs.opengroup.org/onlinepubs/000095399/basedefs/xbd_chap11.html*

[1] Di norma, il programmatore non accede direttamente a variabili di tipo '**`struct flock`**', perché per la gestione dei blocchi si usano semplicemente le funzioni appropriate.

Tabelle riepilogative della libreria C e POSIX

File «stdarg.h» 1359
File «limits.h» 1360
File «stdint.h» 1361
File «inttypes.h» 1365
File «ctype.h» 1375
File «stdlib.h» . . 1379
File «string.h» 1390
File «time.h» 1399
File «stdio.h» per la gestione dei file e degli errori 1406
File «stdio.h» per la composizione dell'output . . . 1421
File «stdio.h» per l'interpretazione dell'input 1428
File «assert.h» 1435
File «stddef.h» 1435
File «locale.h» 1436
File «regex.h» 1437
File «sys/stat.h» 1442

abort 1379 abs() 1379 asctime() 1399 assert() 1435
assert.h 1435 atexit() 1379 atof() 1379 atoi() 1379
atol() 1379 atoll() 1379 bsearch() 1379 BUFSIZ 1406
calloc() 1379 CHAR_BIT 1360 CHAR_MAX 1360 CHAR_MIN
1360 chmod() 1442 clearerr() 1406 clock() 1399
ctime() 1399 ctype.h 1375 difftime() 1399 div() 1379
EOF 1406 exit() 1379 fchmod() 1442 fclose() 1406
feof() 1406 ferror() 1406 fflush() 1406 fgetc() 1406
fgetpos() 1406 fgets() 1406 FILENAME_MAX 1406
fopen() 1406 FOPEN_MAX 1406 fprintf() 1421 fputc()
1406 fputs() 1406 fread() 1406 free() 1379 freopen()
1406 fscanf() 1428 fseek() 1406 fsetpos() 1406
fstat() 1442 ftell() 1406 fwrite() 1406 gets() 1406
gmtime() 1399 INT16_C() 1361 INT16_MAX 1361
INT16_MIN 1361 int16_t 1361 INT32_C() 1361
INT32_MAX 1361 INT32_MIN 1361 int32_t 1361
INT64_C() 1361 INT64_MAX 1361 INT64_MIN 1361
int64_t 1361 INT8_C() 1361 INT8_MAX 1361 INT8_MIN
1361 int8_t 1361 INTMAX_C() 1361 INTMAX_MAX 1361
INTMAX_MIN 1361 intmax_t 1361 INTPTR_MAX 1361
INTPTR_MIN 1361 intptr_t 1361 inttypes.h 1365
INT_FAST16_MAX 1361 INT_FAST16_MIN 1361
int_fast16_t 1361 INT_FAST32_MAX 1361
INT_FAST32_MIN 1361 int_fast32_t 1361
INT_FAST64_MAX 1361 INT_FAST64_MIN 1361
int_fast64_t 1361 INT_FAST8_MAX 1361
INT_FAST8_MIN 1361 int_fast8_t 1361
INT_LEAST16_MAX 1361 INT_LEAST16_MIN 1361
int_least16_t 1361 INT_LEAST32_MAX 1361

INT_LEAST32_MIN 1361 int_least32_t 1361
INT_LEAST64_MAX 1361 INT_LEAST64_MIN 1361
int_least64_t 1361 INT_LEAST8_MAX 1361
INT_LEAST8_MIN 1361 int_least8_t 1361 INT_MAX 1360
INT_MIN 1360 isalnum() 1375 isalpha() 1375
isblank() 1375 iscntrl() 1375 isdigit() 1375
isgraph() 1375 islower() 1375 isprint() 1375
ispunct() 1375 isspace() 1375 isupper() 1375
isxdigit() 1375 labs() 1379 ldiv() 1379 limits.h
1360 llabs() 1379 lldiv() 1379 LLONG_MAX 1360
LLONG_MIN 1360 locale.h 1436 localtime() 1399
LONG_MAX 1360 LONG_MIN 1360 lstat() 1442 L_tmpnam
1406 malloc() 1379 mblen() 1379 mbstowcs() 1379
mbtowc() 1379 MB_LEN_MAX 1360 memchr() 1390
memcmp() 1390 memcpy() 1390 memmove() 1390
memset() 1390 mkdir() 1442 mkfifo() 1442 mknod()
1442 mktime() 1399 offsetof() 1435 perror() 1406
PRId16 1365 PRId32 1365 PRId64 1365 PRId8 1365
PRIdFAST16 1365 PRIdFAST32 1365 PRIdFAST64 1365
PRIdFAST8 1365 PRIdLEAST16 1365 PRIdLEAST32 1365
PRIdLEAST64 1365 PRIdLEAST8 1365 PRIdMAX 1365
PRIdPTR 1365 PRIi16 1365 PRIi32 1365 PRIi64 1365
PRIi8 1365 PRIiFAST16 1365 PRIiFAST32 1365
PRIiFAST64 1365 PRIiFAST8 1365 PRIiLEAST16 1365
PRIiLEAST32 1365 PRIiLEAST64 1365 PRIiLEAST8 1365
PRIiMAX 1365 PRIiPTR 1365 printf() 1421 PRIo16 1365
PRIo32 1365 PRIo64 1365 PRIo8 1365 PRIoFAST16 1365
PRIoFAST32 1365 PRIoFAST64 1365 PRIoFAST8 1365
PRIoLEAST16 1365 PRIoLEAST32 1365 PRIoLEAST64 1365

PRIoLEAST8 1365 PRIoMAX 1365 PRIoPTR 1365 PRIu16 1365 PRIu32 1365 PRIu64 1365 PRIu8 1365 PRIuFAST16 1365 PRIuFAST32 1365 PRIuFAST64 1365 PRIuFAST8 1365 PRIuLEAST16 1365 PRIuLEAST32 1365 PRIuLEAST64 1365 PRIuLEAST8 1365 PRIuMAX 1365 PRIuPTR 1365 PRIx16 1365 PRIX16 1365 PRIx32 1365 PRIX32 1365 PRIx64 1365 PRIX64 1365 PRIX8 1365 PRIx8 1365 PRIxFAST16 1365 PRIXFAST16 1365 PRIxFAST32 1365 PRIXFAST32 1365 PRIxFAST64 1365 PRIXFAST64 1365 PRIXFAST8 1365 PRIxFAST8 1365 PRIxLEAST16 1365 PRIXLEAST16 1365 PRIxLEAST32 1365 PRIXLEAST32 1365 PRIxLEAST64 1365 PRIXLEAST64 1365 PRIXLEAST8 1365 PRIxLEAST8 1365 PRIxMAX 1365 PRIXMAX 1365 PRIxPTR 1365 PRIXPTR 1365 PTRDIFF_MAX 1361 PTRDIFF_MIN 1361 ptrdiff_t 1361 putchar() 1406 puts() 1406 qsort() 1379 rand() 1379 realloc() 1379 regcomp() 1437 regerror() 1437 regex.h 1437 regexec() 1437 regex_t 1437 regfree() 1437 regmatch_t 1437 1437 regoff_t 1437 remove() 1406 rename() 1406 rewind() 1406 re_sub 1437 rm_se 1437 rm_so 1437 scanf() 1428 SCHAR_MAX 1360 SCHAR_MIN 1360 SCNd16 1365 SCNd32 1365 SCNd64 1365 SCNd8 1365 SCNdFAST16 1365 SCNdFAST32 1365 SCNdFAST64 1365 SCNdFAST8 1365 SCNdLEAST16 1365 SCNdLEAST32 1365 SCNdLEAST64 1365 SCNdLEAST8 1365 SCNdMAX 1365 SCNdPTR 1365 SCNi16 1365 SCNi32 1365 SCNi64 1365 SCNi8 1365 SCNiFAST16 1365 SCNiFAST32 1365 SCNiFAST64 1365 SCNiFAST8 1365 SCNiLEAST16 1365 SCNiLEAST32 1365 SCNiLEAST64 1365 SCNiLEAST8 1365 SCNiMAX 1365 SCNiPTR 1365 SCNo16 1365 SCNo32

1365 SCNo64 1365 SCNo8 1365 SCNoFAST16 1365 SCNoFAST32 1365 SCNoFAST64 1365 SCNoFAST8 1365 SCNoLEAST16 1365 SCNoLEAST32 1365 SCNoLEAST64 1365 SCNoLEAST8 1365 SCNoMAX 1365 SCNoPTR 1365 SCNu16 1365 SCNu32 1365 SCNu64 1365 SCNu8 1365 SCNuFAST16 1365 SCNuFAST32 1365 SCNuFAST64 1365 SCNuFAST8 1365 SCNuLEAST16 1365 SCNuLEAST32 1365 SCNuLEAST64 1365 SCNuLEAST8 1365 SCNuMAX 1365 SCNuPTR 1365 SCNx16 1365 SCNx32 1365 SCNx64 1365 SCNx8 1365 SCNxFAST16 1365 SCNxFAST32 1365 SCNxFAST64 1365 SCNxFAST8 1365 SCNxLEAST16 1365 SCNxLEAST32 1365 SCNxLEAST64 1365 SCNxLEAST8 1365 SCNxMAX 1365 SCNxPTR 1365 SEEK_CUR 1406 SEEK_END 1406 SEEK_SET 1406 setbuf() 1406 setlocale() 1436 setvbuf() 1406 SHRT_MAX 1360 SHRT_MIN 1360 SIG_ATOMIC_MAX 1361 SIG_ATOMIC_MIN 1361 sig_atomic_t 1361 SIZE_MAX 1361 size_t 1361 snprintf() 1421 sprintf() 1421 srand() 1379 sscanf() 1428 stat() 1442 stat.h 1442 stdarg.h 1359 stddef.h 1435 stdint.h 1361 stdio.h 1406 1421 1428 stdlib.h 1379 strcat() 1390 strchr() 1390 strcmp() 1390 strcoll() 1390 strcpy() 1390 strcspn() 1390 strerror() 1390 strftime() 1399 string.h 1390 strlen() 1390 strncat() 1390 strncmp() 1390 strncpy() 1390 strpbrk() 1390 strrchr() 1390 strspn() 1390 strstr() 1390 strtod() 1379 strtof() 1379 strtok() 1390 strtol() 1379 strtold() 1379 strtoll() 1379 strtoul() 1379 strtoull() 1379 strxfrm() 1390 st_atime 1442 st_blksize 1442 st_blocks 1442 st_ctime 1442 st_dev 1442 st_gid 1442

st_ino 1442 st_mode 1442 st_mtime 1442 st_nlink 1442
st_rdev 1442 st_size 1442 st_uid 1442 S_IFBLK 1442
S_IFCHR 1442 S_IFDIR 1442 S_IFIFO 1442 S_IFLNK 1442
S_IFMT 1442 S_IFREG 1442 S_IFSOCK 1442 S_IRGRP 1442
S_IROTH 1442 S_IRUSR 1442 S_IRWXG 1442 S_IRWXO 1442
S_IRWXU 1442 S_ISBLK() 1442 S_ISCHR() 1442
S_ISDIR() 1442 S_ISFIFO() 1442 S_ISGID 1442
S_ISLNK() 1442 S_ISREG() 1442 S_ISSOCK() 1442
S_ISUID 1442 S_ISVTX 1442 S_IWGRP 1442 S_IWOTH 1442
S_IWUSR 1442 S_IXGRP 1442 S_IXOTH 1442 S_IXUSR 1442
time() 1399 time.h 1399 tmpfile() 1406 tmpnam() 1406
TMP_MAX 1406 tolower() 1375 toupper() 1375
UCHAR_MAX 1360 UINT16_C() 1361 UINT16_MAX 1361
uint16_t 1361 UINT32_C() 1361 UINT32_MAX 1361
uint32_t 1361 UINT64_C() 1361 UINT64_MAX 1361
uint64_t 1361 UINT8_C() 1361 UINT8_MAX 1361
uint8_t 1361 UINTMAX_C() 1361 UINTMAX_MAX 1361
uintmax_t 1361 UINTPTR_MAX 1361 uintptr_t 1361
UINT_FAST16_MAX 1361 uint_fast16_t 1361
UINT_FAST32_MAX 1361 uint_fast32_t 1361
UINT_FAST64_MAX 1361 uint_fast64_t 1361
UINT_FAST8_MAX 1361 uint_fast8_t 1361
UINT_LEAST16_MAX 1361 uint_least16_t 1361
UINT_LEAST32_MAX 1361 uint_least32_t 1361
UINT_LEAST64_MAX 1361 uint_least64_t 1361
UINT_LEAST8_MAX 1361 uint_least8_t 1361 UINT_MAX
1360 ULLONG_MAX 1360 ULONG_MAX 1360 umask() 1442
ungetc() 1406 USHRT_MAX 1360 va_arg() 1359
va_copy() 1359 va_end() 1359 va_list 1359

va_start() 1359 vfprintf() 1421 vfscanf() 1428
vprintf() 1421 vscanf() 1428 vsnprintf() 1421
vsprintf() 1421 vsscanf() 1428 WCHAR_MAX 1361
WCHAR_MIN 1361 wchar_t 1361 wcstombs() 1379
wctomb() 1379 WINT_MAX 1361 WINT_MIN 1361 wint_t
1361 _Exit() 1379 _IOFBF 1406 _IOLBF 1406 _IONBF 1406

File «stdarg.h»

Macroistruzione	Descrizione
`void va_start (va_list` ***ap***`,` `  `***parametro_n***`);`	Inizializza la variabile ***ap***, di tipo '**`va_list`**', in modo che punti all'area di memoria immediatamente successiva al parametro indicato, il quale deve essere l'ultimo.
tipo `va_arg (va_list` ***ap***`,` ***tipo***`);`	Restituisce il contenuto dell'area di memoria a cui punta ***ap***, utilizzando il tipo indicato, incrementando contestualmente il puntatore in modo che, al termine, si trovi nell'area di memoria immediatamente successiva.
`void va_copy (va_list` ***dst***`,` `  va_list` ***org***`);`	Copia il puntatore ***org*** nella variabile ***dst***.
`void va_end (va_list` ***ap***`);`	Conclude l'utilizzo del puntatore ***ap***.

«

Macro-variabile	Descrizione
`CHAR_BIT`	Quantità di bit utilizzata per rappresentare il tipo '**`char`**', con o senza segno. In altri termini è l'unità di memorizzazione più piccola con cui si può gestire l'insieme di caratteri minimo. Di norma si tratta di 8 bit.
`SCHAR_MIN` `SCHAR_MAX`	Il valore minimo e il valore massimo rappresentabile in una variabile '**`signed char`**'.
`UCHAR_MAX`	Il valore massimo rappresentabile in una variabile '**`unsigned char`**'. Il valore minimo è zero.
`CHAR_MIN` `CHAR_MAX`	Il valore minimo e il valore massimo rappresentabile in una variabile '**`char`**'. Questi valori dipendono dal fatto che il tipo '**`char`**' sia da intendere equivalente a un tipo '**`unsigned char`**' o '**`signed char`**', da cui ereditano i limiti.
`MB_LEN_MAX`	La quantità massima di byte che possono essere usati per rappresentare un carattere multibyte, qualunque sia la configurazione locale.
`SHRT_MIN` `SHRT_MAX`	Il valore minimo e il valore massimo rappresentabile in una variabile '**`short int`**'.
`USHRT_MAX`	Il valore massimo rappresentabile in una variabile '**`unsigned short int`**'. Il valore minimo è zero.
`INT_MIN` `INT_MAX`	Il valore minimo e il valore massimo rappresentabile in una variabile '**`int`**'.
`UINT_MAX`	Il valore massimo rappresentabile in una variabile '**`unsigned int`**'. Il valore minimo è zero.

Macro-variabile	Descrizione
`LONG_MIN` `LONG_MAX`	Il valore minimo e il valore massimo rappresentabile in una variabile '**`long int`**'.
`ULONG_MAX`	Il valore massimo rappresentabile in una variabile '**`unsigned long int`**'. Il valore minimo è zero.
`LLONG_MIN` `LLONG_MAX`	Il valore minimo e il valore massimo rappresentabile in una variabile '**`long long int`**'.
`ULLONG_MAX`	Il valore massimo rappresentabile in una variabile '**`unsigned long long int`**'. Il valore minimo è zero.

File «stdint.h»

Con segno	Senza segno	Descrizione
`int8_t` `int16_t` `int32_t` `int64_t`	`uint8_t` `uint16_t` `uint32_t` `uint64_t`	Tipo intero, facoltativo, il cui rango è prestabilito esattamente.

Con segno	Senza segno	Descrizione
`INT8_MIN` `INT8_MAX` `INT16_MIN` `INT16_MAX` `INT32_MIN` `INT32_MAX` `INT64_MIN` `INT64_MAX`	`UINT8_MAX` `UINT16_MAX` `UINT32_MAX` `UINT64_MAX`	Limiti minimi e massimi dei tipi '**`int`*n*`_t`**' e '**`uint`*n*`_t`**'.
`int_least8_t` `int_least16_t` `int_least32_t` `int_least64_t`	`uint_least8_t` `uint_least16_t` `uint_least32_t` `uint_least64_t`	Tipo intero con un rango minimo stabilito.

Con segno	Senza segno	Descrizione
`INT_LEAST8_MIN` `INT_LEAST8_MAX` `INT_LEAST16_MIN` `INT_LEAST16_MAX` `INT_LEAST32_MIN` `INT_LEAST32_MAX` `INT_LEAST64_MIN` `INT_LEAST64_MAX`	`UINT_LEAST8_MAX` `UINT_LEAST16_MAX` `UINT_LEAST32_MAX` `UINT_LEAST64_MAX`	Limiti minimi e massimi dei tipi '**`int_least`*n*`_t`**' e '**`uint_least`*n*`_t`**'.
`INT8_C(`***val***`)` `INT16_C(`***val***`)` `INT32_C(`***val***`)` `INT64_C(`***val***`)`	`UINT8_C(`***val***`)` `UINT16_C(`***val***`)` `UINT32_C(`***val***`)` `UINT64_C(`***val***`)`	Macroistruzione per attribuire l'estensione che definisce il tipo corretto a un valore costante, da intendere secondo il tipo '**`int_least`*n*`_t`**' o '**`uint_least`*n*`_t`**'.
`int_fast8_t` `int_fast16_t` `int_fast32_t` `int_fast64_t`	`uint_fast8_t` `uint_fast16_t` `uint_fast32_t` `uint_fast64_t`	Tipo intero con un rango minimo stabilito, con caratteristiche ottimali per la velocità elaborativa.

Con segno	Senza segno	Descrizione
`INT_FAST8_MIN` `INT_FAST8_MAX` `INT_FAST16_MIN` `INT_FAST16_MAX` `INT_FAST32_MIN` `INT_FAST32_MAX` `INT_FAST64_MIN` `INT_FAST64_MAX`	`UINT_FAST8_MAX` `UINT_FAST16_MAX` `UINT_FAST32_MAX` `UINT_FAST64_MAX`	Limiti minimi e massimi dei tipi '**`int_fast`*n*`_t`**' e '**`uint_fast`*n*`_t`**'.
`intptr_t`	`uintptr_t`	Tipo facoltativo intero capace di contenere il valore di un puntatore, convertibile da e verso '**`void *`**'.
`INTPTR_MIN` `INTPTR_MAX`	`UINTPTR_MAX`	Limiti minimi e massimi dei tipi '**`intptr_t`**' e '**`uintptr_t`**'.
`intmax_t`	`uintmax_t`	Tipo intero di rango massimo.
`INTMAX_MIN` `INTMAX_MAX`	`UINTMAX_MAX`	Limiti minimi e massimi dei tipi '**`intmax_t`**' e '**`uintmax_t`**'.

Con segno	Senza segno	Descrizione
`INTMAX_C(`***val***`)`	`UINTMAX_C(`***val***`)`	Macroistruzione per attribuire l'estensione che definisce il tipo corretto a un valore costante, da intendere secondo il tipo '**`intmax_t`**' o '**`uintmax_t`**'.
`PTRDIFF_MIN` `PTRDIFF_MAX`		Limiti minimi e massimi del tipo '**`ptrdiff_t`**'.
`SIG_ATOMIC_MIN` `SIG_ATOMIC_MAX`		Limiti minimi e massimi del tipo '**`sig_atomic_t`**'.
	`SIZE_MAX`	Limite massimo del tipo '**`size_t`**' (senza segno).
`WCHAR_MIN` `WCHAR_MAX`		Limiti minimi e massimi del tipo '**`wchar_t`**'.
`WINT_MIN` `WINT_MAX`		Limiti minimi e massimi del tipo '**`wint_t`**'.

File «inttypes.h»

Macro-variabili per la composizione dell'output	Macro-variabili per l'interpretazione dell'input	Esempi schematici
PRId8 PRId16 PRId32 PRId64	SCNd8 SCNd16 SCNd32 SCNd64	`int32_t i = INT32_MAX;` `…` `printf ("i = %010" PRId32 "\n", i);` `…` `scanf ("%" SCNd32, &i);`
PRIi8 PRIi16 PRIi32 PRIi64	SCNi8 SCNi16 SCNi32 SCNi64	`int32_t i = INT32_MAX;` `…` `printf ("i = %010" PRIi32 "\n", i);` `…` `scanf ("%" SCNi32, &i);`
PRIdLEAST8 PRIdLEAST16 PRIdLEAST32 PRIdLEAST64	SCNdLEAST8 SCNdLEAST16 SCNdLEAST32 SCNdLEAST64	`int_least32_t i = INT_LEAST32_MAX;` `…` `printf ("i = %010" PRIdLEAST32 "\n", i);` `…` `scanf ("%" SCNdLEAST32, &i);`

Macro-variabili per la composizione dell'output	Macro-variabili per l'interpretazione dell'input	Esempi schematici
PRIiLEAST8 PRIiLEAST16 PRIiLEAST32 PRIiLEAST64	SCNiLEAST8 SCNiLEAST16 SCNiLEAST32 SCNiLEAST64	`int_least32_t i = INT_LEAST32_MAX;` `…` `printf ("i = %010" PRIiLEAST32 "\n", i);` `…` `scanf ("%" SCNiLEAST32, &i);`
PRIdFAST8 PRIdFAST16 PRIdFAST32 PRIdFAST64	SCNdFAST8 SCNdFAST16 SCNdFAST32 SCNdFAST64	`int_fast32_t i = INT_FAST32_MAX;` `…` `printf ("i = %010" PRIdFAST32 "\n", i);` `…` `scanf ("%" SCNdFAST32, &i);`
PRIiFAST8 PRIiFAST16 PRIiFAST32 PRIiFAST64	SCNiFAST8 SCNiFAST16 SCNiFAST32 SCNiFAST64	`int_fast32_t i = INT_FAST32_MAX;` `…` `printf ("i = %010" PRIiFAST32 "\n", i);` `…` `scanf ("%" SCNiFAST32, &i);`

Macro-variabili per la composizione dell'output	Macro-variabili per l'interpretazione dell'input	Esempi schematici
PRIdMAX PRIdPTR PRIiMAX PRIiPTR	SCNdMAX SCNdPTR SCNiMAX SCNiPTR	`intmax_t i = INTMAX_MAX;` `…` `printf ("i = %020" PRIiMAX "\n", i);` `…` `scanf ("%" SCNiMAX, &i);`
PRIo8 PRIo16 PRIo32 PRIo64	SCNo8 SCNo16 SCNo32 SCNo64	`uint32_t i = UINT32_MAX;` `…` `printf ("i = %011" PRIo32 "\n", i);` `…` `scanf ("%" SCNo32, &i);`
PRIoLEAST8 PRIoLEAST16 PRIoLEAST32 PRIoLEAST64	SCNoLEAST8 SCNoLEAST16 SCNoLEAST32 SCNoLEAST64	`uint_least32_t i = UINT_LEAST32_MAX;` `…` `printf ("i = %011" PRIoLEAST32 "\n", i);` `…` `scanf ("%" SCNoLEAST32, &i);`

Macro-variabili per la composizione dell'output	Macro-variabili per l'interpretazione dell'input	Esempi schematici
PRIoFAST8 PRIoFAST16 PRIoFAST32 PRIoFAST64	SCNoFAST8 SCNoFAST16 SCNoFAST32 SCNoFAST64	`uint_fast32_t i = UINT_FAST32_MAX;` `…` `printf ("i = %011" PRIoFAST32 "\n", i);` `…` `scanf ("%" SCNoFAST32, &i);`
PRIoMAX PRIoPTR	SCNoMAX SCNoPTR	`uintmax_t i = INTMAX_MAX;` `…` `printf ("i = %022" PRIoMAX "\n", i);` `…` `scanf ("%" SCNoMAX, &i);`
PRIu8 PRIu16 PRIu32 PRIu64	SCNu8 SCNu16 SCNu32 SCNu64	`uint32_t i = UINT32_MAX;` `…` `printf ("i = %010" PRIu32 "\n", i);` `…` `scanf ("%" SCNu32, &i);`
PRIuLEAST8 PRIuLEAST16 PRIuLEAST32 PRIuLEAST64	SCNuLEAST8 SCNuLEAST16 SCNuLEAST32 SCNuLEAST64	`uint_least32_t i = UINT_LEAST32_MAX;` `…` `printf ("i = %010" PRIuLEAST32 "\n", i);` `…` `scanf ("%" SCNuLEAST32, &i);`

Macro-variabili per la composizione dell'output	Macro-variabili per l'interpretazione dell'input	Esempi schematici
PRIuFAST8 PRIuFAST16 PRIuFAST32 PRIuFAST64	SCNuFAST8 SCNuFAST16 SCNuFAST32 SCNuFAST64	`uint_fast32_t i = UINT_FAST32_MAX;` `…` `printf ("i = %010" PRIuFAST32 "\n", i);` `…` `scanf ("%" SCNuFAST32, &i);`
PRIuMAX PRIuPTR	SCNuMAX SCNuPTR	`uintmax_t i = INTMAX_MAX;` `…` `printf ("i = %022" PRIuMAX "\n", i);` `…` `scanf ("%" SCNuMAX, &i);`
PRIx8 PRIX8 PRIx16 PRIX16 PRIx32 PRIX32 PRIx64 PRIX64	SCNx8 SCNx16 SCNx32 SCNx64	`uint32_t i = UINT32_MAX;` `…` `printf ("i = %08" PRIx32 "\n", i);` `…` `scanf ("%" SCNx32, &i);`

Macro-variabili per la composizione dell'output	Macro-variabili per l'interpretazione dell'input	Esempi schematici
PRIxLEAST8 PRIXLEAST8 PRIxLEAST16 PRIXLEAST16 PRIxLEAST32 PRIXLEAST32 PRIxLEAST64 PRIXLEAST64	SCNxLEAST8 SCNxLEAST16 SCNxLEAST32 SCNxLEAST64	`uint_least32_t i = UINT_LEAST32_MAX;` `…` `printf ("i = %08" PRIxLEAST32 "\n", i);` `…` `scanf ("%" SCNxLEAST32, &i);`
PRIxFAST8 PRIXFAST8 PRIxFAST16 PRIXFAST16 PRIxFAST32 PRIXFAST32 PRIxFAST64 PRIXFAST64	SCNxFAST8 SCNxFAST16 SCNxFAST32 SCNxFAST64	`uint_fast32_t i = UINT_FAST32_MAX;` `…` `printf ("i = %08" PRIxFAST32 "\n", i);` `…` `scanf ("%" SCNxFAST32, &i);`
PRIxMAX PRIXMAX PRIxPTR PRIXPTR	SCNxMAX SCNxPTR	`uintmax_t i = UINTMAX_MAX;` `…` `printf ("i = %016" PRIxMAX "\n", i);` `…` `scanf ("%" SCNxMAX, &i);`

Funzione	Descrizione
`intmax_t imaxabs (intmax_t` ***j*** `);`	Restituisce il valore assoluto del numero passato come argomento.
`imaxdiv_t imaxdiv (intmax_t` ***numer*** `,` `intmax_t` ***denom*** `);`	Restituisce il risultato della divisione dei due argomenti, in una struttura contenente il risultato intero e il resto della divisione.

Funzione	Descrizione
`intmax_t strtoimax (const char *restrict s, char **restrict p, int base);` `uintmax_t strtouimax (const char *restrict s, char **restrict p, int base);`	Converte la stringa fornita come primo argomento in un numero intero, come si vede dal modello sintattico, interpretando la stringa come numero espresso nella base rappresentata dal parametro ***base***. La conversione avviene fino a dove è possibile riconoscere caratteri che compongono un valore valido; se il secondo argomento è un puntatore a un puntatore valido (un puntatore a un'area di memoria che può contenere a sua volta un puntatore dal tipo '**`char`**'), al suo interno viene memorizzato l'indirizzo finale della scansione, a partire dal quale si trovano caratteri non decifrabili, oppure dove si trova il carattere nullo di terminazione della stringa.

Funzione	Descrizione
`intmax_t wcstoimax (const wchar_t *restrict wcs, wchar_t **restrict p, int base);` `uintmax_t wcstouimax (const wchar_t *restrict wcs, wchar_t **restrict p, int base);`	Converte la stringa estesa fornita come primo argomento in un numero intero, come si vede dal modello sintattico, interpretando la stringa estesa come numero espresso nella base rappresentata dal parametro ***base***. La conversione avviene fino a dove è possibile riconoscere caratteri estesi che compongono un valore valido; se il secondo argomento è un puntatore a un puntatore valido (un puntatore a un'area di memoria che può contenere a sua volta un puntatore dal tipo '**`wchar_t`**'), al suo interno viene memorizzato l'indirizzo finale della scansione, a partire dal quale si trovano caratteri estesi non decifrabili, oppure dove si trova il carattere nullo di terminazione della stringa.

Funzione	Descrizione
`int isalnum (int c);`	L'argomento rappresentato dal parametro ***c*** è un carattere senza segno convertito in un intero, oppure l'equivalente di '**`EOF`**'. La funzione restituisce un valore diverso da zero se l'argomento corrisponde a un carattere alfabetico o numerico. Equivale alla corrispondenza con ***isalpha()*** o con ***isdigit()***.
`int isalpha (int c);`	L'argomento rappresentato dal parametro ***c*** è un carattere senza segno convertito in un intero, oppure l'equivalente di '**`EOF`**'. La funzione restituisce un valore diverso da zero se l'argomento corrisponde a un carattere alfabetico. Equivale alla corrispondenza con ***isupper()*** o con ***islower()***.
`int isblank (int c);`	L'argomento rappresentato dal parametro ***c*** è un carattere senza segno convertito in un intero, oppure l'equivalente di '**`EOF`**'. La funzione restituisce un valore diverso da zero se l'argomento corrisponde a un carattere per la spaziatura orizzontale delle parole.

Funzione	Descrizione
`int iscntrl (int c);`	L'argomento rappresentato dal parametro ***c*** è un carattere senza segno convertito in un intero, oppure l'equivalente di '**`EOF`**'. La funzione restituisce un valore diverso da zero se l'argomento corrisponde a un carattere di controllo.
`int isdigit (int c);`	L'argomento rappresentato dal parametro ***c*** è un carattere senza segno convertito in un intero, oppure l'equivalente di '**`EOF`**'. La funzione restituisce un valore diverso da zero se l'argomento corrisponde a un carattere che rappresenta una cifra decimale.
`int isgraph (int c);`	L'argomento rappresentato dal parametro ***c*** è un carattere senza segno convertito in un intero, oppure l'equivalente di '**`EOF`**'. La funzione restituisce un valore diverso da zero se l'argomento corrisponde a un carattere che ha una rappresentazione grafica, escluso lo spazio.
`int islower (int c);`	L'argomento rappresentato dal parametro ***c*** è un carattere senza segno convertito in un intero, oppure l'equivalente di '**`EOF`**'. La funzione restituisce un valore diverso da zero se l'argomento corrisponde a un carattere alfabetico minuscolo.

Funzione	Descrizione
`int isprint (int c);`	L'argomento rappresentato dal parametro ***c*** è un carattere senza segno convertito in un intero, oppure l'equivalente di '**`EOF`**'. La funzione restituisce un valore diverso da zero se l'argomento corrisponde a un carattere che ha una rappresentazione grafica, incluso lo spazio.
`int ispunct (int c);`	L'argomento rappresentato dal parametro ***c*** è un carattere senza segno convertito in un intero, oppure l'equivalente di '**`EOF`**'. La funzione restituisce un valore diverso da zero se l'argomento corrisponde a un carattere di punteggiatura.
`int isspace (int c);`	L'argomento rappresentato dal parametro ***c*** è un carattere senza segno convertito in un intero, oppure l'equivalente di '**`EOF`**'. La funzione restituisce un valore diverso da zero se l'argomento corrisponde a un carattere di spaziatura, sia orizzontale, sia verticale, incluso il salto pagina e il ritorno a carrello.

Funzione	Descrizione
`int isupper (int c);`	L'argomento rappresentato dal parametro ***c*** è un carattere senza segno convertito in un intero, oppure l'equivalente di '**`EOF`**'. La funzione restituisce un valore diverso da zero se l'argomento corrisponde a un carattere alfabetico maiuscolo.
`int isxdigit (int c);`	L'argomento rappresentato dal parametro ***c*** è un carattere senza segno convertito in un intero, oppure l'equivalente di '**`EOF`**'. La funzione restituisce un valore diverso da zero se l'argomento corrisponde a un carattere che rappresenta una cifra esadecimale (espressa indifferentemente con lettere minuscole o maiuscole).
`int tolower (int c);`	L'argomento rappresentato dal parametro ***c*** è un carattere senza segno convertito in un intero, oppure l'equivalente di '**`EOF`**'. La funzione restituisce il carattere fornito come argomento, dopo la conversione in minuscolo, ammesso che ci possa essere una conversione.

Funzione	Descrizione
`int toupper (int c);`	L'argomento rappresentato dal parametro ***c*** è un carattere senza segno convertito in un intero, oppure l'equivalente di '**`EOF`**'. La funzione restituisce il carattere fornito come argomento, dopo la conversione in maiuscolo, ammesso che ci possa essere una conversione.

File «stdlib.h»

Funzione	Descrizione
`int atoi (const char *s);` `long int atol (const char *s);` `long long int atoll (const char *s);` `double atof (const char *s);`	Converte la stringa fornita come argomento in un numero intero o in virgola mobile, come si vede dal modello sintattico.

Funzione	Descrizione
`float strtof (const char *restrict s, char **restrict p);` `double strtod (const char *restrict s, char **restrict p);` `long double strtold (const char *restrict s, char **restrict p);`	Converte la stringa fornita come primo argomento in un numero in virgola mobile, come si vede dal modello sintattico. La conversione avviene fino a dove è possibile riconoscere caratteri che compongono un valore valido; se il secondo argomento è un puntatore a un puntatore valido (un puntatore a un'area di memoria che può contenere a sua volta un puntatore dal tipo '**char**'), al suo interno viene memorizzato l'indirizzo finale della scansione, a partire dal quale si trovano caratteri non decifrabili, oppure dove si trova il carattere nullo di terminazione della stringa.

Funzione	Descrizione
`long int strtol (const char *restrict s, char **restrict p, int base);` `long long int strtoll (const char *restrict s, char **restrict p, int base);` `unsigned long int strtoul (const char *restrict s, char **restrict p, int base);` `unsigned long long int strtoull (const char *restrict s, char **restrict p, int base);`	Converte la stringa fornita come primo argomento in un numero intero, come si vede dal modello sintattico, interpretando la stringa come numero espresso nella base rappresentata dal parametro ***base***. La conversione avviene fino a dove è possibile riconoscere caratteri che compongono un valore valido; se il secondo argomento è un puntatore a un puntatore valido (un puntatore a un'area di memoria che può contenere a sua volta un puntatore dal tipo '**`char`**'), al suo interno viene memorizzato l'indirizzo finale della scansione, a partire dal quale si trovano caratteri non decifrabili, oppure dove si trova il carattere nullo di terminazione della stringa.
`void srand (unsigned int seed);`	Modifica il seme per la generazione di numeri casuali attraverso la funzione ***rand()***.

Funzione	Descrizione
`int rand (void);`	Restituisce il numero casuale successivo.
`void *malloc (size_t` ***size*** `);`	Richiede l'allocazione di memoria di almeno ***size*** byte, restituendo il puntatore all'inizio della stessa se l'operazione ha successo, oppure il puntatore nullo se l'allocazione fallisce.
`void *calloc (size_t` ***nmemb*** `,` `size_t` ***size*** `);`	Richiede l'allocazione di memoria di almeno ***nmemb*** elementi da ***size*** byte ciascuno, restituendo il puntatore all'inizio della stessa se l'operazione ha successo, oppure il puntatore nullo se l'allocazione fallisce.

Funzione	Descrizione
`void *realloc (void *`***ptr***`, size_t` ***size***`);`	Richiede la riallocazione della memoria già allocata precedentemente a partire dall'indirizzo ***ptr***, in modo da avere almeno ***size*** byte, recuperando il contenuto precedente, per ciò che è possibile. La riallocazione può avvenire in corrispondenza di un indirizzo differente da quello originale, ma può fallire, restituendo così solo il puntatore nullo.
`void free (void *`***ptr***`);`	Libera la memoria allocata precedente a partire dall'indirizzo ***ptr***.
`int atexit (void (*`***func***`) (void));`	Accumula in un elenco il puntatore a una funzione che non richiede argomenti, da eseguire, assieme alle altre dell'elenco, quando viene chiamata la funzione ***exit()***. La funzione ***atexit()*** restituisce un valore numerico da intendere come *Vero* o *Falso*, a indicare se l'operazione ha avuto successo o meno.

Funzione	Descrizione
`void exit (int` ***status*** `);`	Conclude il funzionamento del programma, ma prima esegue le funzioni accumulate con l'ausilio di ***atexit()***, quindi chiude i file e infine passa il valore ricevuto come argomento in modo tale che sia restituito dal programma stesso.
`void _Exit (int` ***status*** `);`	Conclude il funzionamento del programma in modo brutale, senza occuparsi di nulla, a parte il far sì che il programma restituisca il valore indicato come argomento.
`void abort (void);`	Produce l'emissione del segnale '**`SIGABRT`**' (*abort*) che porta alla morte del processo elaborativo.
`char *getenv (const char *`***name***`);`	Restituisce il puntatore all'inizio della stringa che rappresenta il contenuto della variabile di ambiente indicata per nome, come argomento.

Funzione	Descrizione
`int system (const char *`***string***`);`	Esegue il comando indicato come argomento, attraverso il sistema operativo, restituendo il valore di uscita del comando stesso.
`void qsort (void *`***base***`,` `size_t` ***nmemb***`,` `size_t` ***size***`,` `int (*`***compar***`)` `(const void *,` `const void *));`	Riordina un array che inizia dall'indirizzo ***base***, essendo composto da ***size*** elementi da ***nmemb*** byte ognuno, utilizzando per il confronto la funzione ***compar***.
`void *bsearch (const void *`***key***`,` `const void *`***base***`,` `size_t` ***nmemb***`,` `size_t` ***size***`,` `int (*`***compar***`)` `(const void *,` `const void *));`	Scandisce un array che inizia dall'indirizzo ***base***, essendo composto da ***size*** elementi da ***nmemb*** byte ognuno, il quale risulta già ordinato secondo la funzione ***compar***, alla ricerca della corrispondenza con il valore ****key***, la cui dimensione deve essere sempre di ***nmemb*** byte.
`int abs (int` ***j***`);` `long int labs (long int` ***j***`);` `long long int llabs` `(long long int` ***j***`);`	Restituisce il valore assoluto di ***j***.

Funzione	Descrizione
`div_t div (int` ***numeratore*****`,`** `int` ***denominatore*****`);`** `ldiv_t ldiv (long int` ***numeratore*****`,`** `long int` ***denominatore*****`);`** `lldiv_t lldiv` `(long long int` ***numeratore*****`,`** `long long int` ***denominatore*****`);`**	Restituisce il risultato della divisione dei due argomenti, in una struttura contenente il risultato intero e il resto della divisione.
`int mblen (const char *`***s***`, size_t` ***n***`);`	Restituisce la lunghezza in byte del primo carattere multibyte contenuto nella stringa fornita come primo argomento. La scansione termina comunque se raggiunge la quantità di byte indicata dal secondo argomento. Se la stringa contiene una sequenza multibyte errata o incompleta, il valore restituito è −1. Se al posto della stringa multibyte si fornisce il puntatore nullo, la funzione restituisce il valore uno o zero, a seconda che sia prevista o meno una codifica multibyte con una gestione dello stato (*shift state*).

Funzione	Descrizione
`int mbtowc (wchar_t *restrict` ***pwc*****`,`** `const char *restrict` ***s*****`,`** `size_t` ***n*****`);`**	Converte il carattere multibyte contenuto nella stringa ***s***, per un massimo di ***n*** byte, nel carattere esteso a cui punta ***pwc***, restituendo la quantità di byte utilizzati dalla stringa di origine, oppure −1 se si presentano errori. Se al posto della stringa ***s*** si mette il puntatore nullo, si ottiene un valore pari a uno o zero, a seconda che sia prevista o meno una codifica multibyte con una gestione dello stato (*shift state*).

Funzione	Descrizione
`int wctomb (char *s, wchar_t wc;`	Converte il carattere esteso ***wc*** in una sequenza multibyte che va a essere contenuta nella stringa ***s***, restituendo la quantità di byte prodotti, oppure −1 se si presentano errori. Se al posto della stringa ***s*** si mette il puntatore nullo, si ottiene un valore pari a uno o zero, a seconda che sia prevista o meno una codifica multibyte con una gestione dello stato (*shift state*).

Funzione	Descrizione
`size_t mbstowcs (wchar_t *restrict` ***pwcs*****`,`** `const char *restrict` ***s*****`,`** `size_t` ***n*****`);`**	Converte la stringa multibyte ***s*** nella stringa estesa ***pwcs***, producendo al massimo ***n*** caratteri estesi nella destinazione (incluso il carattere nullo di terminazione). La funzione restituisce la quantità di caratteri estesi copiati nella destinazione, escludendo il carattere nullo di terminazione, oppure l'equivalente di -1 in caso di errori. Se al posto della destinazione viene messo il puntatore nullo, l'operazione viene simulata ignorando il valore di ***n*** e senza memorizzare il risultato; pertanto è utile per contare lo spazio necessario nella destinazione.

Funzione	Descrizione
`size_t wcstombs (char *restrict s, wchar_t *restrict pwcs, size_t n);`	Converte la stringa estesa ***pwcs*** nella stringa multibyte ***s***, producendo al massimo ***n*** byte nella destinazione (incluso il carattere nullo di terminazione). La funzione restituisce la quantità di byte copiati nella destinazione, escludendo il carattere nullo di terminazione, oppure l'equivalente di -1 in caso di errori. Se al posto della destinazione viene messo il puntatore nullo, l'operazione viene simulata ignorando il valore di ***n*** e senza memorizzare il risultato; pertanto è utile per contare lo spazio necessario nella destinazione.

File «string.h»

«

Funzione	Descrizione
`void *memcpy (void *restrict dst, const void *restrict org, size_t n);`	Copia ***n*** caratteri a partire dall'indirizzo indicato da ***org***, per riprodurli a partire dall'indirizzo ***dst***, alla condizione che i due insiemi non risultino sovrapposti. La funzione restituisce l'indirizzo ***dst***.
`void *memmove (void *dst, const void *org, size_t n);`	Copia ***n*** caratteri a partire dall'indirizzo indicato da ***org***, per riprodurli a partire dall'indirizzo ***dst***, senza il vincolo che gli insiemi siano disgiunti. La funzione restituisce l'indirizzo ***dst***.
`char *strcpy (char *restrict dst, const char *restrict org);`	Copia la stringa ***org*** nell'array a cui punta ***dst***, includendo anche il carattere nullo di conclusione delle stringhe, alla condizione che le due stringhe non si sovrappongano. La funzione restituisce ***dst***.

Funzione	Descrizione
`char *strncpy (char *restrict dst, const char *restrict org, size_t n);`	Copia ***n*** caratteri della stringa ***org*** nell'array a cui punta ***dst***, contando tra i caratteri copiati anche il carattere nullo di conclusione delle stringhe, alla condizione che le due stringhe non si sovrappongano. Se la stringa di origine è più corta di ***n***, i caratteri mancanti sono rimpiazzati dal carattere nullo di conclusione delle stringhe. La funzione restituisce ***dst***.
`char *strcat (char *restrict dst, const char *restrict org);`	Copia la stringa ***org*** a partire dalla fine della stringa ***dst*** (sovrascrivendo il carattere nullo preesistente), alla condizione che le due stringhe non siano sovrapposte. La funzione restituisce ***dst***.

Funzione	Descrizione
`char *strncat (char *restrict dst, const char *restrict org, size_t n);`	Copia al massimo ***n*** caratteri della stringa ***org*** a partire dalla fine della stringa ***dst*** (sovrascrivendo il carattere nullo preesistente), aggiungendo alla fine il carattere nullo di terminazione, il tutto alla condizione che le due stringhe non siano sovrapposte. La funzione restituisce ***dst***.
`int memcmp (const void *s1, const void *s2, size_t n);`	Confronta i primi ***n*** caratteri delle aree di memoria a cui puntano ***s1*** e ***s2***, restituendo: un valore pari a zero se le due sequenze si equivalgono; un valore maggiore di zero se la sequenza di ***s1*** è maggiore di ***s2***; un valore minore di zero se la sequenza di ***s1*** è minore di ***s2***.
`int strcmp (const char *s1, const char *s2);`	Confronta due stringhe restituendo: un valore pari a zero se sono uguali; un valore maggiore di zero se la stringa ***s1*** è maggiore di ***s2***; un valore minore di zero se la stringa ***s1*** è minore di ***s2***.

Funzione	Descrizione
`int strcoll (const char *s1, const char *s2);`	La funzione ***strcoll()*** è analoga a ***strcmp()***, con la differenza che la comparazione avviene sulla base della configurazione locale (la categoria '**`LC_COLLATE`**'). Nel caso della configurazione locale '**`C`**' la funzione si comporta esattamente come ***strcmp()***.
`int strncmp (const char *s1, char *s2, size_t n);`	La funzione ***strncmp()*** si comporta in modo analogo a ***strcmp()***, con la differenza che la comparazione si arresta al massimo dopo ***n*** caratteri.

Funzione	Descrizione
`size_t strxfrm (char *restrict dst, const char *restrict org, size_t n);`	La funzione ***strxfrm()*** trasforma la stringa ***org*** sovrascrivendo la stringa ***dst*** in modo relativo alla configurazione locale. In pratica, la stringa trasformata che si ottiene può essere comparata con un'altra stringa trasformata nello stesso modo attraverso la funzione ***strcmp()*** ottenendo lo stesso esito che si avrebbe confrontando le stringhe originali con la funzione ***strcoll()***.
`void *memchr (const void *s, int c, size_t n);`	Cerca un carattere a partire da una certa posizione in memoria, scandendo al massimo una quantità determinata di caratteri, restituendo il puntatore al carattere trovato. Se nell'ambito specificato non trova il carattere, restituisce il puntatore nullo.

Funzione	Descrizione
`char *strchr (const char *s, int c);`	Cerca un carattere all'interno di una stringa, restituendo il puntatore al carattere trovato, oppure il puntatore nullo se la ricerca fallisce. Nella scansione viene preso in considerazione anche il carattere nullo di terminazione della stringa.
`char *strrchr (const char *s, int c);`	Cerca un carattere all'interno di una stringa, restituendo il puntatore all'ultimo carattere corrispondente trovato, oppure il puntatore nullo se la ricerca fallisce. Nella scansione viene preso in considerazione anche il carattere nullo di terminazione della stringa.
`size_t strspn (const char *s, const char *accept);`	Calcola la lunghezza massima iniziale della stringa ***s***, composta esclusivamente da caratteri contenuti nella stringa ***accept***, restituendo tale valore.

Funzione	Descrizione
`size_t strcspn (const char *s, const char *reject);`	Calcola la lunghezza massima iniziale della stringa ***s***, composta esclusivamente da caratteri differenti da quelli contenuti nella stringa ***reject***.
`char *strpbrk (const char *s, const char *accept);`	Scandisce la stringa ***s*** alla ricerca del primo carattere che risulti contenuto nella stringa ***accept***, restituendo il puntatore al carattere trovato, oppure, in mancanza di alcuna corrispondenza, il puntatore nullo.
`char *strstr (const char *string, const char *substring);`	Cerca la stringa ***substring*** nella stringa ***string*** restituendo il puntatore alla prima corrispondenza trovata (nella stringa ***string***). Se la corrispondenza non c'è, la funzione restituisce il puntatore nullo.

Funzione	Descrizione
`char *strtok (char *restrict string, const char *restrict delim);`	Serve a suddividere una stringa in unità, definite *token*, specificando un elenco di caratteri da intendere come delimitatori, in una seconda stringa. La funzione va usata in fasi successive, fornendo solo inizialmente la stringa da suddividere che continua poi a essere utilizzata se al suo posto viene fornito il puntatore nullo. La funzione restituisce, di volta in volta, il puntatore alla sottostringa contenente l'unità individuata, oppure il puntatore nullo, se non può trovarla.
`void *memset (void *s, int c, size_t n);`	Inizializza una certa area di memoria, a partire dall'indirizzo ***s***, con la ripetizione del carattere ***c***, tradotto in un carattere senza segno, copiandolo per ***n*** volte. La funzione restituisce ***s***.
`char *strerror (int errnum);`	Trasforma un numero nella descrizione del tipo di errore corrispondente.

Funzione	Descrizione
`size_t strlen (const char *s);`	Calcola la lunghezza di una stringa, escludendo dal conteggio il carattere nullo di terminazione.

File «time.h»

Funzione	Descrizione
`clock_t clock (void);`	Restituisce il tempo di CPU espresso in unità '**`clock_t`**', utilizzato dal processo elaborativo a partire dall'avvio del programma. Se la funzione non è in grado di dare questa indicazione, allora restituisce il valore −1, o più precisamente '**`(clock_t) (-1)`**'.
`time_t time (time_t *timer);`	Determina il tempo attuale secondo il calendario del sistema operativo, restituendolo nella forma del tipo '**`time_t`**'. Se il puntatore di tipo '**`time_t *`**' è valido, la stessa informazione che viene restituita viene anche memorizzata nell'indirizzo indicato da tale puntatore.

Funzione	Descrizione
`double difftime (time_t` ***time1*****`,`** `time_t` ***time0***`);`	Calcola la differenza tra due date, espresse in forma '**`time_t`**', restituendo l'intervallo in secondi.

Funzione	Descrizione
`time_t mktime (struct tm *`***timeptr***`);`	Riceve come argomento il puntatore a una variabile strutturata di tipo '**`struct tm`**', contenente le informazioni sull'ora locale, e determina il valore di quella data secondo la rappresentazione interna, di tipo '**`time_t`**'. La funzione non tiene conto del giorno della settimana e del giorno dell'anno; inoltre, ammette anche valori al di fuori degli intervalli stabiliti per i vari membri della struttura; infine, considera un valore negativo per il membro '***`timeptr`***`->tm_isdst`' come la richiesta di determinare se sia o meno in vigore l'ora estiva per la data indicata. Se la funzione non è in grado di restituire un valore rappresentabile nel tipo '**`time_t`**', o comunque se non può eseguire il suo compito, restituisce il valore −1, o più precisamente '**`(time_t) (-1)`**'. Se invece tutto procede regolarmente, la funzione provvede anche a correggere i valori dei vari membri della struttura e a ricalcolare il giorno della settimana e dell'anno.

Funzione	Descrizione
`struct tm *gmtime (const time_t *timer);`	Converte una data espressa nella forma del tipo '**`time_t`**', in una data suddivisa nella struttura '**`tm`**', relativa al tempo universale coordinato (UTC).
`struct tm *localtime (const time_t *timer);`	Converte una data espressa nella forma del tipo '**`time_t`**', in una data suddivisa nella struttura '**`tm`**', relativa all'ora locale.
`char *asctime (const struct tm *timeptr);`	Converte un'informazione data-orario, espressa nella forma di una struttura '**`struct tm`**', in una stringa che esprime l'ora locale, usando però una rappresentazione fissa in lingua inglese.
`char *ctime (const time_t *timer);`	Converte un'informazione data-orario, espressa nella forma del tipo '**`time_t`**', in una stringa che esprime l'ora locale, usando però una rappresentazione fissa in lingua inglese.

Funzione	Descrizione
`size_t strftime (char *restrict` ***s***`, size_t` ***maxsize***`, const char *restrict` ***format***`, const struct tm *restrict` ***timeptr***`);`	Interpreta il contenuto di una struttura di tipo '**`struct tm`**' e lo traduce in un testo, secondo una stringa di composizione libera. Il comportamento è affine a quello di ***printf()***, dove l'input è costituito dalla struttura contenente le informazioni data-orario.

Specificatore di conversione	Corrispondenza
`%C`	*century* Il secolo, ottenuto dividendo l'anno per 100 e ignorando i decimali.
`%y` `%Y`	*year* L'anno: nel primo caso si mostrano solo le ultime due cifre, mentre nel secondo si mostrano tutte.
`%b` | `%h` `%B`	Rispettivamente, il nome abbreviato e il nome per esteso del mese.
`%m`	*month* Il numero del mese, da 01 a 12.

Specificatore di conversione	Corrispondenza
`%d` `%e`	*day* Il giorno del mese, in forma numerica, da 1 a 31, utilizzando sempre due cifre: nel primo caso si aggiunge eventualmente uno zero; nel secondo si aggiunge eventualmente uno spazio.
`%a` `%A`	Rispettivamente, il nome abbreviato e il nome per esteso del giorno della settimana.
`%H` `%L`	*hour* L'ora, espressa rispettivamente a 24 ore e a 12 ore.
`%p`	La sigla da usare, secondo la configurazione locale, per specificare che si tratta di un'ora antimeridiana o pomeridiana. Nella convenzione inglese si ottengono, per esempio, le sigle «AM» e «PM».
`%r`	L'ora espressa a 12 ore, completa dell'indicazione se trattasi di ora antimeridiana o pomeridiana, secondo le convenzioni locali.
`%R`	L'ora e i minuti, equivalente a '**`%H:%M`**'.
`%M`	*minute* I minuti, da 00 a 59.
`%S`	*second* I secondi, espresso con valori da 00 a 60.
`%T`	*time* L'ora, i minuti e i secondi, equivalente a '**`%H:%M:%S`**'.

Specificatore di conversione	Corrispondenza
`%z` `%Z`	*time zone* La rappresentazione del fuso orario, nel primo caso come distanza dal tempo coordinato universale (UTC), mentre nel secondo si usa una rappresentazione conforme alla configurazione locale.
`%j`	*julian* Il giorno dell'anno, usando sempre tre cifre numeriche: da 001 a 366.
`%g` `%G`	L'anno a cui appartiene la settimana secondo lo standard ISO 8601: nel primo caso si mostrano solo le ultime due cifre, mentre nel secondo si ha l'anno per esteso. Secondo lo standard ISO 8601 la settimana inizia con lunedì e la prima settimana dell'anno è quella che include il 4 gennaio.
`%V`	Il numero della settimana secondo lo standard ISO 8601. I valori vanno da 01 a 53. Secondo lo standard ISO 8601 la settimana inizia con lunedì e la prima settimana dell'anno è quella che include il 4 gennaio.
`%u` `%w`	Il giorno della settimana, espresso in forma numerica, dove, rispettivamente, si conta da 1 a 7, oppure da 0 a 6. Zero e sette rappresentano la domenica; uno è il lunedì.
`%U` `%W`	Il numero della settimana, contando, rispettivamente, dalla prima domenica o dal primo lunedì di gennaio. Si ottengono cifre da 00 a 53.
`%x`	La data, rappresentata secondo le convenzioni locali.
`%X`	L'ora, rappresentata secondo le convenzioni locali.

Specificatore di conversione	Corrispondenza
`%c`	La data e l'ora, rappresentate secondo le convenzioni locali.
`%D`	*date* La data, rappresentata come '**`%m/%d/%Y`**'.
`%F`	La data, rappresentata come '**`%Y-%m-%d`**'.
`%n`	Viene rimpiazzato dal codice di interruzione di riga.
`%t`	Viene rimpiazzato da una tabulazione orizzontale.
`%%`	Viene rimpiazzato da un carattere di percentuale.

File «stdio.h» per la gestione dei file e degli errori

«

Macro-variabile	Significato mnemonico	Descrizione
`_IOFBF`	*input output fully buffered*	Indica simbolicamente la richiesta di utilizzo di una memoria tampone a blocchi.
`_IOLBF`	*input output line buffered*	Indica simbolicamente la richiesta di utilizzo di una memoria tampone gestita a righe di testo.
`_IONBF`	*input output with no buffering*	Indica simbolicamente la richiesta di non utilizzare alcuna memoria tampone.

Macro-variabile	Significato mnemonico	Descrizione
`BUFSIZE`	*buffer size*	Rappresenta la dimensione predefinita della memoria tampone.
`EOF`	*end of file*	È un numero intero di tipo '**`int`**', negativo, che rappresenta il raggiungimento della fine del file. È in pratica ciò che si ottiene leggendo oltre la fine del file.
`FOPEN_MAX`	*file open max*	Il numero di file che un processo elaborativo può aprire simultaneamente, in base alle limitazioni poste dal sistema operativo.
`FILENAME_MAX`		La dimensione di un array di elementi '**`char`**', tale da essere abbastanza grande da contenere il nome del file più lungo (incluse le eventuali sequenze multibyte) che il sistema consenta di gestire.
`L_tmpnam`	*temporary name*	La dimensione di un array di elementi '**`char`**', tale da essere abbastanza grande da contenere il nome di un file temporaneo generato dalla funzione ***tmpnam()***.
`SEEK_CUR`	*seek current*	Indica di eseguire un posizionamento a partire dalla posizione corrente del file.

Macro-variabile	Significato mnemonico	Descrizione
`SEEK_END`		Indica di eseguire un posizionamento a partire dalla fine di un file.
`SEEK_SET`		Indica di eseguire un posizionamento a partire dall'inizio di un file.
`TMP_MAX`		Rappresenta la quantità massima di nomi di file differenti che possono essere generati dalla funzione ***tmpnam()***.

Modalità di accesso ai file	Mnemonico	Descrizione
`r`	*read*	Accesso in sola lettura di un file di testo.
`w`	*write*	Accesso a un file di testo in scrittura, che implica la creazione del file all'apertura, ovvero il suo troncamento a zero, se esiste già.
`a`	*append*	Accesso a un file di testo in aggiunta, che implica la creazione del file all'apertura, ovvero la sua estensione se esiste già.
`rb` `wb` `ab`	*binary*	Accesso in lettura, scrittura o aggiunta, ma di tipo binario.

Modalità di accesso ai file	Mnemonico	Descrizione
`r+` `w+` `a+`	*update*	Accesso a un file di testo in lettura, scrittura o aggiunta, assieme alla modalità di aggiornamento. In pratica, con la lettura è consentita anche la scrittura; con la scrittura e l'aggiunta è consentita anche la rilettura.
`rb+` | `r+b` `wb+` | `w+b` `ab+` | `a+b`		Accesso a un file binario in lettura, scrittura o aggiunta, assieme alla modalità di aggiornamento. In pratica, con la lettura è consentita anche la scrittura; con la scrittura e l'aggiunta è consentita anche la rilettura. Si può osservare che il segno '**+**' può essere messo indifferentemente in mezzo o alla fine.

Funzione	Descrizione
`int remove (const char *`***filename***`);`	Cancella il file il cui nome viene fornito come argomento. Il nome del file va espresso secondo le convenzioni del sistema operativo. Restituisce zero se l'operazione ha successo, altrimenti un valore differente.

Funzione	Descrizione
`int rename (const char *old, const char *new);`	Cambia il nome del file indicato come primo argomento, in modo che assuma quello del secondo argomento. Se l'operazione avviene con successo restituisce zero, altrimenti produce un valore differente.
`FILE *tmpfile (void);`	Crea e apre un file temporaneo binario in aggiornamento ('**wb+**'), restituendone il puntatore. Se il file temporaneo non può essere creato, la funzione restituisce il puntatore nullo.
`char *tmpnam (char *s);`	Genera il nome di un file che può essere usato come file temporaneo. La funzione richiede come argomento un array di almeno '**L_tmpnam**' caratteri, da usare per scriverci il nome e per restituirne il puntatore. Se alla funzione viene passato il puntatore nullo, allora questa usa un'area di memoria statica che viene sovrascritta a ogni chiamata successiva della funzione stessa. Se la funzione non può eseguire il suo lavoro, restituisce il puntatore nullo.

Funzione	Descrizione
`FILE *fopen (const char *restrict` ***filename*****,** `const char *restrict` ***io_mode*****);**	Apre il file indicato come primo argomento, secondo la modalità espressa dalla stringa che costituisce il secondo argomento, restituendo il puntatore che ne rappresenta il flusso aperto. Se l'operazione fallisce la funzione restituisce il puntatore nullo e aggiorna il valore della variabile globale ***errno***.
`FILE *freopen (const char *restrict` ***filename*****,** `const char *restrict` ***io_mode*****,** `FILE *restrict` ***stream*****);**	Apre il file indicato come primo argomento, secondo la modalità espressa dalla stringa che costituisce il secondo argomento, utilizzando il flusso di file individuato dal puntatore che costituisce l'ultimo argomento, restituendo lo stesso puntatore. Se il puntatore indicato come ultimo argomento riguarda un flusso di file ancora aperto, questo viene chiuso e quindi riaperto. Se l'operazione fallisce la funzione restituisce il puntatore nullo e aggiorna il valore della variabile globale ***errno***. Lo scopo di questa funzione è quello di ridirigere i flussi di file, associando file differenti.

Funzione	Descrizione
`int fclose (FILE *`***stream***`);`	Chiude il flusso di file individuato dal puntatore che costituisce l'argomento della funzione. La funzione restituisce il valore zero se l'operazione ha successo, altrimenti produce il valore corrispondente alla macrovariabile ***EOF***. Va sottolineato che un flusso già chiuso non deve essere chiuso nuovamente, perché in tal caso l'effetto che se ne produce è imprecisato.

Funzione	Descrizione
`int setvbuf (FILE *restrict` ***stream*****,** `char *restrict` ***buffer*****,** `int` ***buf_mode*****,** `size_t` ***size*****);**	Attribuisce una memoria tampone a un file che è appena stato aperto e per il quale non è ancora stato eseguito alcun accesso. Il primo argomento della funzione è il puntatore al flusso relativo e il secondo è il puntatore all'inizio dell'array di caratteri da usare come memoria tampone. Se al posto del riferimento alla memoria tampone si indica un puntatore nullo, si intende che la funzione debba allocare automaticamente lo spazio necessario; se invece l'array viene fornito, è evidente che deve rimanere disponibile per tutto il tempo in cui il flusso rimane aperto. Il terzo argomento atteso dalla funzione è un numero che esprime la modalità di funzionamento della memoria tampone. Questo numero viene fornito attraverso l'indicazione di una tra le macro-variabili ***_IOFBF***, ***_IOLBF*** e ***_IONBF***. Il quarto argomento indica la dimensione dell'array da usare come memoria tampone: se l'array viene fornito effettivamente, si tratta della dimensione che può essere utilizzata; altrimenti è la dimensione richiesta per l'allocazione automatica. La funzione restituisce zero se l'operazione richiesta è eseguita con successo; diversamente restituisce un valore differente.

Funzione	Descrizione
`void setbuf (FILE *restrict` ***stream*****`,`** `char *restrict` ***buffer*****`);`	Si tratta di una versione semplificata di '**`setvbuf()`**' che non restituisce alcun valore, che prevede implicitamente una modalità di gestione completa della memoria tampone ('**`_IOFBF`**'), che richiede implicitamente un array di '**`BUFSIZ`**' elementi. Anche in questo caso, se l'argomento corrispondente al parametro ***buffer*** è un puntatore nullo, l'allocazione avviene in modo automatico.
`int fflush (FILE *`***stream***`);`	Scarica la memoria tampone del flusso di file indicato, procedendo così alla memorizzazione dei dati rimasti in sospeso. Restituisce zero se l'operazione viene completata con successo, altrimenti restituisce il valore corrispondente alla macro-variabile ***EOF***.
`int fgetc (FILE *`***stream***`);` `int getc (FILE *`***stream***`);`	Legge un carattere (senza segno) dal flusso di file indicato come argomento e ne restituisce il valore numerico (positivo). Se l'operazione fallisce, la funzione restituisce il valore corrispondente alla macro-variabile ***EOF***. Tradizionalmente, ***fgetc()*** è sempre una funzione, mentre ***getc()*** potrebbe essere una macroistruzione che valuta anche più volte l'espressione che costituisce l'argomento.

Funzione	Descrizione
`int ungetc (int c, FILE *stream);`	Rimanda indietro il carattere ***c*** nel flusso di file ***stream***; in altri termini dovrebbe annullare l'effetto dell'ultima chiamata a una funzione ***fgetc()*** o ***getc()***.
`int fputc (int c, FILE *stream);` `int putc (int c, FILE *stream);`	Scrive un carattere, rappresentato dal primo argomento, nel flusso di file indicato come secondo argomento, restituendo lo stesso valore del carattere scritto, se l'operazione si conclude con successo, oppure il valore corrispondente a '**EOF**' se l'operazione fallisce. Tradizionalmente, ***fputc()*** è sempre una funzione, mentre ***putc()*** potrebbe essere una macroistruzione che valuta anche più volte le espressioni che costituiscono gli argomenti.
`int putchar (int c);`	Scrive un carattere, rappresentato dall'argomento, nello standard output, restituendo lo stesso valore del carattere scritto, se l'operazione si conclude con successo, oppure il valore corrispondente a '**EOF**' se l'operazione fallisce. Tradizionalmente si tratta di una macroistruzione che valuta anche più volte l'espressione che costituisce l'argomento.

Funzione	Descrizione
`char *fgets (char *restrict s,` `int n,` `FILE *restrict stream);`	Legge al massimo $n-1$ caratteri (elementi '**`char`**') attraverso il flusso di file ***stream***, copiandoli in memoria a partire dall'indirizzo ***s*** e aggiungendo alla fine il carattere nullo di terminazione delle stringhe. La lettura si esaurisce prima di $n-1$ caratteri se viene incontrato il codice di interruzione di riga, il quale viene rappresentato nella stringa a cui punta ***s***, ovvero se si raggiunge la fine del file. In ogni caso, la stringa ***s*** viene terminata correttamente con il carattere nullo. La funzione restituisce la stringa ***s*** se la lettura avviene con successo, ovvero se ha prodotto almeno un carattere; altrimenti, il contenuto dell'array a cui punta ***s*** non viene modificato e la funzione restituisce il puntatore nullo. Se si creano errori imprevisti, la funzione potrebbe restituire il puntatore nullo, ma senza garantire che l'array ***s*** sia rimasto intatto.

Funzione	Descrizione
`int fputs (const char *restrict s, FILE *restrict stream);`	Copia la stringa a cui punta ***s*** nel file rappresentato dal flusso di file ***stream***. La copia della stringa avviene escludendo però il carattere nullo di terminazione. Va osservato che questa funzione, pur essendo contrapposta evidentemente a '**`fgets()`**', **non conclude la riga** del file, ovvero, non aggiunge il codice di interruzione di riga. Per ottenere la conclusione della riga di un file di testo, occorre inserire nella stringa, espressamente, il carattere '**`\n`**'. La funzione restituisce il valore rappresentato da '**`EOF`**' se l'operazione di scrittura produce un errore; altrimenti restituisce un valore positivo qualunque.
`char *gets (char *s);`	Legge una riga dallo standard input, copiandola in memoria a partire dall'indirizzo ***s*** e aggiungendo alla fine il carattere nullo di terminazione delle stringhe. Per il resto, il funzionamento è conforme a quello di ***fgets()***.
`int puts (const char *s);`	Copia la stringa a cui punta ***s*** nello standard output, aggiungendo in coda il codice di interruzione di riga. Per il resto, il funzionamento è analogo a quello di ***fputs()***.

Funzione	Descrizione
`size_t fread (void *restrict` ***ptr*****`,`** `size_t` ***size*****`,`** `size_t` ***nmemb*****`,`** `FILE *restrict` ***stream*****`);`**	Legge dal flusso di file ***stream***, ***nmemb*** blocchi da ***size*** byte, copiando questi dati in memoria a partire dall'indirizzo ***ptr***. Restituisce la quantità di blocchi da '**`size`**' byte che sono stati copiati con successo; pertanto, se questo valore è inferiore a ***nmemb***, si è verificato un problema.
`size_t fwrite` `(const void *restrict` ***ptr*****`,`** `size_t` ***size*****`,`** `size_t` ***nmemb*****`,`** `FILE *restrict` ***stream*****`);`**	Scrive nel flusso di file ***stream***, ***nmemb*** blocchi da ***size*** byte, leggendo questi dati dalla memoria a partire dall'indirizzo ***ptr***. Restituisce la quantità di blocchi da '**`size`**' byte che sono stati copiati con successo; pertanto, se questo valore è inferiore a ***nmemb***, si è verificato un problema.

Funzione	Descrizione
`int fseek (FILE *`***stream***`,` `long int` ***offset***`,` `int` ***whence***`);`	Sposta la posizione corrente relativa al flusso di file ***stream*** (associato preferibilmente a un file binario), nella nuova posizione determinata dai parametri ***whence*** e ***offset***. Il parametro ***whence*** viene fornito attraverso una macro-variabile che può essere ***SEEK_SET***, ***SEEK_CUR*** o ***SEEK_END***, indicando rispettivamente l'inizio del file, la posizione corrente o la fine del file. Dalla posizione indicata dal parametro ***whence*** viene aggiunta, algebricamente, la quantità di byte indicata dal parametro ***offset***. La funzione restituisce zero se può eseguire l'operazione, altrimenti dà un risultato diverso.
`long int ftell (FILE *`***stream***`);`	Restituisce la posizione corrente del flusso di file indicato come argomento. Questo valore può essere usato con ***fseek()***, al posto dello scostamento (il parametro ***offset***), indicando come posizione di riferimento l'inizio del file, ovvero '**`SEEK_SET`**'. Se la funzione non riesce a fornire la posizione, restituisce il valore −1 (tradotto in '**`long int`**') e annota il fatto nella variabile ***errno***.

Funzione	Descrizione
`void rewind (FILE *`***stream***`);`	Riposiziona il flusso di file all'inizio. In pratica è come utilizzare la funzione ***fseek()*** specificando uno scostamento pari a zero a partire da '**`SEEK_SET`**', ignorando il valore restituito.
`int fgetpos (FILE *restrict` ***stream***`,` `fpos_t *restrict` ***pos***`);`	Memorizza nella variabile a cui punta il parametro ***pos*** le informazioni sulla posizione corrente del file, assieme allo stato di interpretazione relativo alle sequenze multibyte. Restituisce zero se l'operazione è stata compiuta con successo, altrimenti dà un altro valore
`int fsetpos (FILE *`***stream***`,` `const fpos_t *`***pos***`);`	Utilizza la variabile a cui punta ***pos*** per ripristinare la posizione memorizzata, assieme allo stato di avanzamento dell'interpretazione di una sequenza multibyte. Restituisce zero se l'operazione è stata compiuta con successo, altrimenti dà un altro valore e aggiorna la variabile ***errno***.
`void clearerr (FILE *`***stream***`);`	Azzera gli indicatori di errore e di fine file per il flusso di file indicato come argomento, senza restituire alcunché.
`int feof (FILE *`***stream***`);`	Controlla lo stato dell'indicatore di fine file per il flusso di file indicato. Se questo non è attivo restituisce zero, altrimenti restituisce un valore diverso da zero.

Funzione	Descrizione
`int ferror (FILE *`***stream***`);`	Controlla lo stato dell'indicatore di errore per il flusso di file indicato. Se questo non è attivo restituisce zero, altrimenti restituisce un valore diverso da zero.
`void perror (const char *`***s***`);`	Prende in considerazione la variabile ***errno*** e cerca di tradurla in un messaggio testuale da emettere attraverso lo standard error (con tanto di terminazione della riga, in modo da riposizionare a capo il cursore). Se il parametro ***s*** corrisponde a una stringa non vuota, il testo di questa viene posto anteriormente al messaggio, separandolo con due punti e uno spazio (‘: ’). Il contenuto del messaggio è lo stesso che si otterrebbe con la funzione ***strerror()***, fornendo come argomento la variabile ***errno***.

File «stdio.h» per la composizione dell'output

«

`%` [***simbolo***] [***n_ampiezza***] [***. n_precisione***] [`hh` | `h` | `l` | `ll` | `j` | `z` | `t` | `L`] ***tipo***

Simbolo	Tipo di argomento	Conversione applicata
`%…d` `%…i`	`int`	Numero intero con segno da rappresentare in base dieci.

Simbolo	Tipo di argomento	Conversione applicata
`%…u`	`unsigned int`	Numero intero senza segno da rappresentare in base dieci.
`%…o`	`unsigned int`	Numero intero senza segno da rappresentare in ottale (senza lo zero iniziale che viene usato spesso per caratterizzare un tale tipo di rappresentazione).
`%…x` `%…X`	`unsigned int`	Numero intero senza segno da rappresentare in esadecimale (senza il prefisso '**`0x`**' o '**`0X`**' che viene usato spesso per caratterizzare un tale tipo di rappresentazione).
`%…c`	`int`	Un carattere singolo, dopo la conversione in '**`unsigned char`**'.
`%…s`	`char *`	Una stringa.
`%…f`	`double`	Un numero a virgola mobile, da rappresentare in notazione decimale fissa: ***[-] iii . dddddd***
`%…e` `%…E`	`double`	Un numero a virgola mobile, da rappresentare in notazione esponenziale: ***[-] i . dddddd*`e`*$\pm$***xx*** ***[-] i . dddddd*`E`*$\pm$***xx***

Simbolo	Tipo di argomento	Conversione applicata
`%…g` `%…G`	`double`	Un numero a virgola mobile, rappresentato in notazione decimale fissa o in notazione esponenziale, a seconda di quale si presti meglio in base ai vincoli posti da altri componenti dello specificatore di conversione.
`%p`	`void *`	Un puntatore generico rappresentato in qualche modo in forma grafica.
`%n`	`int *`	Questo specificatore non esegue alcuna conversione e si limita a memorizzare un valore intero (di tipo '**`int`**') nella variabile a cui punta l'argomento. Per la precisione, viene memorizzata la quantità di caratteri generati fino a quel punto dalla conversione.
`%%`		Questo specificatore si limita a produrre un carattere di percentuale ('**`%`**') che altrimenti non sarebbe rappresentabile.

Simbolo	Corrispondenza
`%+…` `%#+…` `%+0`***ampiezza***`…` `%#+0`***ampiezza***`…`	Il segno «+» fa sì che i numeri con segno lo mostrino anche se è positivo. Può combinarsi con lo zero e il cancelletto.

Simbolo	Corrispondenza
%0***ampiezza***... %+0***ampiezza***... %#0***ampiezza***... %#+0***ampiezza***...	Lo zero fa sì che siano inseriti degli zeri a sinistra per allineare a destra il valore, nell'ambito dell'ampiezza specificata. Può combinarsi con il segno «+» e il cancelletto.
%***ampiezza***... % ***ampiezza***...	In mancanza di uno zero iniziale, in presenza dell'indicazione dell'ampiezza, il valore viene allineato a destra usando degli spazi. È possibile esprimere esplicitamente l'intenzione di usare gli spazi mettendo proprio uno spazio, ma in generale non è richiesto. Se si mette lo spazio letteralmente, questo non è poi compatibile con lo zero, mentre le combinazioni con gli altri simboli sono ammissibili.
%−***ampiezza***... %−+***ampiezza***... %#−***ampiezza***... %#−+***ampiezza***...	Il segno meno, usato quando la conversione prevede l'uso di una quantità fissa di caratteri con un valore che appare di norma allineato a destra, fa sì che il risultato sia allineato a sinistra. Il segno meno si può combinare il segno «+» e il cancelletto.

Simbolo	Corrispondenza
`%#…`	Il cancelletto richiede una modalità di rappresentazione alternativa, ammesso che questa sia prevista per il tipo di conversione specificato. È compatibili con gli altri simboli, ammesso che il suo utilizzo serva effettivamente per ottenere una rappresentazione alternativa.

Simbolo	Tipo	Simbolo	Tipo
`%…hhd` `%…hhi`	`signed char`	`%…hhu` `%…hho` `%…hhx` | `%…hhX`	`unsigned char`
`%…hd` `%…hi`	`short int`	`%…hu` `%…ho` `%…hx` | `%…hX`	`unsigned short int`
`%…ld` `%…li`	`long int`	`%…lu` `%…lo` `%…lx` | `%…lX`	`unsigned long int`
`%…lc`	`wint_t`	`%…ls`	`wchar_t *`

Simbolo	Tipo	Simbolo	Tipo
%…lld %…lli	long long int	%…llu %…llo %…llx| %…llX	unsigned long long int
%…jd %…ji	intmax_t	%…ju %…jo %…jx|%…jX	uintmax_t
%…zd %…zi	size_t	%…zu %…zo %…zx|%…zX	size_t
%…td %…ti	ptrdiff_t	%…tu %…to %…tx|%…tX	ptrdiff_t
%…Le|%…LE %…Lf|%…LF %…Lg|%…LG	long double		

Funzione	Descrizione
`int sprintf (char *restrict s, const char *restrict format, ...);`	Produce il risultato della composizione memorizzandolo a partire dal puntatore costituito dal primo parametro (*s*) e aggiungendo il carattere nullo di terminazione.
`int snprintf (char *restrict s, size_t n, const char *restrict format, ...);`	Produce al massimo $n-1$ caratteri, aggiungendo sempre il carattere nullo di terminazione.
`int fprintf (FILE *restrict stream, const char *restrict format, ...);`	Scrive il risultato della composizione attraverso il flusso di file *stream*.
`int printf (const char *restrict format, ...);`	Scrive il risultato della composizione attraverso lo standard output.
`int vsprintf (char *restrict s, const char *restrict format, va_list arg);`	Come la funzione *sprintf()*, ricevendo gli argomenti variabili attraverso un puntatore al loro inizio.

Funzione	Descrizione
`int vsnprintf (char *restrict` ***s***`, size_t` ***n***`, const char *restrict` ***format***`, va_list` ***arg***`);`	Come la funzione ***snprintf()***, ricevendo gli argomenti variabili attraverso un puntatore al loro inizio.
`int vfprintf (FILE *restrict` ***stream***`, const char *restrict` ***format***`, va_list` ***arg***`);`	Come la funzione ***fprintf()***, ricevendo gli argomenti variabili attraverso un puntatore al loro inizio.
`int vprintf (const char *restrict` ***format***`, va_list` ***arg***`);`	Come la funzione ***printf()***, ricevendo gli argomenti variabili attraverso un puntatore al loro inizio.

File «stdio.h» per l'interpretazione dell'input

«

`%[*] [`***n_ampiezza***`] [hh|h|l|ll|j|z|t|L]` ***tipo***

Simbolo	Tipo di argomento	Conversione applicata
`%…d`	`int *`	Numero intero con segno rappresentato in base dieci.
`%…i`	`int *`	Numero intero con segno rappresentare in base dieci o in base otto, avendo come prefisso uno zero, oppure in base sedici, avendo come prefisso '**`0x`**' o '**`0X`**'.
`%…u`	`unsigned int *`	Numero intero senza segno rappresentato in base dieci.

Simbolo	Tipo di argomento	Conversione applicata
`%…o`	`unsigned int *`	Numero intero senza segno rappresentato in ottale (con o senza lo zero iniziale).
`%…x`	`unsigned int *`	Numero intero senza segno rappresentato in esadecimale (con o senza il prefisso '**`0x`**' o '**`0X`**').
`%…c`	`char *`	Interpreta un solo carattere, o più caratteri se si specifica l'ampiezza. Nella lettura contano anche gli spazi o qualunque altro carattere e non viene aggiunto il carattere nullo di terminazione.
`%…s`	`char *`	Interpreta una sequenza di caratteri che non siano spazi, aggiungendo alla fine il carattere nullo di terminazione.
`%…a` `%…e` `%…f` `%…g`	`double *`	Un numero a virgola mobile rappresentato in notazione decimale fissa o in notazione esponenziale: **[**`-`**]** ***iii . dddddd*** **[**`-`**]** ***i . dddddd*****`e`**$\pm$***xx*** **[**`-`**]** ***i . dddddd*****`E`**$\pm$***xx***
`%p`	`void *`	Interpreta il valore di un puntatore che sia rappresentato nello stesso modo in cui farebbe la funzione '**`printf("%p",`** ***puntatore*****`)`**'.

Simbolo	Tipo di argomento	Conversione applicata
`%n`	`int *`	Questo specificatore non esegue alcuna conversione e si limita a memorizzare la quantità di caratteri (‘**`char`**’) letti fino a quel punto.
`%…[…]`	`char *`	Interpreta una stringa non vuota contenente solo i caratteri elencati tra parentesi quadre, aggiungendo alla fine il carattere nullo di terminazione. Se tra i caratteri si cerca anche la parentesi quadra chiusa, questa va messa all’inizio dell’elenco: ‘**`%…[]…]`**’.
`%…[^…]`	`char *`	Interpreta una stringa non vuota contenente solo caratteri diversi da quelli elencati tra parentesi quadre, aggiungendo alla fine il carattere nullo di terminazione. Se tra i caratteri da escludere si vuole indicare anche la parentesi quadra chiusa, questa va messa all’inizio dell’elenco: ‘**`%…[^]…]`**’.
`%%`		Interpreta un carattere di percentuale tra i dati in ingresso, ma senza memorizzare alcunché.

Simbolo	Tipo	Simbolo	Tipo
%…hhd %…hhi	signed char *	%…hhu %…hho %…hhx %…hhn	unsigned char *
%…hd %…hi	short int *	%…hu %…ho %…hx %…hn	unsigned short int *
%…ld %…li	long int *	%…lu %…lo %…lx %…ln	unsigned long int *
		%…lc %…ls %…lc %…l[…]	wchar_t *

Simbolo	Tipo	Simbolo	Tipo
%…lld %…lli	long long int *	%…llu %…llo %…llx %…lln	unsigned long long int *
%…jd %…ji	intmax_t *	%…ju %…jo %…jx %…jn	uintmax_t *
%…zd %…zi	size_t *	%…zu %…zo %…zx %…zn	size_t *
%…td %…ti	ptrdiff_t *	%…tu %…to %…tx %…tn	ptrdiff_t *

Simbolo	Tipo	Simbolo	Tipo
`%…Le` `%…Lf` `%…Lg`	`long double *`		

Funzione	Descrizione
`int fscanf` `(FILE *restrict` ***stream*****,** `const char *restrict` ***format*****,** `...);`	Scandisce l'input proveniente dal flusso_di_file che costituisce il primo parametro (***stream***), restituendo la quantità di valori assegnati alle variabili rispettive, oppure il valore corrispondente alla macro-variabile ***EOF*** nel caso si verifichi un errore prima di qualunque conversione.
`int sscanf` `(const char *restrict` ***s*****,** `const char *restrict` ***format*****,** `...);`	Scandisce il contenuto della stringa indicata come primo parametro (***s***), restituendo la quantità di valori assegnati alle variabili rispettive, oppure il valore corrispondente alla macro-variabile ***EOF*** nel caso si verifichi un errore prima di qualunque conversione.

Funzione	Descrizione
`int scanf (const char *restrict` ***format*****`,`** `...);`	Scandisce lo standard input, restituendo la quantità di valori assegnati alle variabili rispettive, oppure il valore corrispondente alla macro-variabile ***EOF*** nel caso si verifichi un errore prima di qualunque conversione.
`int vfscanf (FILE *restrict` ***stream*****`,`** `const char *restrict` ***format*****`,`** `va_list` ***arg*****`);`**	Come ***fscanf()***, con la differenza che gli argomenti variabili sono sostituiti da un puntatore al loro inizio.
`int vsscanf (const char *restrict` ***s*****`,`** `const char *restrict` ***format*****`,`** `va_list` ***arg*****`);`**	Come ***sscanf()***, con la differenza che gli argomenti variabili sono sostituiti da un puntatore al loro inizio.
`int vscanf (const char *restrict` ***format*****`,`** `va_list` ***arg*****`);`**	Come ***scanf()***, con la differenza che gli argomenti variabili sono sostituiti da un puntatore al loro inizio.

File «assert.h»

Macroistruzione	Descrizione
`void assert (`***tipo_scalare espressione***`);`	Nell'uso di questa macroistruzione, in pratica si scrive solo l'espressione tra parentesi, senza indicare espressamente il tipo scalare. Se l'espressione si traduce in un valore pari a zero, l'asserzione fallisce e viene mostrato un messaggio di errore, con le informazioni necessarie per risalire alla posizione nel file sorgente.

File «stddef.h»

Macroistruzione	Descrizione
`size_t offsetof (`***type, member***`);`	Restituisce lo scostamento che separa un membro di una struttura dall'inizio della stessa.

«

Funzione	Descrizione
`char *setlocale (int` ***category***`, NULL);`	Restituisce un puntatore alla stringa che descrive la configurazione locale corrente, riferita alla categoria specificata. Se non è in grado di fornire l'informazione, fornisce un puntatore nullo.
`char *setlocale (int` ***category***`,` `const char *`***locale***`);`	Imposta la configurazione locale, secondo il contenuto della stringa che costituisce il secondo parametro. La funzione restituisce un puntatore che descrive la stessa configurazione, oppure, se l'operazione fallisce, restituisce il puntatore nullo.

Macro-variabile	Descrizione
`LC_ALL`	Individua simultaneamente tutte le categorie relative alla configurazione locale.
`LC_COLLATE`	Categoria che definisce l'ordine alfabetico dei caratteri tipografici.
`LC_CTYPE`	Categoria che definisce il modo di raggruppare i caratteri tipografici per tipologia.
`LC_MONETARY`	Categoria che definisce le convenzioni legate alla rappresentazione dei valori che esprimono importi in valuta.

Macro-variabile	Descrizione
`LC_NUMERIC`	Categoria che definisce il modo in cui vanno rappresentati i valori numerici, soprattutto per quanto riguarda la separazione tra parte intera e parte decimale.
`LC_TIME`	Categoria che definisce il modo corretto di esprimere le informazioni data-orario.
`LC_MESSAGES`	POSIX: Categoria che definisce il formato dei messaggi emessi per informazioni generali e di quelli diagnostici.

File «regex.h»

Funzione	Descrizione
`int regcomp (regex_t *restrict` ***re***`, const char *restrict` ***regex***`, int` ***cflags***`);`	Compila l'espressione regolare descritta dalla stringa ***regex***, componendo il contenuto della variabile strutturata ***re***, tenendo conto delle opzioni ***cflags***.
`void regfree (regex_t *`***re***`);`	Libera la memoria associata all'espressione regolare compilata nella variabile ***re***.

Funzione	Descrizione
`int regexec (const regex_t *restrict` ***re***`, const char *restrict` ***s***`, size_t` ***n***`, regmatch_t` ***m***`[restrict], int` ***eflags***`);`	Compara l'espressione regolare ***re*** con la stringa ***s***, tenendo conto delle opzioni ***eflags***, immettendo le sottostringhe estratte nell'array di stringhe ***m***, sapendo che questo può contenere al massimo ***n*** stringhe.
`size_t regerror (int` ***e***`, const regex_t *restrict` ***re***`, char *restrict` ***t***`, size_t` ***n***`);`	Sulla base del numero di errore ***e*** e dell'espressione regolare ***re***, produce un messaggio di errore nella stringa ***t*** che non può essere più lunga di ***n*** caratteri.

Tipo	Nome	Membri noti del tipo '**`regex_t`**'
`size_t`	`re_sub`	Quantità di sottoespressioni tra parentesi tonde.

Tipo	Nome	Membri noti di una variabile di tipo '**`regmatch_t`**'
`regoff_t`	`rm_so`	Scostamento in byte, dall'inizio della stringa, corrispondente all'inizio della sottostringa individuata.
`regoff_t`	`rm_eo`	Scostamento in byte, dall'inizio della stringa, corrispondente al carattere **successivo** alla sottostringa individuata.

Macro-variabile	Da usare come opzioni della funzione ***regcomp()*** per la compilazione di un'espressione regolare.
`REG_EXTENDED`	L'espressione regolare fornita è di tipo ERE (estesa). Se non si usa questa opzione, si intende che l'espressione sia di tipo BRE.
`REG_ICASE`	L'espressione regolare fornita va valutata senza distinguere tra lettere maiuscole o minuscole.
`REG_NOSUB`	Per la compilazione dell'espressione regolare non si intende tenere conto della corrispondenza eventuale di sottostringhe; in altri termini, non si vogliono considerare le parentesi '**`\(`**' e '**`\)`**', oppure '**`(`**' e '**`)`**' (a seconda che si tratti di ERE o BRE). In tal caso, l'espressione regolare serve per il confronto, ma non per estrapolare porzioni del risultato ottenuto.
`REG_NEWLINE`	In condizioni normali, il codice *new-line* contenuto nella stringa con cui l'espressione regolare deve essere confrontata, viene trattato come gli altri caratteri. Con l'opzione '**`REG_NEWLINE`**', invece, l'operatore '**`^`**' individua l'inizio di un testo che segue un codice *new-line*, mentre l'operatore '**`$`**' individua la fine di un testo che precede un codice *new-line*.

Macro-variabile Significato	Da usare come opzioni della funzione ***regexec()*** per la comparazione di un'espressione regolare già compilata con una stringa.
`REG_NOTBOL`	In condizioni normali, il carattere '`^`' trova corrispondenza con l'inizio di una stringa. Con questa opzione, si inibisce tale corrispondenza (*not begin of line*).
`REG_NOTEOL`	In condizioni normali, il carattere '`$`' trova corrispondenza con la fine di una stringa. Con questa opzione, si inibisce tale corrispondenza (*not end of line*).

Macro-variabile	Tipo di errore restituito dalla funzione ***regcomp()*** o da ***regexec()***.
`REG_BADBR`	Il contenuto di '`\{…\}`' (nel caso di BRE) o di '`{…}`' (nel caso di ERE), risulta non valido: potrebbe non trattarsi di un numero, oppure potrebbe esserci un numero troppo grande, oppure potrebbero esserci più di due numeri, oppure il primo potrebbe essere più grande del secondo. Infatti, il contenuto di tale raggruppamento deve essere un numero singolo, oppure due numeri separati da una virgola, dove il primo deve essere inferiore al secondo.
`REG_BADPAT`	Espressione regolare non valida (errore di sintassi).

Macro-variabile	Tipo di errore restituito dalla funzione ***regcomp()*** o da ***regexec()***.
`REG_BADRPT`	Un operatore di ripetizione, del tipo '`?`', '`*`' o '`+`', non è preceduto a un'espressione regolare, ovvero si trova in una posizione sbagliata.
`REG_ECOLLATE`	Elemento di collazione (*collating element*) non valido, nell'ambito della configurazione locale attuale.
`REG_ECTYPE`	Riferimento a una classe di caratteri non valida.
`REG_EESCAPE`	L'espressione regolare termina con '`\`' e ciò non è ammissibile.
`REG_ESUBREG`	Una sequenza '`\`***n***', dove ***n*** è un numero, è errata.
`REG_EBRACE`	Le parentesi graffe che descrivono la ripetizione di qualcosa non bilanciano. Può trattarsi di sequenze del tipo '`\{`…`\}`' per le espressioni BRE o del tipo '`{`…`}`' per le espressioni ERE.
`REG_EBRACK`	Parentesi quadre non bilanciate (parentesi aperta e non chiusa, o viceversa).
`REG_EPAREN`	Le parentesi tonde che descrivono delle sottoespressioni non bilanciano. Può trattarsi di sequenze del tipo '`\(`…`\)`' per le espressioni BRE o del tipo '`(`…`)`' per le espressioni ERE.
`REG_ERANGE`	Un'estremità di un intervallo di valori non è valido.
`REG_ESPACE`	Il procedimento di interpretazione dell'espressione regolare porta all'esaurimento della memoria disponibile.

Macro-variabile	Tipo di errore restituito dalla funzione ***regcomp()*** o da ***regexec()***.
`REG_NOMATCH`	Non c'è corrispondenza tra espressione regolare e stringa.

File «sys/stat.h»

«

Funzione	Descrizione
`int chmod (const char *`***path***`,` `mode_t` ***mode***`);`	Cambia i permessi di un file, individuato dal suo percorso nel file system, rappresentati da una variabile di tipo '**`mode_t`**' (il tipo di file non può essere cambiato).
`int fchmod (int` ***fdn***`,` `mode_t` ***mode***`);`	Cambia i permessi di un file aperto, individuato da un descrittore, rappresentati da una variabile di tipo '**`mode_t`**' (il tipo di file non può essere cambiato).
`int fstat (int` ***fdn***`,` `struct stat *`***buffer***`);`	Aggiorna i membri della struttura a cui punta ***buffer***, con le informazioni relative al file aperto con descrittore ***fdn***.

Funzione	Descrizione
`int lstat (const char *restrict` ***path*****`,`** `struct stat *restrict` ***buffer*****`);`**	Aggiorna i membri della struttura a cui punta ***buffer***, con le informazioni relative al file individuato dal percorso ***path***. Ma a differenza di ***stat()***, se il file richiesto è un collegamento simbolico, si ottengono le informazioni del file che rappresenta il collegamento stesso, ignorando ciò a cui questo punterebbe.
`int mkdir (const char *`***path*****`,`** `mode_t` ***mode*****`);`**	Crea una directory, specificata attraverso un percorso del file system, con i permessi indicati nel parametro ***mode***, dove non si può specificare il tipo di file. I permessi richiesti vengono filtrati dalla maschera dei permessi.
`int mkfifo (const char *`***path*****`,`** `mode_t` ***mode*****`);`**	Crea un file FIFO, individuato dal suo percorso nel file system, utilizzando i permessi richiesti, subordinatamente al filtro della maschera dei permessi.

Funzione	Descrizione
`int mknod (const char *`***path***`,` `mode_t mode, dev_t dev);`	Crea virtualmente un file di qualunque tipo, specificando il percorso nel file system, il tipo di file e i permessi (in questo caso il parametro ***mode*** serve principalmente per specificare il tipo di file) e il numero di dispositivo complessivo, nel caso particolare di un file di dispositivo. In condizioni normali, non dovrebbe essere possibile la creazione di una directory, ma se anche fosse possibile, sarebbe sconsigliabile l'uso di questa funzione per tale scopo.
`int stat` `(const char *restrict` ***path***`,` `struct stat *restrict` ***buffer***`);`	Aggiorna i membri della struttura a cui punta ***buffer***, con le informazioni relative al file individuato dal percorso ***path***.
`mode_t umask (mode_t` ***mask***`);`	Definisce la maschera dei permessi per il processo in corso, restituendo il valore precedente della maschera.

Macroistruzione	Descrizione
`S_ISBLK (`***mode***`)`	Restituisce *Vero* se il parametro ***mode*** rappresenta un file di dispositivo a blocchi.
`S_ISCHR (`***mode***`)`	Restituisce *Vero* se il parametro ***mode*** rappresenta un file di dispositivo a caratteri.
`S_ISFIFO (`***mode***`)`	Restituisce *Vero* se il parametro ***mode*** rappresenta un file FIFO.
`S_ISREG (`***mode***`)`	Restituisce *Vero* se il parametro ***mode*** rappresenta un file puro e semplice.
`S_ISDIR (`***mode***`)`	Restituisce *Vero* se il parametro ***mode*** rappresenta una directory.
`S_ISLNK (`***mode***`)`	Restituisce *Vero* se il parametro ***mode*** rappresenta un collegamento simbolico.
`S_ISSOCK (`***mode***`)`	Restituisce *Vero* se il parametro ***mode*** rappresenta un socket di dominio Unix.

Tipo	Nome	Membri noti del tipo derivato '**`struct stat`**'
`dev_t`	`st_dev`	Numero di dispositivo dell'unità contenente il file.
`ino_t`	`st_ino`	Numero di inode del file.
`mode_t`	`st_mode`	Tipo di file e permessi di accesso relativi.
`nlink_t`	`st_nlink`	Collegamenti riferiti al file.
`uid_t`	`st_uid`	Numero UID dell'utente proprietario del file.

Tipo	Nome	Membri noti del tipo derivato '**struct stat**'
`gid_t`	`st_gid`	Numero GID del gruppo proprietario del file.
`dev_t`	`st_rdev`	Numero di dispositivo rappresentato, nel caso si tratti effettivamente di un file di dispositivo.
`off_t`	`st_size`	Dimensione del file.
`time_t`	`st_atime`	Data e orario dell'ultimo accesso al file.
`time_t`	`st_mtime`	Data e orario dell'ultima modifica apportata al contenuto del file.
`time_t`	`st_ctime`	Data e orario dell'ultima modifica di inode (data di creazione dell'inode).
`blksize_t`	`st_blksize`	Dimensione del blocco per le operazioni di input-output.
`blkcnt_t`	`st_blocks`	Dimensione del file espressa in blocchi.

Macro-variabile	Da usare per la definizione di un tipo di file, in variabili di tipo '**mode_t**'.
`S_IFMT`	Maschera che raccoglie tutti i bit che individuano il tipo di file.
`S_IFBLK`	File di dispositivo a blocchi.
`S_IFCHR`	File di dispositivo a caratteri.
`S_IFIFO`	File FIFO.
`S_IFREG`	File puro e semplice.
`S_IFDIR`	Directory.

Macro-variabile	Da usare per la definizione di un tipo di file, in variabili di tipo '**mode_t**'.
S_IFLNK	Collegamento simbolico.
S_IFSOCK	Socket di dominio Unix.

Macro-variabile	Da usare per la definizione dei permessi di un file, in variabili di tipo '**mode_t**'.
S_ISUID	SUID.
S_ISGID	SGID.
S_ISVTX	Sticky.
S_IRWXU	Lettura, scrittura ed esecuzione per l'utente proprietario.
S_IRUSR	Lettura per l'utente proprietario.
S_IWUSR	Scrittura per l'utente proprietario.
S_IXUSR	Esecuzione per l'utente proprietario.
S_IRWXG	Lettura, scrittura ed esecuzione per il gruppo.
S_IRGRP	Lettura per il gruppo.
S_IWGRP	Scrittura per il gruppo.
S_IXGRP	Esecuzione per il gruppo.
S_IRWXO	Lettura, scrittura ed esecuzione per gli altri utenti.

Macro-variabile	Da usare per la definizione dei permessi di un file, in variabili di tipo '**mode_t**'.
S_IROTH	Lettura per gli altri utenti.
S_IWOTH	Scrittura per gli altri utenti.
S_IXOTH	Esecuzione per gli altri utenti.

Gettext

Gettext[1] è un sistema che aiuta nella traduzione dei messaggi dei programmi e al loro mantenimento, il quale però non fa parte della libreria standard dei linguaggi di programmazione. Ci possono essere molti modi per realizzare un programma multilingua, ma Gettext rappresenta probabilmente il metodo più semplice in pratica, consentendo la traduzione successiva senza interferire con un eseguibile già pronto, purché predisposto per questo.

71.1 Principio di funzionamento

La logica di Gettext è molto semplice: il programma incorpora solo i messaggi in inglese; all'esterno si associano dei file, uno per ogni linguaggio disponibile, con le traduzioni corrispondenti. Non è necessario «codificare» i messaggi in qualche modo, perché la corrispondenza avviene in modo letterale, in base al testo originale.

```
msgid "%s: cannot create the temporary file %s\n"
msgstr "%s: non è possibile creare il file temporaneo %s\n"
```

L'esempio, che mostra un estratto ipotetico di un file PO di Gettext (*Portable object*), serve a comprendere il concetto: la stringa preceduta dalla parola chiave '**`msgid`**' (*message identity*) è quella di riferimento, che viene rimpiazzata automaticamente da quella sottostante, preceduta dalla parola chiave '**`msgstr`**'.

Le stringhe e le traduzioni di Gettext sono costanti, nel senso che '**`%s`**' viene preso come tale, mentre è il programma che lo sostituisce opportunamente. In questo senso, bisogna considerare che Gettext è nato per il linguaggio C, per essere usato in stringhe che siano argomento di funzioni come ***printf()*** e ***sprintf()***.

71.2 Fasi di preparazione

«

La predisposizione di un programma per Gettext potrebbe essere fatta in modo più o meno automatico, attraverso strumenti specifici, oppure si può procedere in modo più semplice, anche se più oneroso dal punto di vista del tempo impiegato. Qui si intende mostrare questo modo più semplice per permettere al lettore di comprendere il concetto. La documentazione di Gettext è di per sé molto dettagliata.

Per prima cosa, il sorgente C deve essere predisposto attraverso l'inclusione di alcuni file di intestazione, quindi le stringhe vengono inglobate dalla funzione ***gettext()***. Quello che segue è il classico programma che visualizza un messaggio ed esce; si suppone che si tratti del file '`ciao.c`':

```
#include <stdio.h>
int main ()
{
    printf ("Hello world\n");
}
```

Nel listato successivo si vede come deve essere trasformato; va osservata l'inclusione del file di intestazione '`libintl.h`':

```
#include <stdio.h>
#include <libintl.h>
#include <locale.h>

#define PACKAGE "ciao"
#define LOCALEDIR "/var/tmp"

int main ()
{
    setlocale (LC_ALL, "");
    bindtextdomain (PACKAGE, LOCALEDIR);
    textdomain (PACKAGE);

    printf (gettext ("Hello world\n"));
}
```

Le funzioni '**bindtextdomain**' e '**textdomain**' utilizzano come argomenti delle macro (costanti manifeste), in modo da generalizzare il funzionamento e rendere esterna la definizione di queste componenti. A parte questi particolari, si nota che ***printf()*** non ha più come argomento la costante di prima, ma la funzione ***gettext()***.

Il programma può essere compilato, anche se non è ancora disponibile una traduzione:

```
$ cc -o ciao ciao.c[Invio]
```

La fase successiva richiede la creazione di un file PO, attraverso l'aiuto del programma '**xgettext**':

```
$ xgettext ciao.c[Invio]
```

Quello che si ottiene nella directory corrente è il file '`messages.po`', contenente esattamente il testo seguente:

```
# SOME DESCRIPTIVE TITLE.
```

```
# Copyright (C) YEAR Free Software Foundation, Inc.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"POT-Creation-Date: 2000-05-15 23:05+0200\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: ENCODING\n"

#: ciao.c:18
msgid "Hello world\n"
msgstr ""
```

Questo file deve essere modificato, in particolare per ciò che riguarda le prime direttive, oltre che per aggiungere la traduzione della frase che viene visualizzata dal programma. Per esempio, così:

```
# Ciaomondo PO file.
# Copyright (C) 2000 Pinco Pallino
# Pinco Pallino <ppinco@dinkel.brot.dg>, 2000.
#
msgid ""
msgstr ""
"Project-Id-Version: ciao-0.1\n"
"POT-Creation-Date: 2000-05-15 23:05+0200\n"
"PO-Revision-Date: 2000-05-15 22:52+0200\n"
"Last-Translator: Pinco Pallino <ppinco@dinkel.brot.dg>\n"
"Language-Team: Italian <it@li.org>\n"
"MIME-Version: 1.0\n"
```

```
"Content-Type: text/plain; charset=iso-8859-1\n"
"Content-Transfer-Encoding: 8bit\n"

#: ciao.c:18
msgid "Hello world\n"
msgstr "Ciao mondo\n"
```

Si osservi che è stato necessario togliere la riga contenente il commento speciale '**#, fuzzy**'.

Il file viene salvato con un nome appropriato; per esempio '`ciao.po`'. Quindi si passa alla sua compilazione, per ottenere il file '`ciao.mo`':

```
$ msgfmt -vvvv -o ciao.mo ciao.po[Invio]
```

Dal momento che il file in questione contiene la traduzione in italiano del programma, deve essere collocato all'interno della gerarchia '`it/LC_MESSAGES/`', a sua volta a partire dalla directory dichiarata con la funzione ***bindtextdomain()***, cioè '`/var/tmp/it/LC_MESSAGES/`' secondo quanto definito nel sorgente C.

A questo punto, dopo la collocazione appropriata del file compilato della traduzione, se la configurazione locale è corretta, lanciando l'eseguibile '**ciao**' si dovrebbe vedere il messaggio tradotto. Eventualmente si veda quanto descritto nella sezione 16.11 per quanto riguarda la configurazione della localizzazione.

71.3 Abbinamento a un «pacchetto»

«

Perché Gettext sappia qual è il file che contiene i messaggi tradotti, nell'ambito della configurazione locale, fa riferimento a un nome che viene definito «pacchetto», che di solito si sceglie opportunamente simile a quello del programma per il quale si fa la traduzione:

```
textdomain ("pippo");
```

L'esempio mostra l'istruzione da usare in un programma C per stabilire il nome del pacchetto secondo Gettext. Questo nome stabilisce che Gettext debba cercare il file '***sigla_locale***`/LC_MESSAGES/pippo.mo`'. Gettext determina il nome della prima parte del percorso, corrispondente a ciò che qui è stato mostrato con la metavariabile ***sigla_locale***, analizzando alcune variabili di ambiente; precisamente segue questo ordine:

- '**`LANGUAGE`**'
- '**`LC_ALL`**'
- altre variabili '**`LC_`***'
- '**`LANG`**'

Dal valore contenuto in queste variabili si estrae la prima parte: quella che arriva fino al primo punto, se c'è. In pratica, se per ipotesi la variabile '**`LANG`**' contiene il valore '**`it_IT.ISO-8859-1`**', per Gettext è importante solo '**`it_IT`**'. Tuttavia, anche questa informazione tende a essere eccessiva, dal momento che contiene, oltre al linguaggio, anche l'area nazionale. In pratica, alla fine contano solo le prime due lettere, che esprimono il linguaggio in base allo standard ISO 639 (tabella 13.4).

Gettext analizza il contenuto delle variabili di ambiente perché con la funzione ***setlocale()*** è stata azzerata internamente la definizione '**LC_ALL**'. Usando la funzione ***setlocale()*** si potrebbe imporre un certo linguaggio, indipendentemente dalle variabili di ambiente relative.

Tornando all'esempio iniziale, si tratta del file '`it/LC_MESSAGES/pippo.mo`', che in condizioni normali, Gettext cerca a partire dalla directory '`/usr/share/locale/`' (o eventualmente da un'altra posizione in base al modo in cui è stato compilato). Tuttavia è possibile richiedere espressamente una collocazione differente attraverso un'istruzione già vista da collocare nel programma interessato:

```
bindtextdomain ("pippo", "/var/tmp");
```

In tal caso, se si scrive questo in un programma, Gettext va a cercare precisamente il file '`/var/tmp/it/LC_MESSAGES/pippo.mo`'.

71.4 Creazione e mantenimento dei file PO

È già stato mostrato in breve come si crea un file PO attraverso il programma '**xgettext**'. È il caso di osservare che '**xgettext**' può ricevere l'indicazione di più file sorgenti che fanno capo allo stesso dominio di traduzione:

```
xgettext [opzioni] file_sorgente...
```

In particolare, tra le opzioni può essere interessante segnalare '**--default-domain=*dominio***', che serve a '**xgettext**' per conoscere il dominio a cui si fa riferimento, creando così il file '***dominio*.po**', invece del solito '`messages.po`'.

```
$ xgettext --default-domain=ciao *.c[Invio]
```

L'esempio mostra come ottenere il file 'ciao.po' a partire da tutti i file che terminano con l'estensione '.c'.

Quando si aggiornano i sorgenti di un programma già tradotto, si pone il problema di aggiornare nello stesso modo i file PO precedenti. Per fare questo si deve ricreare il file PO iniziale non tradotto, nel modo appena visto, quindi si usa il programma '**msgmerge**':

```
msgmerge [opzioni] file_po_originale file_po_successivo > file_po_aggiornato
```

In pratica, '**msgmerge**' fonde assieme due file PO, preservando le traduzioni del primo file riferite a messaggi che si trovano ancora nel secondo. Per esempio, se si dispone già del file 'vecchio.po' con le traduzioni, mentre con '**xgettext**' è appena stato generato un file PO non tradotto per lo stesso programma, che qui viene chiamato 'non_tradotto.po', si può ottenere un nuovo file PO con le traduzioni vecchie ancora valide e con i messaggi nuovi da tradurre:

```
$ msgmerge vecchio.po non_tradotto.po > nuovo.po[Invio]
```

Naturalmente, questa operazione si fa nel momento in cui ci si accinge ad aggiornare materialmente la traduzione del programma, altrimenti questo lavoro non avrebbe senso, dal momento che un file PO contenente messaggi non tradotti non può essere compilato.

```
msgfmt [opzioni] file_po
```

Il programma '**msgfmt**' è quello che si occupa di compilare i file PO

ottenendo i file MO, adatti alla propria piattaforma. È praticamente indispensabile utilizzare l'opzione '**--output-file=*file_mo***' ('**-o**'), per indicare il nome del file da creare. Inoltre, è opportuno utilizzare più di una volta l'opzione '**--verbose**' ('**-v**') per avere una visione chiara del procedimento, ovvero dei motivi per i quali alle volte il file non viene compilato.

```
$ msgfmt -vvvv --output-file=prova.mo prova.po[Invio]
```

L'esempio mostra l'utilizzo tipico di questo programma, dove in particolare viene richiesto un livello di dettaglio delle informazioni generate molto elevato (quattro volte '**-v**').

71.4.1 Commenti «fuzzy»

Ogni volta che qualche indicazione all'interno di un file PO è incerta, in quanto predefinita o determinata automaticamente in modo non sicuro, viene aggiunto un commento speciale contenente la parola '**fuzzy**'. In presenza di commenti del genere si richiede un intervento manuale, dopo il quale deve essere rimossa tale parola, altrimenti '**msgfmt**' si rifiuta di completare la compilazione dei file PO.

71.5 Gettext con i programmi Perl

Esiste la possibilità di utilizzare Gettext anche nei programmi Perl. Per questo è necessario includere nel programma Perl il riferimento a un modulo esterno: Perl-gettext. Il tutto si svolge in maniera molto simile a un programma C, inserendo inizialmente le istruzioni seguenti:

```
use POSIX;
use Locale::gettext;
setlocale (LC_ALL, "");
textdomain ("dominio_gettext");
[bindtextdomain ("dominio_gettext", "directory");]
```

Per esempio, se si tratta del programma «Pippo», il dominio per Gettext potrebbe essere convenientemente «pippo», arrivando al risultato seguente:

```
use POSIX;
use Locale::gettext;
setlocale (LC_ALL, "");
textdomain ("pippo");
bindtextdomain ("pippo", "/opt/pippo/locale");
```

Potrebbe essere che la funzione ***bindtextdomain()*** non si comporti come previsto; in tal caso sarebbe meglio evitarne l'uso.

Per il resto, tutto funziona come per i sorgenti scritti in C:

```
print STDOUT (gettext ("Hello world\n"));
```

Tuttavia, Perl non è identico al C, per cui occorre osservare alcune situazioni specifiche. In particolare, non è possibile inserire in un argomento della funzione ***gettext()*** una variabile di Perl che deve essere espansa, perché questa espansione avverrebbe prima che ***gettext()*** possa ricevere tale argomento. Pertanto, l'esempio seguente non può essere tradotto:

```
# Esempio errato.
print STDOUT (gettext ("Il file $file contiene "
                       "caratteri non validi\n"));
```

Il modo giusto di agire è quello di sostituire ***print()*** con ***printf()***, come nell'esempio seguente:

```
# Esempio corretto.
printf STDOUT (gettext ("Il file %s contiene "
                        "caratteri non validi\n"),
               $file);
```

Infatti, il parametro '**`%s`**' viene sostituito alla fine da '**`printf`**', per cui inizialmente la stringa non viene modificata.

Un altro problema da considerare sono i messaggi lunghi, che richiedono più righe. In Perl si potrebbe fare una cosa del genere:

```
printf STDOUT
  (gettext
    ( "Usage: %s --input-type=TYPE INPUT_FILE REPORT_FILE\n"
    . "       %s --help\n"
    . "       %s --version\n"
    . "\n"
    . "Check for HTTP and FTP URI inside a text.\n"
    . "\n"
    . "Options:\n"
    . "--help              display this help and exit.\n"
    . "--version           display version information \n"
    . "                      and exit.\n"
    . "--input-type=TYPE   define the input type:\n"
    . "    standard        input is a simple text file;\n"
    . "    html, sgml      input is a typical SGML file;\n"
    . "    texi, texinfo   input is a Texinfo source \n"
    . "                      file.\n"
    . "\n"
    . "Arguments:\n"
    . "\n"
    . "INPUT_FILE          the input file.\n"
    . "\n"
    . "REPORT_FILE         a file that is generated with \n"
```

```
    . "                          the reported errors.\n"),
    $program_name, $program_name, $program_name);
```

Ma questo non viene riconosciuto da '**xgettext**' che riesce a prelevare solo la prima riga:

```
#: urichk:55
#, c-format
msgid "Usage: %s --input-type=TYPE INPUT_FILE REPORT_FILE\n"
msgstr ""
```

In queste situazioni eccezionali, occorre intervenire a mano nel file PO; sia la prima volta che si crea il file, sia tutte le volte successive in cui lo si aggiorna.

71.5.1 Adattare il sorgente Perl per facilitare l'estrazione dei messaggi da tradurre

«

Se si prendono delle piccole precauzioni nella scrittura delle stringhe con Perl, è possibile filtrare il sorgente successivamente, per passarlo a '**xgettext**' in modo che questo possa interpretarlo correttamente (come se fosse un programma C). In pratica, occorre fare in modo che le stringhe siano unite, senza ottenerle attraverso un concatenamento. L'esempio già mostrato va modificato nel modo seguente:

```
    printf STDOUT
      (gettext
        ("\
Usage: %s --input-type=TYPE INPUT_FILE REPORT_FILE\
       %s --help\
       %s --version\
\
Check for HTTP and FTP URI inside a text.\
```

```
\
Options:\
--help                display this help and exit.\
--version             display version information and exit.\
--input-type=TYPE     define the input type:\
    standard          input is a simple text file;\
    html, sgml        input is a typical SGML file;\
    texi, texinfo     input is a Texinfo source file.\
\
Arguments:\
\
INPUT_FILE            the input file.\
\
REPORT_FILE           a file that is generated with the \
                        reported errors."),
        $program_name, $program_name, $program_name);
```

A questo punto, si può realizzare un piccolo programma Perl che inserisce il codice '**\n**' alla fine delle righe (sostituendolo alla barra obliqua inversa) e sostituisce il codice '**\@**' con '**@**':

```
#!/usr/bin/perl
$line = "";
while ($line = <STDIN>)
  {
    $line =~ s/\\$/\\n\\/;
    $line =~ s/\\\@/\@/g;
    print STDOUT ($line);
  }
```

Come si vede, il programma Perl che si ottiene legge dallo standard input ed emette il risultato della sua trasformazione attraverso lo standard output.

71.5.2 Alleviare gli inconvenienti di un modulo in più

«

Scrivere un programma Perl che faccia uso di Gettext, significa costringere a installare il modulo Perl-gettext. Purtroppo, una delle cose che complicano di più l'utilizzo di programmi Perl sono i moduli aggiuntivi necessari che devono essere installati perfettamente come previsto.

Questo potrebbe sembrare un problema secondario; invece non lo è affatto. A questo punto, se si vuole consentire al proprio programma Perl di funzionare anche in un ambiente non tanto amichevole, si deve prevedere una via di uscita:

```
#!/usr/bin/perl
#
…

use POSIX;
use Locale::gettext;
setlocale (LC_ALL, "");
textdomain ("pippo");

#sub gettext
#{
#    return $_[0];
#}

…
```

Come si vede nell'esempio, appare la dichiarazione di una funzione commentata, il cui scopo sarebbe quello di sostituirsi alla funzione ***gettext()*** del modulo '**`Locale::gettext`**'. Se non si dispone di Perl-gettext basta commentare la prima parte e togliere i commenti

dalla seconda: ovviamente i messaggi rimangono così nella lingua di partenza.

```
#!/usr/bin/perl
#
...

#use POSIX;
#use Locale::gettext;
#setlocale (LC_ALL, "");
#textdomain ("pippo");

sub gettext
{
    return $_[0];
}

...
```

[1] **Gettext** GNU GPL

Parte iv

COBOL

72 Manuale COBOL 1467
72.1 Caratteristiche del linguaggio 1477
72.2 Modulo di programmazione 1491
72.3 Divisione «IDENTIFICATION DIVISION» 1500
72.4 Divisione «ENVIRONMENT DIVISION» 1502
72.5 Divisione «DATA DIVISION» 1538
72.6 Descrizione delle variabili 1557
72.7 Tabelle 1571
72.8 Nomi di condizione, raggruppamento e qualificazione 1586
72.9 Modello di definizione della variabile 1594
72.10 Note sull'utilizzo dell'insieme di caratteri universale con il COBOL 1611
72.11 Divisione «PROCEDURE DIVISION» 1613
72.12 Istruzioni della divisione «PROCEDURE DIVISION» 1628
72.13 Riordino e fusione 1713
72.14 Riferimenti 1727
73 Programmare in COBOL 1729
73.1 Preparazione 1733
73.2 Esempi elementari 1743
73.3 Esempi elementari con i file 1785

73.4 Approfondimento: una tecnica per simulare la ricorsione in COBOL 1825
73.5 Riferimenti 1858

72.1 Caratteristiche del linguaggio 1477
72.1.1 Organizzazione del programma in forma sorgente 1477
72.1.2 Insieme dei caratteri 1478
72.1.3 Struttura del linguaggio 1480
72.1.4 Notazione sintattica 1490
72.2 Modulo di programmazione 1491
72.2.1 Indicatore 1493
72.2.2 Area A e area B 1495
72.2.3 Interpunzione 1496
72.3 Divisione «IDENTIFICATION DIVISION» 1500
72.3.1 Struttura 1500
72.3.2 Codifica della divisione 1501
72.4 Divisione «ENVIRONMENT DIVISION» 1502
72.4.1 Struttura 1503
72.4.2 Sezione «CONFIGURATION SECTION» 1503
72.4.3 Sezione «INPUT-OUTPUT SECTION» 1508
72.5 Divisione «DATA DIVISION» 1538
72.5.1 Sezione «FILE SECTION» 1540
72.5.2 Sezione «WORKING-STORAGE SECTION» . 1555
72.5.3 Altri livelli speciali 1556
72.6 Descrizione delle variabili 1557

72.6.1 Oggetto della dichiarazione 1560
72.6.2 Ridefinizione di una variabile 1560
72.6.3 Opzione «PICTURE» 1562
72.6.4 Opzione «USAGE» 1563
72.6.5 Opzione «SIGN» 1566
72.6.6 Opzione «OCCURS» 1567
72.6.7 Opzione «SYNCHRONIZED» 1568
72.6.8 Opzione «JUSTIFIED RIGHT» 1569
72.6.9 Opzione «BLANK WHEN ZERO» 1569
72.6.10 Opzione «VALUE» 1570
72.6.11 Opzione «RENAMES» 1570
72.7 Tabelle 1571
72.7.1 Dichiarazione di una tabella 1571
72.7.2 Riferimento al contenuto di una tabella 1573
72.7.3 Indice 1574
72.7.4 Tabelle di dimensione variabile 1576
72.7.5 Tabelle ordinate 1579
72.7.6 Scansione delle tabelle 1580
72.8 Nomi di condizione, raggruppamento e qualificazione 1586
72.8.1 Nomi di condizione 1586
72.8.2 Raggruppamento 1589
72.8.3 Qualificazione 1591
72.9 Modello di definizione della variabile 1594

72.9.1 Dichiarazione del modello di definizione della variabile 1595
72.9.2 Variabili alfanumeriche 1596
72.9.3 Variabili alfanumeriche modificate 1599
72.9.4 Variabili numeriche 1601
72.9.5 Variabili numeriche modificate 1604
72.10 Note sull'utilizzo dell'insieme di caratteri universale con il COBOL 1611
72.10.1 Stringhe letterali 1612
72.10.2 modello di definizione delle variabili 1612
72.10.3 Costanti figurative 1613
72.11 Divisione «PROCEDURE DIVISION» 1613
72.11.1 Gruppo di sezioni «DECLARATIVES» 1615
72.11.2 Sezioni e segmenti 1618
72.11.3 Gruppi di istruzioni e istruzioni condizionali . . . 1619
72.11.4 Sezioni, paragrafi e qualificazione 1621
72.11.5 Espressioni aritmetiche 1621
72.11.6 Espressioni condizionali 1623
72.11.7 Avverbi comuni 1627
72.12 Istruzioni della divisione «PROCEDURE DIVISION» 1628
72.12.1 Istruzione «ACCEPT» 1628
72.12.2 Istruzione «ADD» 1633
72.12.3 Istruzione «CLOSE» 1635

72.12.4 Istruzione «COMPUTE» 1636
72.12.5 Istruzione «DELETE» 1636
72.12.6 Istruzione «DISPLAY» 1638
72.12.7 Istruzione «DIVIDE» 1639
72.12.8 Istruzione «EXIT» 1642
72.12.9 Istruzione «GO TO» 1644
72.12.10 Istruzione «IF» 1645
72.12.11 Istruzione «INSPECT» 1646
72.12.12 Istruzione «MOVE» 1654
72.12.13 Istruzione «MULTIPLY» 1658
72.12.14 Istruzione «OPEN» 1660
72.12.15 Istruzione «PERFORM» 1664
72.12.16 Istruzione «READ» 1675
72.12.17 Istruzione «REWRITE» 1681
72.12.18 Istruzione «SEARCH» 1684
72.12.19 Istruzione «SET» 1695
72.12.20 Istruzione «START» 1699
72.12.21 Istruzione «STOP RUN» 1703
72.12.22 Istruzione «STRING» 1704
72.12.23 Istruzione «SUBTRACT» 1707
72.12.24 Istruzione «WRITE» 1709
72.13 Riordino e fusione 1713
72.13.1 Riordino 1713
72.13.2 Fusione 1717

72.13.3 Gestire i dati in ingresso o in uscita attraverso delle procedure 1720
72.13.4 Lettura del risultato dell'ordinamento o della fusione attraverso una procedura 1722
72.13.5 Acquisizione dei dati per il riordino da una procedura 1724
72.14 Riferimenti 1727

01 1548 66 1589 88 1586 ACCEPT 1628 ADD 1633 BLANK WHEN ZERO 1569 BLOCK CONTAINS 1543 CLOSE 1635 CODE-SET 1547 COMPUTE 1636 CONFIGURATION SECTION 1503 DATA DIVISION 1538 DATA RECORD 1545 DECLARATIVES 1615 DELETE 1636 DEPENDING ON 1576 DISPLAY 1638 DIVIDE 1639 ENVIRONMENT DIVISION 1502 EXIT 1642 FD 1541 FILE-CONTROL 1509 FILE SECTION 1540 FILLER 1560 GO TO 1644 IDENTIFICATION DIVISION 1500 IF 1645 INPUT-OUTPUT SECTION 1508 INSPECT 1646 I-O-CONTROL 1528 JUSTIFIED RIGHT 1569 LABEL RECORD 1545 MERGE 1717 MOVE 1654 MULTIPLY 1658 OBJECT-COMPUTER 1504 OCCURS 1567 1571 OPEN 1660 PERFORM 1664 PICTURE 1594 PROCEDURE DIVISION 1613 1628 1713 READ 1675 RECORD CONTAINS 1546 REDEFINES 1548 1560 RELEASE 1724 RENAMES 1589 RETURN 1722 REWRITE 1681 SD 1543 SEARCH 1684 SELECT 1512 1516 1522 SET 1695 SIGN IS 1566 SORT 1524 1713 SOURCE-COMPUTER 1504 SPECIAL-NAMES 1505 START 1699 STOP RUN 1703 STRING 1704 SUBTRACT 1707 SYNCHRONIZED 1568 USAGE 1563 VALUE 1570 VALUE OF 1547 WORKING-STORAGE SECTION 1555 WRITE 1709

Ogni manuale COBOL tradizionale riporta una premessa che cita le origini del linguaggio e le fonti a cui si fa riferimento. Questo tipo di premessa ha soprattutto un valore storico e con tale spirito viene inserita qui.

Il testo seguente è una traduzione tratta dalla pubblicazione *COBOL*, edita dalla Conferenza sui linguaggi dei sistemi di elaborazione dati, CODASYL (*Conference on data system languages*), stampata a cura dell'ufficio stampa del governo degli Stati Uniti d'America.

> «*Questa pubblicazione si basa sul sistema COBOL sviluppato nel 1959 da un comitato composto da utenti governativi e costruttori di elaboratori. Le organizzazioni che hanno preso parte ai lavori iniziali sono state:*
>
> *Air Material Command, U.S. Air Force;*
> *Bureau of Standards, U.S. Department of Commerce;*
> *Burroughs Corporation;*
> *David Tylor Model Basin, Bureau of Ships, U.S. Navy;*
> *Electronic Data processing Division, Minneapolis-Honeywell Regulator Company;*
> *International Business Machines Corporation;*
> *Radio Corporation of America;*
> *Sylvania Electric Products, Inc.;*
> *UNIVAC Division of Sperry Rand Corporation.*
>
> *Oltre alle suddette organizzazioni, le seguenti altre partecipano ai lavori del Gruppo di revisione:*
>
> *Allstate Insurance Company;*
> *The Bendix Corporation, Computer Division;*
> *Control Data Corporation;*

E.I. du Pont de Nemours and Company;
General Electric Company;
General Motors Corporation;
The National Cash Register Company;
Philco Corporation;
Standard Oil Company (New Jersey);
United States Steel Corporation.

Questo manuale COBOL è risultato dalla collaborazione fra tutte le organizzazioni citate.

Nessuna garanzia, espressa o tacita, è fornita dal comitato o dai singoli collaboratori, circa l'esattezza e il buon funzionamento del sistema di programmazione e del linguaggio. Inoltre, sia il comitato, sia i suoi collaboratori, non si assumono alcuna responsabilità in ordine a quanto esposto.

È ragionevole attendersi che molti perfezionamenti e aggiunte vengano fatte al COBOL. Si farà ogni sforzo per assicurare che miglioramenti e correzioni siano apportate con criteri di continuità, tenendo debito conto degli investimenti effettuati dagli utenti nel settore della programmazione. Tuttavia, tali garanzie potranno essere efficacemente mantenute soltanto da coloro che apporteranno perfezionamento o correzioni.

Sono state predisposte apposite procedure per l'aggiornamento del COBOL. Le richieste di informazioni circa tali procedure e sulle modalità per proporre modifiche dovranno essere inoltrate al comitato esecutivo della Conferenza sui linguaggi dei sistemi di elaborazione dati.

Gli autori e i titolari dei diritti di autore e di riproduzione del materiale così protetto, usato nel presente manuale:

FLOW-MATIC (marchio depositato dalla Sperry Rand Corporation) - Sistema di programmazione per i calcolatori UNIVAC® I e II, Data Automation Systems © 1958, 1959 Sperry Rand Corporation; IBM Commercial Traslator, Codice F28-8013, © 1959 IBM; FACT, DSI 27A5260-2760 © 1960 della Minneapolis-Honeywell; hanno esplicitamente autorizzato l'uso di tale materiale, in tutto o in parte, nelle specifiche del COBOL. Tale autorizzazione si estende alla riproduzione e all'uso delle specifiche COBOL in manuali di programmazione o in pubblicazioni analoghe.

Qualsiasi organizzazione che intenda riprodurre il rapporto COBOL e le specifiche iniziali in tutto o in parte, usando idee ricavate da tale rapporto o utilizzando il rapporto stesso come elemento base per un manuale di istruzione o per qualsiasi altro scopo, è libera di farlo. Tuttavia, si richiede a tutte queste organizzazioni di riprodurre la presente sezione, come parte dell'introduzione. Coloro che invece utilizzano brevi citazioni, come nelle rassegne dei nuovi libri, sono pregati di citare la fonte ma non di riprodurre l'intera sezione.»

Successivamente alla Conferenza sui linguaggi dei sistemi di elaborazione dati, CODASYL, il compito di definire lo standard del linguaggio COBOL è stato preso dall'istituto ANSI (*American national standards institute*), che chiede nuovamente di citare la fonte nei manuali di tale linguaggio. Il testo seguente è citato in lingua originale.

«*Any organization interested in reproducing the COBOL standard and specifications in whole or in part, using ideas from*

this document as the basis for an instruction manual or for any other purpose, is free to do so. However, all such organizations are requested to reproduce the following acknowledgement paragraphs in their entirety as part of the preface to any such publication (any organization using a short passage from this document, such as in a book review, is requested to mention "COBOL" in acknowledgement of the source, but need not quote the acknowledgement):

COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

No warranty, expressed or implied, is made by any contributor or by the CODASYL Programming Language Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

The authors and copyright holders of the copyrighted material used herein have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

FLOW-MATIC (trademark of Sperry Rand Corporation), Programming for the Univac++ I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator Form No. F 28-8013, copyrighted 1959 by IBM, FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell.»

Il linguaggio COBOL nasce nel 1959, come linguaggio standard per l'amministrazione degli uffici e il nome sta per *Common business oriented language*, descrivendo precisamente il suo obiettivo.

L'origine così lontana del linguaggio COBOL è responsabile della prolissità della sua sintassi e dei vincoli di forma nella struttura che il programma sorgente deve avere. Tuttavia, questo linguaggio è eccezionale nella gestione dei dati, avvicinandosi alle funzionalità di un sistema di gestione di basi di dati (ovvero un DBMS).

Il linguaggio COBOL è nato da un comitato di utenti e di produttori di elaboratori, con lo scopo di rimanere uniforme, il più possibile, su tutte le piattaforme. Generalmente si considera, correttamente, che il C rappresenti l'esempio di linguaggio di programmazione standard per definizione, ma i contesti sono differenti: il linguaggio C serve a consentire la migrazione di un sistema operativo da una macchina all'altra, mentre il linguaggio COBOL è fatto per consentire la migrazione di programmi applicativi su architetture fisiche e sistemi operativi differenti.

Il linguaggio COBOL è fatto per poter funzionare su sistemi operativi che possono anche essere privi di qualunque astrazione dell'hardware; pertanto, una porzione apposita nella struttura del sorgente è riservata alla dichiarazione delle unità fisiche per lo scambio dei dati (la divisione '**`ENVIRONMENT DIVISION`**'). Utilizzando il COBOL in un ambiente abbastanza evoluto, quanto può esserlo un sistema Unix, molte informazioni diventano inutili e implicite, ma il fatto che con questo linguaggio ci sia la possibilità di operare con la maggior parte degli elaboratori fabbricati dal 1959 in poi, lo rende speciale e degno di apprezzamento per lungo tempo.

Il linguaggio COBOL ha subito nel tempo diverse revisioni, indicate generalmente attraverso l'anno di edizione; un punto di riferimento abbastanza comune è lo standard del 1985, a cui questo manuale, in parte, cerca di adeguarsi. Per quanto riguarda l'ente ISO, è disponibile lo standard ISO 1989.

72.1 Caratteristiche del linguaggio

Il linguaggio COBOL si basa convenzionalmente sulla lingua inglese ed è composto sommariamente da ***parole***, ***istruzioni***, gruppi di istruzioni, ***paragrafi*** e ***sezioni***.

72.1.1 Organizzazione del programma in forma sorgente

Ogni programma COBOL deve contenere quattro divisioni, anche se queste dovessero essere vuote, rispettando l'ordine seguente:

1. '**`IDENTIFICATION DIVISION`**'
2. '**`ENVIRONMENT DIVISION`**'
3. '**`DATA DIVISION`**'
4. '**`PROCEDURE DIVISION`**'

La divisione '**`IDENTIFICATION DIVISION`**' serve a identificare il programma. Vi si possono includere informazioni generali, come il nome del programma stesso, la data di edizione, la data di compilazione, il nome dell'elaboratore per il quale è stato scritto e altre annotazioni.

La divisione '**`ENVIRONMENT DIVISION`**' specifica le apparecchiature usata e i file che servono al programma.

La divisione '**DATA DIVISION**' contiene la descrizione dei file e dei record relativi, creati o utilizzati dal programma, assieme a tutte le altre variabili e costanti che servono al programma.

La divisione '**PROCEDURE DIVISION**' specifica il procedimento elaborativo da applicare ai dati.

Le «azioni» descritte nel programma COBOL sono espresse in termini di istruzioni, che possono essere riunite in gruppi di istruzioni e poi in paragrafi.

72.1.2 Insieme dei caratteri

«

I compilatori tradizionali del linguaggio COBOL adottano, a seconda dei casi, il codice ASCII o il codice EBCDIC per la rappresentazione interna dei caratteri; inoltre, in un programma sorgente si può usare soltanto un insieme ristretto di simboli, con l'eccezione del contenuto delle costanti alfanumeriche, che invece è abbastanza libero.

Tabella 72.1. I simboli disponibili nel linguaggio, in generale.

Simboli	Descrizione	Simboli	Descrizione
'**0**'..'**9**'	cifre numeriche	'**A**'..'**Z**'	lettere maiuscole dell'alfabeto inglese (latino)
' '	spazio		
'**+**'	segno più	'**-**'	segno meno o trattino
'*****'	asterisco	'**/**'	barra obliqua
'**$**'	dollaro o segno di valuta	'**,**'	virgola
'**;**'	punto e virgola	'**.**'	punto fermo
'**(**'	parentesi aperta	'**)**'	parentesi chiusa

Simboli	Descrizione	Simboli	Descrizione
‘<’	minore	‘>’	maggiore

Si osservi che il segno di valuta, rappresentato normalmente dal dollaro, può essere ridefinito e rappresentato da un altro simbolo.

Tabella 72.2. Caratteri usati per l’interpunzione.

Simboli	Descrizione	Simboli	Descrizione
‘ ’	spazio bianco		
‘,’	virgola	‘;’	punto e virgola
‘.’	punto fermo	‘"’	virgolette
‘(’	parentesi aperta	‘)’	parentesi chiusa

Tabella 72.3. Caratteri usati per formulare le parole.

Simboli	Descrizione	Simboli	Descrizione
‘A’..‘Z’	lettere alfabetiche maiuscole, senza accenti	‘0’..‘9’	cifre numeriche
‘–’	trattino		

Tabella 72.4. Caratteri usati come operatori aritmetici.

Simboli	Descrizione	Simboli	Descrizione
‘+’	addizione	‘–’	sottrazione
‘*’	moltiplicazione	‘/’	divisione
‘(’	aperta parentesi	‘)’	chiusa parentesi

Tabella 72.5. Caratteri usati nelle relazioni.

Simboli	Descrizione	Simboli	Descrizione
‘=’	uguale a		
‘<’	minore di	‘>’	maggiore di

Si osservi che, al contrario di tanti altri linguaggi, nati però in momenti successivi, il COBOL non prevede l’uso del trattino basso (‘_’).

72.1.3 Struttura del linguaggio

«

Il testo di un programma sorgente COBOL è costruito con stringhe di caratteri e separatori, secondo le regole descritte nelle sezioni successive.

72.1.3.1 Separatori

«

Un separatore è una stringa composta da uno o più caratteri di interpunzione, rispettando le regole seguenti. Si osservi che queste regole non si applicano al contenuto delle costanti non numeriche (le stringhe letterali) e naturalmente non si applicano ai commenti.

- La virgola e il punto e virgola sono separatori, tranne quando appaiono nel modello di definizione di una variabile (‘**`PICTURE`**’), dove invece sono trattati come parte del modello stesso. La virgola e il punto e virgola, se usati come separatori, possono essere impiegati al posto dello spazio.
- Un punto fermo, seguito da uno spazio, è un separatore. Il punto fermo può apparire soltanto dove ciò è permesso esplicitamente dalle regole grammaticali del linguaggio.

- Le parentesi tonde, usate in coppia, aperta e chiusa, sono separatori. Possono essere usate per delimitare indici, espressioni aritmetiche e condizioni.
- Le virgolette sono separatori. Le virgolette di apertura devono essere precedute da uno spazio o da una parentesi aperta; le virgolette di chiusura devono essere seguite, alternativamente da: uno spazio, una virgola, un punto e virgola, un punto fermo oppure una parentesi chiusa.

  Le virgolette possono apparire solo in coppia, per delimitare costanti alfanumeriche, tranne quando le costanti continuano nella riga successiva.
- Lo spazio usato come separatore può precedere o seguire tutti gli altri separatori, tranne nei casi previsti dalle altre regole grammaticali del linguaggio. Uno spazio compreso tra una coppia di virgolette è una costante alfanumerica e non costituisce un separatore.

I caratteri di interpunzione che appaiono all'interno di un modello di definizione di una variabile ('**`PICTURE`**') o di una costante numerica, non sono considerati caratteri di interpunzione, piuttosto sono simboli usati per caratterizzare il modello relativo o la costante (le regole per la dichiarazione di un modello di definizione di una variabile sono descritte nella sezione 72.9).

I modelli di definizione delle variabili sono delimitati solo dallo spazio, dalla virgola, dal punto e virgola o dal punto fermo.

72.1.3.2 Stringhe: «character-string»

«

Nei modelli sintattici, una stringa di caratteri (*character-string*) può essere: un carattere o una sequenza di caratteri contigui, che forma una parola per il linguaggio COBOL; il modello di definizione di una variabili (‘**`PICTURE`**’); un commento. Una stringa di caratteri di questi contesti è delimitata da separatori.

72.1.3.3 Parole

«

Una «parola» per il linguaggio COBOL è una stringa composta al massimo da 30 caratteri, che può essere:

- una parola definita dall’utente, ovvero *user-defined word*;
- un nome di sistema, ovvero *system-name*;
- una parola riservata, ovvero *reserved word*.

Le parole riservate o di sistema non possono essere utilizzate per fini diversi, pertanto non possono essere ridefinite dall’utente.

72.1.3.4 Parole definite dall’utente

«

Una parola definita dall’utente è una parola COBOL che deve essere fornita per soddisfare la sintassi di un’istruzione. Tale parola può essere composta utilizzando soltanto le lettere alfabetiche maiuscole, le cifre numeriche e il trattino (‘**-**’), tenendo conto che il trattino non può trovarsi all’inizio o alla fine di tali parole. Si osservi che in alcuni casi le parole sono costituite esclusivamente da cifre numeriche, mentre in tutti gli altri, le parole devono iniziare con una lettera alfabetica.

Tabella 72.6. Classificazione parziale delle parole definite dall'utente.

Definizione tradizionale	Descrizione
condition-name	Il «nome di condizione» è un nome al quale viene assegnato un valore o un insieme di valori o un intervallo di valori, scelti fra tutti quelli che una variabile può assumere. La variabile stessa viene chiamata «variabile di condizione». I nomi di condizione vengono definiti nella divisione '**DATA DIVISION**'. Un nome di condizione può essere usato solo nelle espressioni condizionali, dove viene trattato come un'abbreviazione di una condizione di relazione. Il valore restituito dal nome di condizione è *Vero* se il valore della variabile di condizione associata è uguale a uno di quei valori che sono stati assegnati al nome di condizione.
data-name	Si tratta del nome di una variabile descritta nella divisione '**DATA DIVISION**'. Una variabile di questo tipo rappresenta normalmente un componente che non può essere suddiviso ulteriormente.
file-name	Si tratta del nome di un file descritto all'interno della divisione '**DATA DIVISION**' e può appartenere sia alla sezione '**FD**' (*File description*), sia alla sezione '**SD**' (*Sort description*).
index-name	Si tratta del nome di un indice associato a una certa tabella, usato per selezionare una voce dalla tabella stessa. Un nome di questo tipo si dichiara nella divisione '**DATA DIVISION**'.

Definizione tradizionale	Descrizione
level-number	Si tratta di un numero che indica la posizione nella struttura gerarchica di un record logico, oppure di un numero speciale che rappresenta convenzionalmente delle proprietà speciali di una variabile. Il numero di livello è espresso esclusivamente con una o due cifre numeriche; inoltre, i numeri che vanno da 01 a 49 indicano la posizione in un record, mentre i numeri 66, 77 e 88 identificano proprietà speciali. Normalmente il numero di livello si scrive sempre utilizzando due cifre, aggiungendo eventualmente uno zero iniziale. Il numero di livello si usa nella divisione '**DATA DIVISION**'.
library-name	Si tratta di un nome che serve a individuare una libreria di sorgenti COBOL, da usare per importare codice contenuto in altri file.
mnemonic-name	Si tratta di un nome che fa riferimento a qualcosa, che dipende dall'ambiente in cui si vuole compilare o eseguire il programma. Questo tipo di parole si usa nella divisione '**ENVIRONMENT DIVISION**', precisamente nel paragrafo '**SPECIAL-NAMES**', con lo scopo di poter sostituire facilmente tali associazioni, senza intervenire in altre parti del programma.
paragraph-name	Si tratta del nome che dichiara l'inizio di un paragrafo nella divisione '**PROCEDURE DIVISION**'.
program-name	Si tratta del nome del programma sorgente, come specificato nella divisione '**IDENTIFICATION DIVISION**'.
record-name	Si tratta del nome di un record di un file. Associando idealmente il file a una tabella di dati, il record equivale alla riga di tale tabella. La dichiarazione dei record avviene nella divisione '**DATA DIVISION**'.
section-name	Si tratta del nome che delimita l'inizio di una sezione nella divisione '**PROCEDURE DIVISION**'.

Definizione tradizionale	Descrizione
text-name	Si tratta del nome di identificazione di un componente all'interno della libreria di sorgenti.

Tutte le parole definite dall'utente, a esclusione dei numeri di livello, possono appartenere soltanto a uno dei vari raggruppamenti previsti e devono essere uniche; tuttavia, in alcuni casi è prevista la possibilità di «qualificare» dei nomi, che non sono univoci, in modo da attribuirli al loro contesto preciso (sezione 72.8.3).

72.1.3.5 Parole riservate

Le parole riservate sono quelle parole del linguaggio che fanno parte di un elenco prestabilito e che hanno un significato speciale. Queste parole sono classificate in gruppi in base al loro utilizzo.

Tabella 72.7. Classificazione sintetica delle parole riservate.

Classificazione	Descrizione
parole chiave	Una parola chiave è una parola riservata la cui presenza è richiesta all'interno di un'istruzione (al contrario di altre che possono essere usate soltanto per migliorare l'estetica o la leggibilità delle istruzioni). Le parole chiave devono essere inserite per specificare quel tipo di istruzione.
parole opzionali	Una parola opzionale è una parola riservata facoltativa, che si può usare nelle istruzioni per facilitarne la lettura. La parola opzionale non è obbligatoria, ma se usata va applicata secondo la sintassi prevista.

Classificazione	Descrizione
registri speciali	Una registro speciale identifica un'area di memoria con funzioni speciali. I registri speciali dipendono generalmente dalle caratteristiche del compilatore e non sono standard.
costanti figurative	Una costante figurativa è un nome che identifica un certo valore costante, come alternativa alla rappresentazione letterale.
parole di caratteri speciali	Alcuni caratteri speciali, nell'ambito del contesto appropriato, possono essere rappresentati attraverso parole particolari. Si tratta precisamente degli operatori di relazione.

72.1.3.6 Costanti figurative

«

Per fare riferimento a valori costanti specifici si possono usare alcune parole riservate, note come costanti figurative. Di queste parole chiave esistono sia versioni al singolare, sia al plurale, ma rappresentano sempre la stessa cosa, ovvero un valore singolo o un valore ripetuto, in base al contesto.

Tabella 72.8. Costanti figurative.

Nome	Descrizione
`ZERO` `ZEROS` `ZEROES`	Rappresenta il valore numerico zero o la stringa '**0**' ripetuta più volte.

Nome	Descrizione
`SPACE` `SPACES`	Rappresenta uno o più spazi bianchi.
`HIGH-VALUE` `HIGH-VALUES`	Rappresenta uno o più caratteri con un «valore massimo», in base a qualche criterio, legato alla sequenza di collazione (*collating sequence*) o alla codifica. Generalmente si tratta del valore FF_{16}.
`LOW-VALUE` `LOW-VALUES`	Rappresenta uno o più caratteri con un «valore minimo», in base a qualche criterio, legato alla sequenza di collazione (*collating sequence*) o alla codifica. Generalmente si tratta del valore 00_{16}.
`QUOTE` `QUOTES`	Rappresenta una o più virgolette. Questa costante figurativa non può sostituire le virgolette che delimitano le costanti alfanumeriche.
`ALL` ***valore***	Rappresenta la ripetizione indefinita del valore indicato. Tale valore può essere specificato anche attraverso una costante letterale o una costante figurativa.

72.1.3.7 Parole di caratteri speciali

Gli operatori di relazione si possono rappresentare con i simboli previsti (‘**<**’, ‘**>**’, e ‘**=**’) oppure attraverso parole speciali, ovvero «parole di caratteri speciali», note come *special character word*. La tabella successiva riepiloga l’uso degli operatori di relazione, in tutte le loro forme.

Tabella 72.9. Modelli sintattici per l'uso degli operatori di relazione.

Operatore	Descrizione
`IS [NOT] GREATER THEN` `---  -------`	maggiore di, non maggiore di
`IS [NOT] >` `---  -`	maggiore di, non maggiore di
`IS [NOT] LESS THEN` `---  ----`	minore di, non minore di
`IS [NOT] <` `---  -`	minore di, non minore di
`IS [NOT] EQUAL TO` `---  -----`	uguale a, diverso da
`IS [NOT] =` `---  -`	uguale a, diverso da
`IS GREATER THAN OR EQUAL TO` `-------  -- -----`	maggiore o uguale a
`IS >=` `--`	maggiore o uguale a
`IS LESS THAN OR EQUAL TO` `----  -- -----`	minore o uguale a
`IS <=` `--`	minore o uguale a

72.1.3.8 Rappresentazione delle costanti

«

Le costanti possono essere stringhe di caratteri, il cui valore è implicito in base ai caratteri di cui sono composte, oppure sono costanti figurative, che rappresentano un valore in base al significato verbale che hanno. **Una costante può essere di tipo numerico o alfanumerico e non sono previsti altri tipi.**

Una costante numerica letterale è una stringa composta da cifre numeriche ed eventualmente anche dai segni '**+**', '**−**' e dal punto per la separazione tra la parte intera e la parte decimale (a meno che il punto sia da sostituire con la virgola, avendone scambiato le funzionalità con un'istruzione apposita). Una costante numerica deve contenere almeno una cifra e ha una dimensione massima di cifre che dipende dal compilatore.

Una costante numerica non può contenere più di un segno. Se viene usato il segno, questo deve collocarsi nella posizione più a sinistra; se non appare alcun segno, il valore si intende positivo.

Una costante numerica non può contenere più di un punto decimale e può apparire in qualunque posizione. Se non viene usato il punto decimale, la costante rappresenta un numero intero.

Se nel paragrafo '**SPECIAL-NAMES**' della divisione '**ENVIRONMENT DIVISION**' è specificata la dichiarazione '**DECIMAL-POINT IS COMMA**', la rappresentazione dei valori numerici avviene scambiando il significato del punto e della virgola (in pratica secondo le convenzioni europee).

Una costante alfanumerica è una stringa di caratteri delimitata da virgolette. La stringa può contenere qualsiasi carattere previsto dalla codifica utilizzata dal compilatore; in generale è ammesso almeno l'uso delle lettere minuscole dell'alfabeto latino.

Per rappresentare le virgolette ('**"**') all'interno di una stringa si usa il concatenamento con la costante figurativa '**QUOTE**', come nell'esempio seguente:

```
000000 DISPLAY "Il file ", QUOTE, "mio.txt", QUOTE,
000000         " e` impegnato!".
```

Una costante alfanumerica deve contenere almeno un carattere all'interno delle virgolette. La lunghezza massima di un valore alfanumerico dipende dal compilatore, ma in generale dovrebbe essere garantita la rappresentazione di almeno 200 caratteri.

72.1.4 Notazione sintattica

«

I manuali COBOL adottano generalmente una forma particolare di notazione per descriverne la sintassi, a cui si adegua anche questo.

Nella sintassi le «parole chiave», secondo la definizione del COBOL, sono rappresentate sottolineate, a indicare la loro obbligatorietà, mentre le parole facoltative non sono sottolineate. Nell'esempio seguente, le parole '**IF**', '**NOT**', '**NUMERIC**' e '**ALPHABETIC**' sono parole chiave, mentre la parola '**IS**' è facoltativa:

```
                          /             \
                          | NUMERIC     |
IF identifier IS [NOT]    < -------     >
--                 ---    | ALPHABETIC  |
                          \ ----------- /
```

Tutte le parole scritte con lettere minuscole rappresentano delle metavariabili sintattiche che devono essere espresse dal programmatore in quella posizione. Nell'esempio precedente appare una sola metavariabile denominata '**identifier**'.

Le parentesi graffe servono a rappresentare la scelta tra alternative differenti. Nell'esempio precedente si deve scegliere tra due parole chiave: '**NUMERIC**' o '**ALPHABETIC**'.

Le parentesi quadre rappresentano parti opzionali di un'istruzione; tuttavia si osservi che non si tratta di «parole facoltative», secondo

la definizione del linguaggio COBOL, perché l'uso o meno di tali porzioni di codice implica un risultato differente dell'istruzione.

La presenza di tre punti consecutivi indica che i dati che precedono la notazione possono essere ripetuti successivamente, in funzione delle esigenze del problema che si intende risolvere.

```
MOVE identifier-1 TO identifier-2 ...
----              --
```

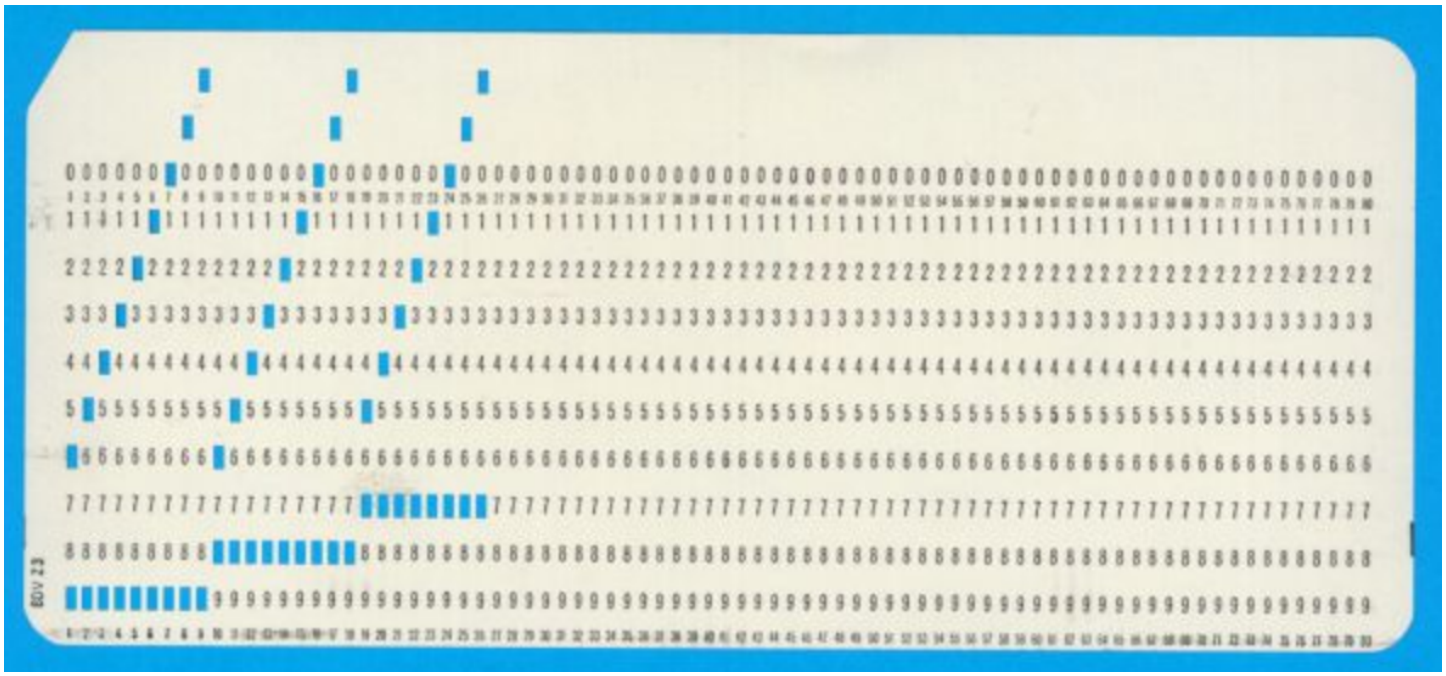
Nell'esempio mostrato, i puntini di sospensione indicano che si possono inserire più variabili (precisamente ciò che è rappresentato come ***identifier-2***). In questo caso, il contenuto della prima variabile viene copiato all'interno di tutte quelle che sono annotate dopo la parola chiave '**`TO`**'.

Quando appare il punto fermo nello schema sintattico, l'istruzione reale deve contenerlo nella stessa posizione relativa.

72.2 Modulo di programmazione

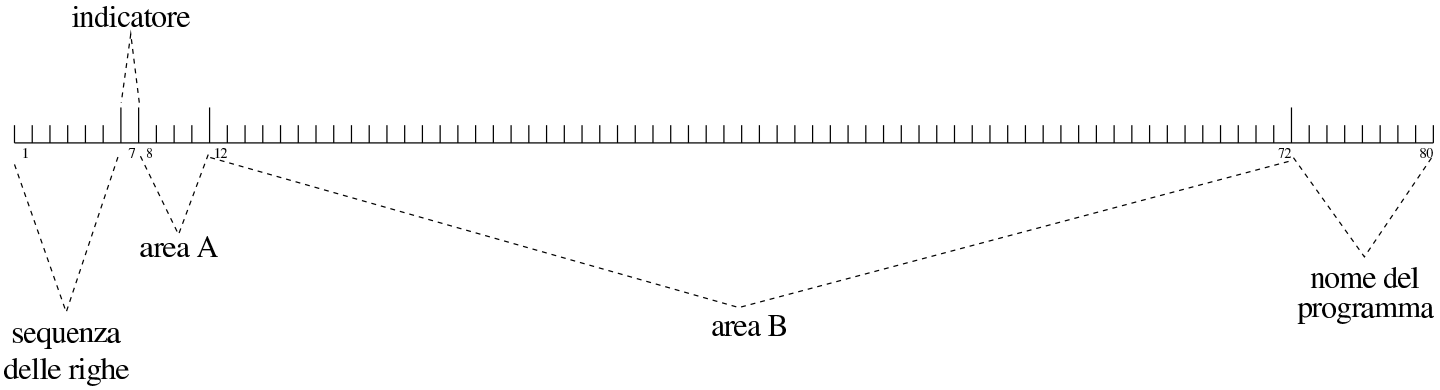
Il linguaggio COBOL nasce quando l'inserimento dei dati in un elaboratore avveniva principalmente attraverso schede perforate, pertanto, da questo derivano delle limitazioni nel modo in cui vanno scritte le sue istruzioni.

Figura 72.13. La scheda perforata classica, da 80 colonne.



Il modulo di programmazione (*coding form*) era un foglio quadrettato che conteneva la guida per la scrittura di un programma, da passare poi a una persona che si incaricava di perforare le schede, copiando il testo di tale modulo. Attualmente strumenti del genere non si usano più, tuttavia occorre sapere che le direttive vanno scritte in uno spazio di colonne prestabilito.

Figura 72.14. Suddivisione delle colonne.



In pratica, il codice COBOL si scrive in un file di testo di 80 colonne, rispettando le convenzioni descritte nella tabella successiva.

Tabella 72.15. Colonne riservate nel codice COBOL.

Colonne	Utilizzo
1..6	Le prime sei colonne servono a indicare un numero di sequenza delle righe. Il numero può essere discontinuo, purché progressivo. Generalmente si utilizzava una sequenza maggiore dell'unità, per consentire l'inserzione successiva di righe ulteriori, che si sarebbero tradotte nell'aggiunta di schede, senza dover perforare nuovamente tutto.
7	La settima colonna serve a inserire un simbolo «indicatore». Generalmente si tratta dell'asterisco, per specificare che si tratta di una riga di commento, del trattino per la continuazione delle stringhe, oppure di una barra obliqua per richiedere un salto pagina in fase di stampa del sorgente.
8..11	Le colonne dall'ottava all'undicesima rappresentano l'«area A», nella quale devono iniziare le dichiarazioni più importanti.
12..72	Le colonne dalla dodicesima alla settantaduesima rappresentano l'«area B», nella quale si mettono tutte le direttive che non possono partire dall'area A.
73..80	Le ultime otto colonne sono riservate per inserire un'etichetta facoltativa di identificazione del programma.

72.2.1 Indicatore

La settima colonna serve per diverse funzioni, distinte in base alla presenza di un simbolo speciale; se in questa colonna si trova uno spazio, la riga viene usata per le funzioni normali. La tabella successiva riepiloga i simboli che possono apparire nella settima colonna e come questi dovrebbero essere interpretati dal compilatore.

Indicatore	Descrizione
`$`	Il dollaro viene usato per specificare delle opzioni in fase di compilazione.
`*`	L'asterisco stabilisce che, la riga in cui appare, contiene un commento che il compilatore deve ignorare completamente. Il commento può essere collocato solo nello spazio complessivo dell'area A e B, ovvero dalla colonna 8 alla colonna 72.
`/`	La barra obliqua serve a richiedere un salto pagina quando gli strumenti di compilazione vengono usati per stampare il sorgente. Ciò che dovesse apparire nell'area A e B di una riga che ha la barra obliqua nella settima colonna viene considerato come un commento.
`D`	La lettera '**`D`**' serve a indicare al compilatore che la riga in questione deve essere presa in considerazione solo se l'opzione '**`WITH DEBUGGING`**' viene utilizzata nel paragrafo '**`SOURCE COMPUTER`**'; in caso contrario la riga deve essere trattata come un commento.
`-`	Un trattino indica che, sulla riga precedente, l'ultima parola o costante non è completa, ma continua sulla riga in cui appare il trattino stesso.

Per quanto riguarda la continuazione di parole e di costanti numeriche su più righe, il troncamento può avvenire in qualsiasi punto, mettendo un trattino nella settima colonna della riga successiva, continuando lì la parola o la costante, a partire dalla colonna 12 fino alla colonna 72 (area B). Gli spazi finali nella riga interrotta e quelli iniziali della riga che riprende, vengono ignorati.

Le costanti alfanumeriche delimitate da virgolette, si separano in modo differente. Sulla riga spezzata, si considerano tutte le informazioni dalle virgolette di apertura fino alla colonna 72 inclusa, mentre

nella riga successiva, la costante deve riprendere aggiungendo altre virgolette di apertura.

Si osservi che ci sono compilatori che si limitano a riconosce solo l'asterisco per i commenti, ignorando tutto il resto. Per questo motivo, è bene evitare l'uso di ogni altro simbolo in questa colonna, quando si vuole scrivere un programma abbastanza compatibile, tenendo conto che si può evitare la continuazione nella riga successiva, perché le istruzioni possono collocarsi su più righe senza spezzare le parole, mentre le costanti alfanumeriche si possono dividere in porzioni più piccole da concatenare.

72.2.2 Area A e area B

Le intestazioni dei paragrafi, delle sezioni e delle divisioni devono iniziare nell'area A. L'intestazione di una divisione consiste nel nome della divisione ('**IDENTIFICATION**', '**ENVIRONMENT**', '**DATA**' o '**PROCEDURE**'), seguito da uno spazio bianco e dalla parola '**DIVISION**', seguita a sua volta da un punto fermo. L'intestazione di una sezione consiste di un nome di sezione seguito da uno spazio bianco e dalla parola '**SECTION**', seguita a sua volta da un punto fermo. L'intestazione di un paragrafo consiste di un nome di paragrafo seguito da un punto fermo e da uno spazio bianco; il primo gruppo di istruzioni del paragrafo può apparire anche sulla stessa riga.

All'interno delle divisioni '**IDENTIFICATION DIVISION**' e '**ENVIRONMENT DIVISION**', le sezioni e i paragrafi sono fissi e sono ammessi solo i nomi previsti espressamente, mentre nella divi-

sione '**PROCEDURE DIVISION**' i nomi dei paragrafi e delle sezioni sono stabiliti liberamente.

All'interno della divisione '**DATA DIVISION**', le sezioni '**FD**' e '**SD**', così come i numeri di livello 01 e 77, devono iniziare nell'area A, mentre gli altri numeri di livello devono iniziare nell'area B.

Nell'area B inizia tutto quello che non può iniziare nell'area A.

72.2.3 Interpunzione

«

La scrittura di un programma COBOL è sottoposta alle regole seguenti che riguardano l'uso dei caratteri di interpunzione.

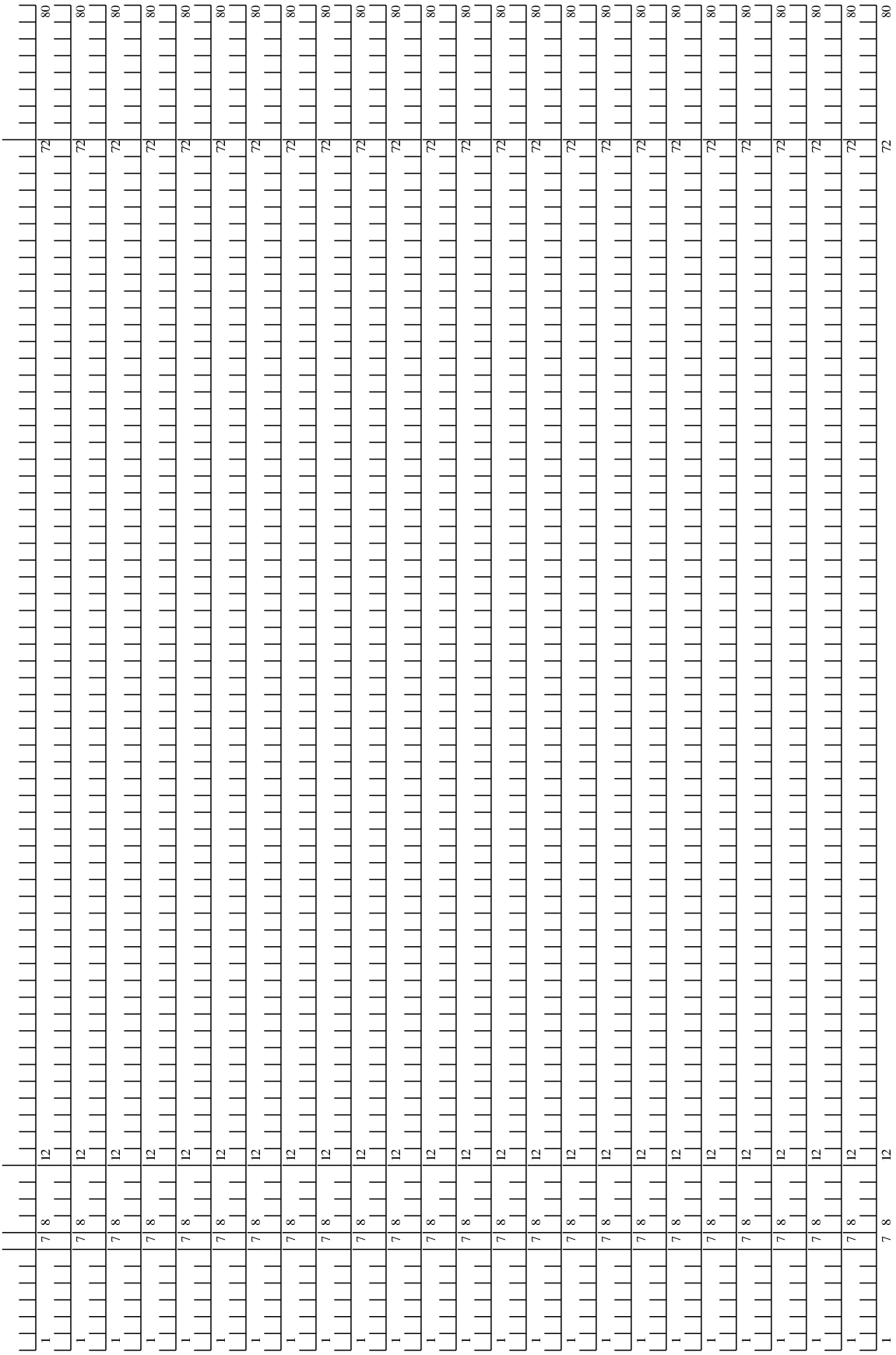
- Un gruppo di istruzioni termina con un punto seguito da uno spazio bianco. Un punto può apparire in un'altra posizione solo se fa parte di una costante alfanumerica, se si tratta del punto decimale di una costante numerica o se viene usato in un modello di definizione di una variabile ('**PICTURE**').
- Una virgola può essere usata fra le istruzioni per facilitare la leggibilità del programma; diversamente, una virgola può apparire solo dove indicato nello schema sintattico. **L'uso delle virgole non è obbligatorio.**
- Il punto e virgola può essere usato al posto della virgola.
- Uno spazio delimita sempre una parola o una costante, a meno che tale spazio sia parte di una costante alfanumerica. Lo spazio inteso come delimitatore può essere ridondante; inoltre, quando il testo di un'istruzione termina esattamente alla fine dell'area B (colonna 72), lo spazio successivo viene a mancare.

Listato 72.17. Un esempio preso da un programma sorgente, dove si mette in evidenza come si continua una parola o una costante alfanumerica nella riga successiva.

```
000100 IDENTIFICATION DIVISION.                                         ESEMPIO0
000200                                                                  ESEMPIO0
000300 PROGRAM-ID.   ESEMPIO0.                                          ESEMPIO0
000400 AUTHOR.       DANIELE GIACOMINI.                                 ESEMPIO0
000500 DATE-WRITTEN. 2005-02-13.                                        ESEMPIO0
000600                                                                  ESEMPIO0
000700                                                                  ESEMPIO0
000800 ENVIRONMENT DIVISION.                                            ESEMPIO0
000900                                                                  ESEMPIO0
001000 INPUT-OUTPUT SECTION.                                            ESEMPIO0
001100                                                                  ESEMPIO0
001200 FILE-CONTROL.                                                    ESEMPIO0
001300                                                                  ESEMPIO0
001400     SELECT FILE-DA-LEGGERE ASSIGN TO DISK                        ESEMPIO0
001500                            ORGANIZATION IS SEQUENTIAL.           ESEMPIO0
001600                                                                  ESEMPIO0
001700 DATA DIVISION.                                                   ESEMPIO0
001800                                                                  ESEMPIO0
001900 FILE SECTION.                                                    ESEMPIO0
002000                                                                  ESEMPIO0
002100 FD  FILE-DA-LEGGERE                                              ESEMPIO0
002200     LABEL RECORD IS STANDARD                                     ESEMPIO0
002300     VALUE OF FILE-ID IS "input.txt".                             ESEMPIO0
002400                                                                  ESEMPIO0
002500 01  RECORD-DA-LEGGERE PIC X(30).                                 ESEMPIO0
002600                                                                  ESEMPIO0
002700 WORKING-STORAGE SECTION.                                         ESEMPIO0
002800 01  EOF               PIC 9   VALUE ZERO.                       ESEMPIO0
002900/                                                                 ESEMPIO0
003000 PROCEDURE DIVISION.                                              ESEMPIO0
003100*                                                                 ESEMPIO0
003200* Qui inizia il paragrafo "MAIN".                                 ESEMPIO0
003300*                                                                 ESEMPIO0
003400 MAIN.                                                            ESEMPIO0
003500     OPEN INPUT FILE-DA-LEG                                       ESEMPIO0
003600-               GERE.                                             ESEMPIO0
003700     READ FILE-DA-LEG                                             ESEMPIO0
003800-         GERE                                                    ESEMPIO0
003900          AT END                                                  ESEMPIO0
```

```
004000                MOVE 1 TO EOF.                                    ESEMPIO0
004100     DISPLAY "Ho aperto il file input.txt e sto per emettere il suESEMPIO0
004200-            "o contenuto sullo schermo:".                        ESEMPIO0
004300     PERFORM LETTURA UNTIL EOF = 1.                               ESEMPIO0
004400     CLOSE FILE-DA-LEGGERE.                                       ESEMPIO0
004500                                                                  ESEMPIO0
004600     STOP RUN.                                                    ESEMPIO0
004700*                                                                 ESEMPIO0
004800* Qui inizia un altro paragrafo.                                  ESEMPIO0
004900*                                                                 ESEMPIO0
005000 LETTURA.                                                         ESEMPIO0
005100     DISPLAY RECORD-DA-LEGGERE.                                   ESEMPIO0
005200     READ FILE-DA-LEGGERE                                         ESEMPIO0
005300          AT END                                                  ESEMPIO0
005400               MOVE 1 TO EOF.                                     ESEMPIO0
005500/                                                                 ESEMPIO0
```

Figura 72.18. Esempio di un modulo di programmazione COBOL.



72.3 Divisione «IDENTIFICATION DIVISION»

La divisione '**IDENTIFICATION DIVISION**' costituisce la prima parte di un programma COBOL. Il suo scopo è quello di contenere delle informazioni sul programma, secondo una classificazione ben stabilita. Le informazioni tipiche che si inseriscono in questa divisione sono il nome del programma (nome che non coincide necessariamente con il nome del file che contiene il sorgente), il nome dell'autore, la data di scrittura del programma, la data di compilazione.

72.3.1 Struttura

La struttura della divisione '**IDENTIFICATION DIVISION**' è sintetizzabile nello schema sintattico seguente:

```
 IDENTIFICATION DIVISION.
 ------------------------

[PROGRAM-ID. program-name].
 ----------

[AUTHOR. [comment-entry]...].
 ------

[INSTALLATION. [comment-entry]...].
 ------------

[DATE-WRITTEN. [comment-entry]...].
 ------------

[DATE-COMPILED. [comment-entry]...].
 -------------

[SECURITY. [comment-entry]...].
 --------
```

La divisione deve iniziare scrivendo '**IDENTIFICATION DIVISION**' a partire dall'area A, ricordando di aggiungere il punto fermo finale.

Tutti i nomi di paragrafo di questa divisione devono iniziare nell'area A e devono terminare con un punto fermo.

Il nome del programma (***program-name***) deve essere una parola COBOL e serve a identificare il programma sorgente, ma non corrisponde necessariamente al nome del file su disco che contiene il sorgente.

Le voci di commento (***comment-entry***), secondo lo schema sintattico, possono essere costituite da una sequenza qualunque di caratteri e possono occupare anche più righe, senza bisogno di indicare il simbolo di continuazione nella settima colonna, avendo cura però di utilizzare per tali voci solo l'area B e di terminarle comunque con un punto fermo.

La data di compilazione è, o dovrebbe essere, posta automaticamente dal compilatore, quando è prevista la stampa del sorgente da parte di questo strumento.

A parte il caso della data di compilazione, che dovrebbe essere fornita dal compilatore, tutte le altre informazioni rimangono invariate.

72.3.2 Codifica della divisione

Il listato successivo dà un'idea di come può essere codificata la divisione '**`IDENTIFICATION DIVISION`**'.

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.    PROVA-INTESTAZIONE.
000300 AUTHOR.        DANIELE GIACOMINI.
000400 INSTALLATION.  NANOLINUX IV,
000500                TINYCOBOL 0.61,
000600                OPENCOBOL 0.31.
000700 DATE-WRITTEN.  2005-02-14.
```

```
000800 DATE-COMPILED.
000900 SECURITY.      SEGRETISSIMO, LIVELLO III.
001000*
001100 ENVIRONMENT DIVISION.
001200 DATA DIVISION.
001300 PROCEDURE DIVISION.
001400 MAIN.
001500     DISPLAY "CIAO A TUTTI!".
001600     STOP RUN.
```

72.4 Divisione «ENVIRONMENT DIVISION»

«

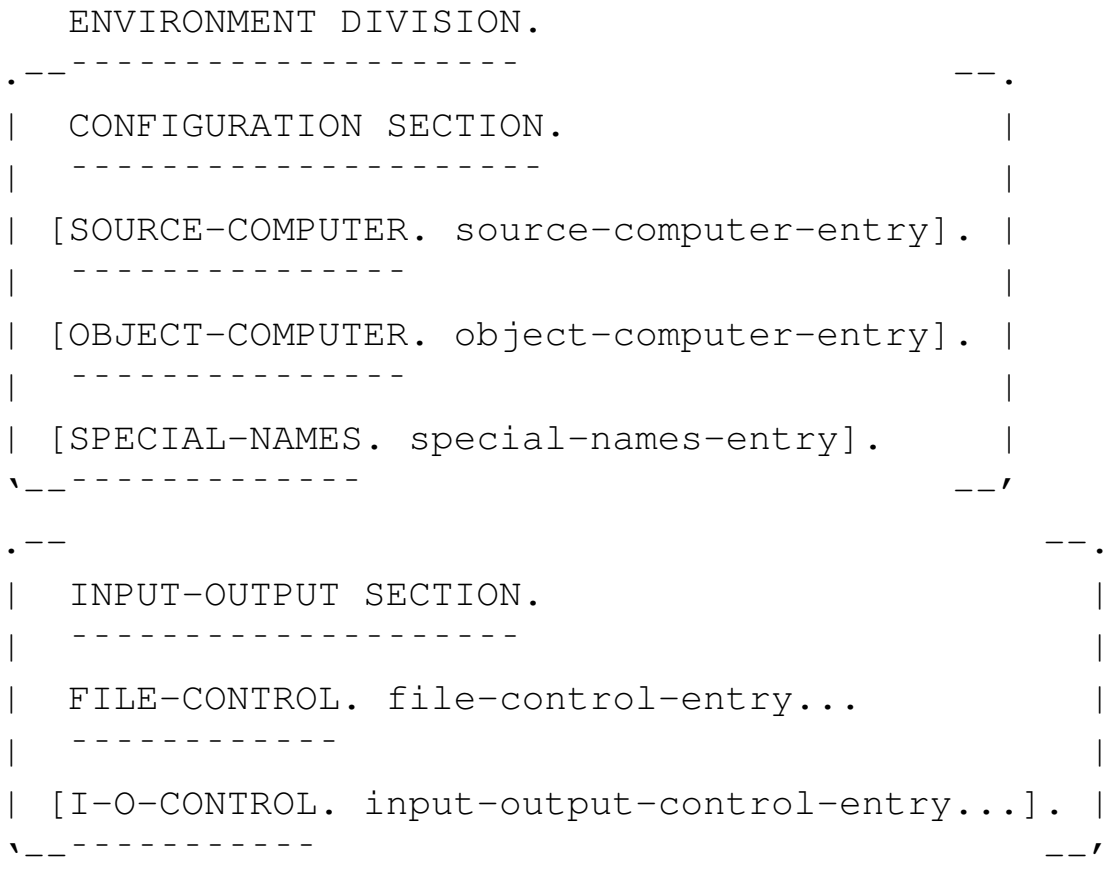
La divisione '**ENVIRONMENT DIVISION**' costituisce la seconda parte di un programma COBOL. La divisione si compone di due sezioni: '**CONFIGURATION SECTION**' e '**INPUT-OUTPUT SECTION**'.

La sezione '**CONFIGURATION SECTION**' serve per indicare delle informazioni relative all'elaboratore usato per la compilazione del programma sorgente e a quello nel quale deve essere eseguito il programma, una volta compilato; inoltre, questa sezione permette di stabilire delle sostituzioni, come nel caso della virgola al posto del punto per separare la parte intera di un numero dalla parte decimale.

La sezione '**INPUT-OUTPUT SECTION**' serve per associare i file usati dal programma con le unità fisiche relative, a indicare le caratteristiche di tali file e a stabilire altri aspetti dello scambio di dati.

72.4.1 Struttura

La struttura della divisione '**ENVIRONMENT DIVISION**' è sintetizzabile nello schema sintattico seguente:



72.4.2 Sezione «CONFIGURATION SECTION»

La sezione '**CONFIGURATION SECTION**' contiene le informazioni sul sistema usato per la compilazione del programma (nel paragrafo '**SOURCE-COMPUTER**'), il sistema nel quale il programma deve essere eseguito (nel paragrafo '**OBJECT-COMPUTER**') e il paragrafo '**SPECIAL-NAMES**' che consente di effettuare alcune sostituzioni a dei valori che altrimenti resterebbero al loro stato predefinito.

```
 CONFIGURATION SECTION.
 ----------------------

[SOURCE-COMPUTER. source-computer-entry].
 ---------------

[OBJECT-COMPUTER. object-computer-entry].
 ---------------

[SPECIAL-NAMES. special-names-entry].
 -------------
```

72.4.2.1 Paragrafo «SOURCE-COMPUTER»

«

Il paragrafo '**SOURCE-COMPUTER**' identifica l'elaboratore presso il quale si intende compilare il programma. Si utilizza secondo lo schema sintattico seguente:

```
SOURCE-COMPUTER. computer-name [WITH DEBUGGING MODE].
---------------                --------------
```

Al posto della metavariabile ***computer-name*** deve essere indicata una parola COBOL, che serve solamente a titolo informativo nel sorgente. Se si specifica l'opzione '**DEBUGGING MODE**' si richiede al compilatore di prendere in considerazione, nel sorgente, tutte le righe annotate con la lettera '**D**' nella settima colonna e le istruzioni '**USE FOR DEBUGGING**', che altrimenti verrebbero semplicemente ignorate.

72.4.2.2 Paragrafo «OBJECT-COMPUTER»

«

Il paragrafo '**OBJECT COMPUTER**' identifica l'elaboratore presso il quale deve essere utilizzato il programma, una volta compilato. Lo schema sintattico per l'utilizzo di questo paragrafo è quello seguente:

```
OBJECT-COMPUTER. computer-name... .
----------------
```

Il nome dell'elaboratore (*computer name*) deve essere una parola COBOL e ha un significato puramente informativo. Alla fine dell'indicazione dell'ultimo nome, deve apparire un punto fermo.

72.4.2.3 Paragrafo «SPECIAL-NAMES»

Il paragrafo '**SPECIAL-NAMES**' serve ad associare un valore a dei nomi prestabiliti, quando si vuole che la funzione loro associata sia diversa da quella predefinita, oppure ad attribuire un «nome mnemonico» a un nome usato dal compilatore, che però non fa parte dello standard. Le dichiarazioni che possono apparire in questo paragrafo dipendono molto dalle caratteristiche del compilatore; quello che si vede nello schema sintattico seguente è il minimo che dovrebbe essere disponibile nella maggior parte dei casi:

```
SPECIAL-NAMES.
-------------

    implementor-name IS mnemonic-name
                     --

    [CURRENCY SIGN IS literal]
     --------      --

    [DECIMAL-POINT IS COMMA].
     -------------    -----
```

Si utilizza la dichiarazione '**CURRENTY SIGN IS**' per fissare il simbolo predefinito da usare come segno di valuta; si usa la dichiarazione '**DECIMAL-POINT IS COMMA**' per rappresentare i valori numerici secondo la forma europea, dove la virgola indica la separazione tra la parte intera e quella decimale.

Il segno di valuta può essere costituito da un solo carattere e sono molto pochi i simboli che si possono usare. Per la precisione, sono esclusi tutti i simboli che invece possono essere usati nei modelli di definizione delle variabili oltre a quelli che si usano come delimitatori. In linea di massima sono da escludere: tutte le cifre numeriche (da '**0**' a '**9**'); lo spazio; le lettere alfabetiche '**A**', '**B**', '**C**', '**D**', '**J**', '**L**', '**N**', '**P**', '**R**', '**S**', '**V**', '**X**', '**Z**'; i caratteri speciali '*****', '**+**', '**-**', '**,**', '**.**', '**;**', '**%**', '**(**', '**)**', '**"**', '**?**'.

Si osservi che anche nel modello di definizione di una variabile ('**PICTURE**'), quando si usa la dichiarazione '**DECIMAL-POINT IS COMMA**', il punto e la virgola si scambiano i ruoli.

L'esempio seguente mostra un pezzo di programma in cui si vede l'uso di queste opzioni. Per la precisione, si assegna la lettera «E» per rappresentare la valuta:

```
000000 ENVIRONMENT DIVISION.
000000 CONFIGURATION SECTION.
000000 SPECIAL-NAMES. DECIMAL-POINT IS COMMA
000000                CURRENCY SIGN IS "E".
```

L'attribuzione di un nome mnemonico a una parola non standard che però fa parte delle funzionalità specifiche del compilatore utilizzato, consente di limitare a questa sezione le modifiche per l'adattamento del programma a un compilatore che ha funzioni simili, ma descritte da parole diverse. Nell'esempio seguente, compilabile con OpenCOBOL, si sostituisce la parola '**CONSOLE**' con '**STANDARD-INPUT**', per identificare la fonte dei dati in ingresso per l'istruzione '**ACCEPT**':

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   TEST-ACCEPT.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-02-27.
000500*
000600 ENVIRONMENT DIVISION.
000700 CONFIGURATION SECTION.
000800 SOURCE-COMPUTER.
000900     OPENCOBOL.
001000 SPECIAL-NAMES.
001100     CONSOLE IS STANDARD-INPUT.
001200*
001300 DATA DIVISION.
001400*
001500 WORKING-STORAGE SECTION.
001600 77  MESSAGGIO            PIC X(30).
001700*
001800 PROCEDURE DIVISION.
001900*
002000 MAIN.
002100     DISPLAY "INSERISCI IL MESSAGGIO".
002200     ACCEPT MESSAGGIO FROM STANDARD-INPUT.
002300     DISPLAY "HAI INSERITO: ", MESSAGGIO.
002400*
002500     STOP RUN.
002600*
```

Nell'esempio appena mostrato sono evidenziate le righe più importanti per la comprensione del meccanismo; si può comprendere che l'istruzione '**ACCEPT**' avrebbe potuto essere scritta semplicemente così:

```
002200     ACCEPT MESSAGGIO FROM CONSOLE.
```

Tuttavia, avendo utilizzato il nome mnemonico '**STANDARD-INPUT**', se con un altro compilatore la console fosse identificata dalla sigla '**SPO**' (*Supervisory printer output*, come avveniva nel COBOL CMS (*Computer management system* della Burroughs negli anni 1980), basterebbe modificare la dichiarazione iniziale:

```
001000 SPECIAL-NAMES.
001100     SPO IS STANDARD-INPUT.
```

Per chiarezza, è il caso di sottolineare che '**STANDARD-INPUT**' ha valore per il compilatore solo in quanto viene dichiarato come nome mnemonico, dal momento che il linguaggio, nella sua veste ufficiale, non prevede la gestione dei flussi standard dei sistemi Unix.

72.4.3 Sezione «INPUT-OUTPUT SECTION»

«

La sezione '**INPUT-OUTPUT SECTION**' si suddivide in due paragrafi: '**FILE-CONTROL**' e '**I-O-CONTROL**'. Il paragrafo '**FILE-CONTROL**' specifica l'organizzazione e l'accesso dei file che vengono usati dal programma e le informazioni correlate a tali file; il paragrafo '**I-O-CONTROL**' serve a specificare informazioni aggiuntive sui file già dichiarati nell'altro paragrafo.

```
 INPUT-OUTPUT SECTION.
 --------------------

 FILE-CONTROL. file-control-entry...
 ------------

[I-O-CONTROL. input-output-control-entry...].
 -----------
```

72.4.3.1 Paragrafo «FILE-CONTROL»

Il paragrafo '**FILE-CONTROL**' serve a dichiarare i file utilizzati dal programma e a definire alcune loro caratteristiche. Tutti i file dichiarati nel paragrafo '**FILE-CONTROL**' devono essere descritti nella divisione '**DATA DIVISION**'; nello stesso modo, tutti i file descritti nella divisione '**DATA DIVISION**', devono essere dichiarati nel paragrafo '**FILE-CONTROL**'.

Il linguaggio COBOL prevede una gestione dei file molto sofisticata, anche se non è detto che i compilatori mettano a disposizione sempre tutte le funzionalità standard. Si distinguono generalmente i tipi, in base alla loro «organizzazione», come sintetizzato nella tabella successiva.

Per il linguaggio COBOL i file sono sempre composti da record, pertanto l'accesso a un file si riferisce sempre a dei record.

Tabella 72.31. Classificazione dei file in base all'organizzazione.

Organizzazione	Descrizione
sequenziale	Il file sequenziale consente un accesso ai record in modo seriale, dal primo all'ultimo. Generalmente, si dichiara un accesso sequenziale ai file quando l'unità di memorizzazione nella quale sono memorizzati è per sua natura sequenziale, come per i nastri magnetici.
relativa (*relative*)	Si tratta di un file ad accesso diretto, dove i record si possono raggiungere specificandone il numero, a partire da uno, avendo anche la possibilità di richiedere qualche spostamento relativo rispetto al record attuale.

Organizzazione	Descrizione
a indice	Si tratta di un file associato a un indice dei record. Attraverso l'indice è possibile raggiungere direttamente i record associati, senza bisogno di eseguire delle scansioni di ricerca.

L'organizzazione del file definisce le potenzialità di accesso, ma in generale sono disponibili diverse varianti nel modo particolare di accedere ai record.

Il paragrafo '**FILE CONTROL**' si articola in dichiarazioni '**SELECT**', una per ogni file, secondo lo schema sintattico sintetico seguente:

```
FILE-CONTROL.
-------------

    SELECT file-name ASSIGN TO hardware-name [altre-opzioni].
    ------           ------

    ...
```

Il modo in cui l'istruzione '**SELECT**' si articola, dipende dall'organizzazione del file e dal metodo di accesso specifico che si vuole attuare sullo stesso. Nella logica originale del linguaggio, in questa fase non viene ancora indicato il nome del file reale, secondo il sistema operativo, perché generalmente per questa informazione si agisce nella divisione '**DATA DIVISION**'; tuttavia, spesso il compilatore permette, o richiede, di specificare il nome del file reale proprio nell'istruzione '**SELECT**'.

72.4.3.2 File fisici e file «logici»

L'organizzazione di un file è una caratteristica immutabile, che stabilisce, oltre che le potenzialità di accesso, anche la sua forma fisica «reale», ovvero quella che viene gestita attraverso l'astrazione del sistema operativo.

L'organizzazione sequenziale è quella più semplice, dove normalmente i record logici del linguaggio corrispondono esattamente al contenuto del file fisico che li contiene.

L'organizzazione relativa richiede la capacità di abbinare delle informazioni ai record logici, per esempio per poter annotare che un record è stato cancellato. Per fare questo, il compilatore può inserire tutte le informazioni necessarie in un file solo, oppure può avvalersi di due file reali: uno per i dati, l'altro per le informazioni sui record.

L'organizzazione a indice richiede tutte le funzionalità di quella relativa, con l'aggiunta di poter gestire l'accesso in base a una o più chiavi. Nei compilatori COBOL attuali, è molto probabile che tutte le informazioni necessarie vengano gestite in un file fisico soltanto, ma originariamente era frequente l'uso di un file per i dati e di altri file per le chiavi (uno per ogni chiave).

In base a questa premessa, si deve intendere che un file che viene creato con una certa organizzazione, può essere usato solo con quella; inoltre, si può contare sul fatto che un file creato con un programma realizzato attraverso un certo compilatore COBOL, non può essere utilizzato con un programma generato con un altro.

Di fronte a questo problema di compatibilità dei dati, i file organizzati in modo sequenziale sono sempre l'unica garanzia per un trasferimento dei dati. D'altra parte, negli anni in cui il linguaggio CO-

BOL aveva il suo massimo splendore, i nastri magnetici rappresentavano l'unità di memorizzazione «standard» tra le varie architetture proprietarie.

72.4.3.3 Istruzione «SELECT» per i file sequenziali

«

Lo schema sintattico semplificato per l'istruzione '**SELECT**', da usare nel paragrafo '**FILE-CONTROL**', per dichiarare un file sequenziale è quello che si può vedere nella figura successiva:

```
                                /                    \
                                |   hardware-name    |
SELECT file-name ASSIGN TO      <                    >
------           ------         | literal-file-name  |
                                \                    /

    .--                 .--    --. --.
    |                   | AREA  |   |
    | RESERVE integer   |       |   |
    | -------           | AREAS |   |
    `--                 `--    --' --'

    [ ORGANIZATION IS [LINE] SEQUENTIAL ]
      ------------     ----  ----------

    [ ACCESS MODE IS SEQUENTIAL ]
      ------      ----------

    [ FILE STATUS IS data-name ].
          ------
```

Il file sequenziale può essere letto o scritto soltanto in modo sequenziale, a partire dall'inizio. Se l'unità di memorizzazione che lo contiene è sequenziale per sua natura, come avviene per un nastro o un lettore di schede perforate, si può avere solo una fase di lettura o una fase di scrittura, senza la possibilità di mescolare le due operazioni, mentre se si dispone di un'unità di memorizzazione ad accesso di-

retto, come nel caso di un disco, si può leggere e poi sovrascrivere lo stesso record.

Nello schema sintattico, la metavariabile ***file-name*** deve essere sostituita con il nome che si vuole attribuire al file nell'ambito del programma (non si tratta del nome che questo ha eventualmente per il sistema operativo). La metavariabile ***hardware-name*** va sostituita con un nome che serve a identificare l'unità di memorizzazione che contiene il file; questo nome dipende dal compilatore ma generalmente si mette '**DISK**' per indicare un file su disco. Altri nomi per la metavariabile ***hardware-name*** potrebbero essere: '**TAPE**', '**PRINTER**', '**PUNCH**', '**READER**' (gli ultimi due sarebbero un perforatore e un lettore di schede).

Il linguaggio COBOL è fatto per poter essere adattato a sistemi operativi molto diversi. In un sistema Unix, l'accesso alle unità di memorizzazione avviene attraverso dei file di dispositivo, pertanto, a seconda del compilatore, potrebbe anche essere superfluo dichiarare il tipo di unità di memorizzazione in questo modo, anche se in passato il linguaggio obbligava a farlo. Proprio per questo motivo, ci sono compilatori che, al posto di indicare il tipo di unità fisica attraverso un nome prestabilito, richiedono di mettere subito il percorso del file a cui si vuole fare riferimento, nonostante il linguaggio preveda per questo una dichiarazione separata nella divisione '**DATA DIVISION**'. In questo senso, nello schema sintattico appare la possibilità di indicare una stringa alfanumerica con il percorso del file (***literal-file-name***).

Nella dichiarazione '**RESERVE** ***integer***', la metavariabile ***integer***

rappresenta un numero intero di record da usare come memoria tampone. Se non si usa questa dichiarazione che, come si vede dallo schema sintattico, è facoltativa, viene usata la dimensione predefinita.

La dichiarazione '**ORGANIZATION IS SEQUENTIAL**' è facoltativa e sottintesa; tuttavia va osservato il significato che assume quando si aggiunge la parola '**LINE**'. In generale, il linguaggio COBOL considera i file come composti da record di dimensione uniforme. Quando però si vuole lavorare con i file di testo, le righe di questi file sono suddivise in base alla presenza del codice di interruzione di riga (che può cambiare da un sistema operativo all'altro). Volendo considerare in COBOL le righe di un file di testo pari a dei record di dimensione variabile, occorre aggiungere l'opzione '**LINE**', così da chiarire che si tratta sì di un'organizzazione sequenziale, ma di un file suddiviso in «righe».

La dichiarazione '**ACCESS MODE IS SEQUENTIAL**' è facoltativa, perché l'accesso a un file organizzato in modo sequenziale può essere solo sequenziale.

La dichiarazione '**FILE STATUS IS** ***data-name***' consente di indicare una variabile (da specificare nella sezione '**WORKING-STORAGE SECTION**' della divisione '**DATA DIVISION**') da usare eventualmente per conoscere lo stato dell'ultima operazione svolta sul file. Questa variabile deve poter rappresentare un valore di due caratteri (il modello di definizione della variabile deve essere '**XX**') e quando contiene il valore zero indica che l'ultima operazione è stata eseguita con successo (si vedano le tabelle 72.48 e 72.49, che appaiono alla fine del capitolo).

Il punto fermo che conclude l'istruzione '**SELECT**' appare una volta sola, alla fine; tutta l'istruzione deve risiedere nell'area B.

Viene mostrato un esempio completo di un programma COBOL che legge un file sequenziale:

Listato 72.34. Programma elementare che legge un file sequenziale.

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   ESEMPIO-SEQUENZIALE.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 INSTALLATION. NANOLINUX IV,
000500               TINYCOBOL 0.61.
000600 DATE-WRITTEN. 2005-02-16.
000700 ENVIRONMENT DIVISION.
000800 INPUT-OUTPUT SECTION.
000900*
001000 FILE-CONTROL.
001100     SELECT FILE-NAME ASSIGN TO DISK
001200                      ORGANIZATION IS SEQUENTIAL
001300                      ACCESS MODE IS SEQUENTIAL
001400                      FILE STATUS IS DATA-NAME.
001500*
001600 DATA DIVISION.
001700 FILE SECTION.
001800 FD  FILE-NAME
001900     LABEL RECORD IS STANDARD
002000     VALUE OF FILE-ID IS "input.seq".
002100 01  RECORD-NAME  PIC X(20).
002200 WORKING-STORAGE SECTION.
002300 01  DATA-NAME    PIC XX.
002400 PROCEDURE DIVISION.
002500 MAIN.
002600     OPEN INPUT  FILE-NAME.
002700     DISPLAY "FILE STATUS: ", DATA-NAME.
```

```
002800     PERFORM READ-FILE UNTIL DATA-NAME NOT = ZERO.
002900     CLOSE FILE-NAME.
003000     STOP RUN.
003100 READ-FILE.
003200     READ FILE-NAME.
003300     DISPLAY "FILE STATUS: " DATA-NAME, " RECORD: ",
003400             RECORD-NAME.
```

Il file indicato come '**FILE-NAME**' è associato in pratica al file '`input.seq`'. Si può supporre che questo file abbia il contenuto seguente, senza alcun codice di interruzione di riga:

```
aaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbb↩
↪ccccccccccccccccccccddddddddddddddddddddd
```

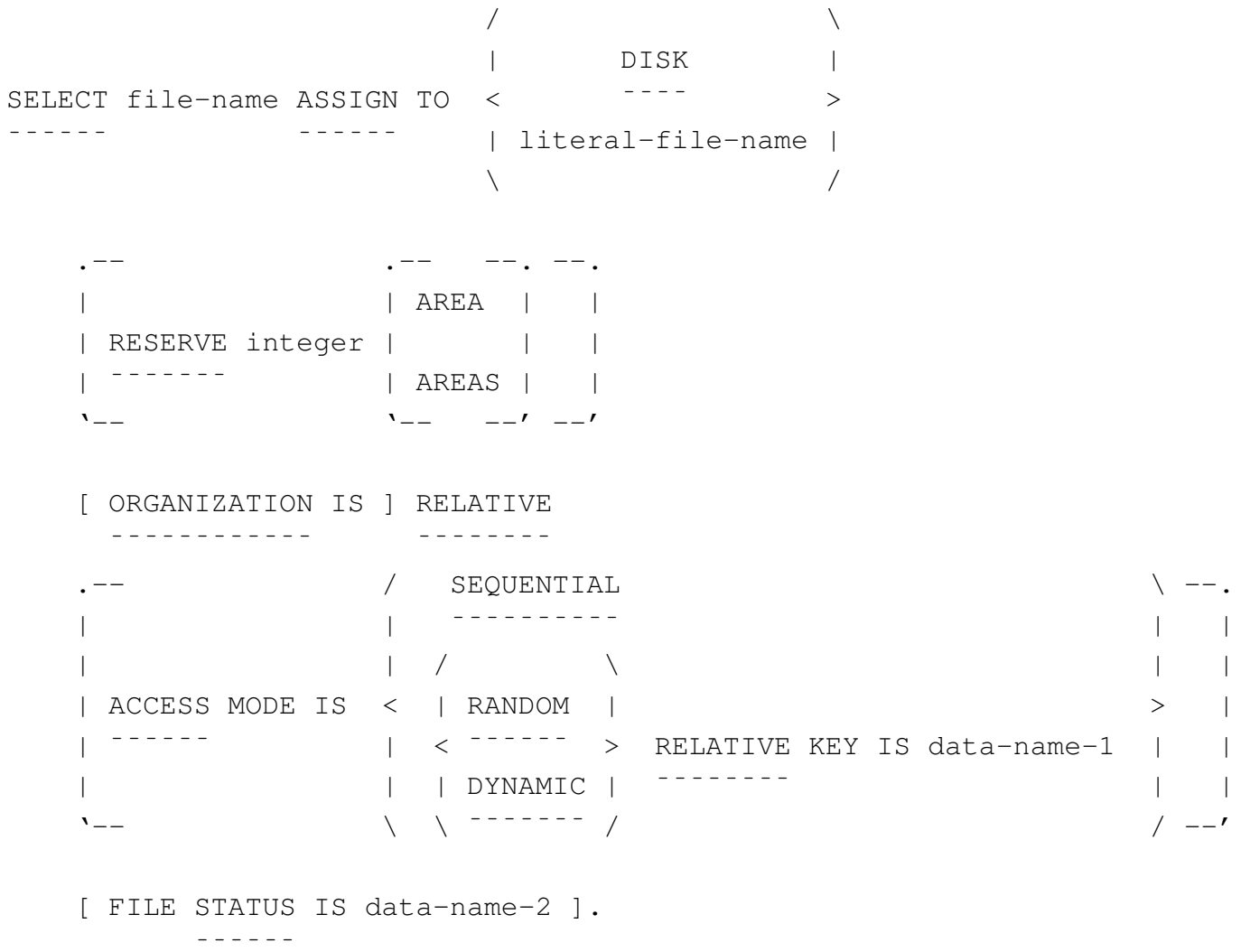
Eseguendo il programma dell'esempio si potrebbe ottenere il testo seguente attraverso lo schermo:

```
FILE STATUS: 00
FILE STATUS: 00 RECORD: aaaaaaaaaaaaaaaaaaaa
FILE STATUS: 00 RECORD: bbbbbbbbbbbbbbbbbbbb
FILE STATUS: 00 RECORD: cccccccccccccccccccc
FILE STATUS: 00 RECORD: ddddddddddddddddddddd
FILE STATUS: 10 RECORD: ddddddddddddddddddddd
```

72.4.3.4 Istruzione «SELECT» per i file relativi

«

Lo schema sintattico semplificato per l'istruzione '**SELECT**', da usare nel paragrafo '**FILE-CONTROL**', per dichiarare un file organizzato in modo «relativo» è quello che si può vedere nella figura successiva:



Il file organizzato in modo relativo può essere utilizzato secondo un accesso sequenziale, oppure facendo riferimento ai record per numero, considerando che il primo ha proprio il numero uno. Quando si individuano i record per numero, si distinguono due modalità di accesso: diretto (‘**RANDOM**’) e dinamico (‘**DYNAMIC**’). L’accesso diretto richiede che per ogni operazione l’indicazione del numero del record a cui si vuole fare riferimento, mentre con l’accesso dinamico è anche possibile eseguire delle operazioni di lettura sequenziali (‘**READ NEXT**’).

L’organizzazione relativa, oltre alle operazioni di lettura e scrittura, prevede la cancellazione dei record, che comunque possono essere

rimpiazzati successivamente attraverso un'operazione di scrittura. Si osservi comunque che un record che risulta essere stato cancellato, non può essere letto.

Osservando lo schema sintattico si può intuire che la prima parte dell'istruzione '**SELECT**' funzioni nello stesso modo di un file organizzato sequenzialmente; la differenza più importante riguarda la definizione del tipo di unità di memorizzazione che, date le caratteristiche dei file organizzati in modo relativo, deve consentire un accesso diretto ai dati.

La dichiarazione '**RESERVE** ***integer***' si usa nello stesso modo del file organizzato sequenzialmente.

L'indicazione dell'organizzazione, attraverso la dichiarazione '**ORGANIZATION IS RELATIVE**' è obbligatoria, anche se probabilmente è sufficiente scrivere soltanto '**RELATIVE**'.

Se non viene specificata la dichiarazione '**ACCESS MODE**', si intende che l'accesso debba avvenire in modo sequenziale, altrimenti vale quanto indicato espressamente. Se l'accesso richiesto è diretto o dinamico, è necessario indicare quale variabile usare per specificare il numero del record, nella posizione occupata nello schema sintattico dalla metavariabile ***data-name-1*** (da specificare ulteriormente nella sezione '**WORKING-STORAGE SECTION**' della divisione '**DATA DIVISION**').

La dichiarazione '**FILE STATUS IS** ***data-name-2***' funziona nello stesso modo descritto a proposito dei file organizzati in modo sequenziale.

Il punto fermo che conclude l'istruzione '**SELECT**' appare una volta sola, alla fine; tutta l'istruzione deve risiedere nell'area B.

Viene mostrato un esempio completo di un programma COBOL che legge un file relativo, ad accesso diretto, scandendo sequenzialmente il numero del record:

Listato 72.38. Programma elementare che legge un file relativo, ad accesso diretto.

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   ESEMPIO-RELATIVO-DIRETTO.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-03-08.
000500 ENVIRONMENT DIVISION.
000600 INPUT-OUTPUT SECTION.
000700*
000800 FILE-CONTROL.
000900     SELECT MIO-FILE ASSIGN TO "input.rel"
001000                     ORGANIZATION IS RELATIVE
001100                     ACCESS MODE IS RANDOM
001200                     RELATIVE KEY IS N-RECORD
001300                     FILE STATUS IS STATO-DEL-FILE.
001400*
001500 DATA DIVISION.
001600 FILE SECTION.
001700 FD  MIO-FILE
001800     LABEL RECORD IS STANDARD.
001900 01  MIO-RECORD  PIC X(20).
002000 WORKING-STORAGE SECTION.
002100 77  N-RECORD             PIC 9999 COMP VALUE IS ZERO.
002200 77  STATO-DEL-FILE       PIC XX.
002300 PROCEDURE DIVISION.
002400 MAIN.
002500     OPEN INPUT MIO-FILE.
002600     DISPLAY "FILE STATUS: ", STATO-DEL-FILE.
002700     PERFORM READ-FILE UNTIL STATO-DEL-FILE NOT = ZERO.
002800     CLOSE MIO-FILE.
002900     STOP RUN.
```

```
003000 READ-FILE.
003100     ADD 1 TO N-RECORD.
003200     READ MIO-FILE
003300          INVALID KEY DISPLAY "INVALID KEY!".
003400     DISPLAY "FILE STATUS: " STATO-DEL-FILE,
003500             " RECORD: ", N-RECORD, " ", MIO-RECORD.
003600
```

Il file indicato come '**MIO-FILE**' è associato in pratica al file '`input.rel`'. Si può supporre che questo file sia composto dall'elenco seguente di record logici:

1. '**aaaaaaaaaaaaaaaaaaaa**'
2. '**bbbbbbbbbbbbbbbbbbbb**'
3. '**ccccccccccccccccccccc**'
4. '**ddddddddddddddddddddd**'

Eseguendo il programma dell'esempio si potrebbe ottenere il testo seguente attraverso lo schermo:

```
FILE STATUS: 00
FILE STATUS: 00 RECORD: 0001 aaaaaaaaaaaaaaaaaaaa
FILE STATUS: 00 RECORD: 0002 bbbbbbbbbbbbbbbbbbbb
FILE STATUS: 00 RECORD: 0003 ccccccccccccccccccccc
FILE STATUS: 00 RECORD: 0004 ddddddddddddddddddddd
INVALID KEY!
FILE STATUS: 23 RECORD: 0005 ddddddddddddddddddddd
```

Segue un altro esempio completo per la lettura di un file relativo, utilizzando un accesso dinamico, partendo dal primo record e selezionando i successivi attraverso la richiesta del prossimo:

Listato 72.40. Programma elementare che legge un file relativo, ad accesso dinamico.

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   ESEMPIO-RELATIVO-DINAMICO.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-03-08.
000500 ENVIRONMENT DIVISION.
000600 INPUT-OUTPUT SECTION.
000700*
000800 FILE-CONTROL.
000900     SELECT MIO-FILE ASSIGN TO "input.rel"
001000                     ORGANIZATION IS RELATIVE
001100                     ACCESS MODE IS DYNAMIC
001200                     RELATIVE KEY IS N-RECORD
001300                     FILE STATUS IS STATO-DEL-FILE.
001400*
001500 DATA DIVISION.
001600 FILE SECTION.
001700 FD  MIO-FILE
001800     LABEL RECORD IS STANDARD.
001900 01  MIO-RECORD  PIC X(20).
002000 WORKING-STORAGE SECTION.
002100 77  N-RECORD             PIC 9999 COMP VALUE IS 1.
002200 77  STATO-DEL-FILE       PIC XX.
002300 PROCEDURE DIVISION.
002400 MAIN.
002500     OPEN INPUT MIO-FILE.
002600     DISPLAY "FILE STATUS: ", STATO-DEL-FILE.
002700     READ MIO-FILE
002800          INVALID KEY DISPLAY "INVALID KEY!".
002900     PERFORM READ-FILE UNTIL STATO-DEL-FILE NOT = ZERO.
003000     CLOSE MIO-FILE.
003100     STOP RUN.
003200 READ-FILE.
003300     DISPLAY "FILE STATUS: " STATO-DEL-FILE,
```

```
003400              " RECORD: ", N-RECORD, " ", MIO-RECORD.
003500     READ MIO-FILE NEXT RECORD
003600          AT END DISPLAY "END OF FILE!".
003700
```

Il file che viene letto è lo stesso dell'esempio precedente e il risultato si dovrebbe ottenere, si può vedere così:

```
FILE STATUS: 00
FILE STATUS: 00 RECORD: 0001 aaaaaaaaaaaaaaaaaaaa
FILE STATUS: 00 RECORD: 0002 bbbbbbbbbbbbbbbbbbbb
FILE STATUS: 00 RECORD: 0003 cccccccccccccccccccc
FILE STATUS: 00 RECORD: 0004 dddddddddddddddddddd
END OF FILE!
```

72.4.3.5 Istruzione «SELECT» per i file a indice

«

Lo schema sintattico semplificato per l'istruzione '**SELECT**', da usare nel paragrafo '**FILE-CONTROL**', per dichiarare un file organizzato a indici è quello che si può vedere nella figura successiva:

```
                            /                    \
                            |       DISK         |
SELECT file-name ASSIGN TO  <        ----        >
------           ------     | literal-file-name  |
                            \                    /

    .--                .--   --. --.
    |                  | AREA  |   |
    | RESERVE integer  |       |   |
    | -------          | AREAS |   |
    '--                '--   --' --'

    [ ORGANIZATION IS ] INDEXED
      ------------      -------

    .--                /              \ --.
    |                  |  SEQUENTIAL  |   |
    |                  |              |   |
    | ACCESS MODE IS   <  RANDOM      >   |
    | ------           |  ------      |   |
    |                  |  DYNAMIC     |   |
    '--                \  -------     / --'

    RECORD KEY IS data-name-1 [WITH DUPLICATES]
    ------                    ----------

    [ ALTERNATE RECORD KEY IS data-name-2 [WITH DUPLICATES] ]...
      ---------                           ----------

    [ FILE STATUS IS data-name-3 ].
           ------
```

Un file organizzato a indice è un file che consente un accesso diretto ai record in base a una chiave costituita da una porzione del record stesso. A titolo di esempio, si può immaginare un file contenente i dati anagrafici dei dipendenti di un'azienda, che in una posizione precisa dei record riporta il numero di matricola di ognuno; in tal modo, il numero di matricola può essere usato per definire la chiave

di accesso ai record.

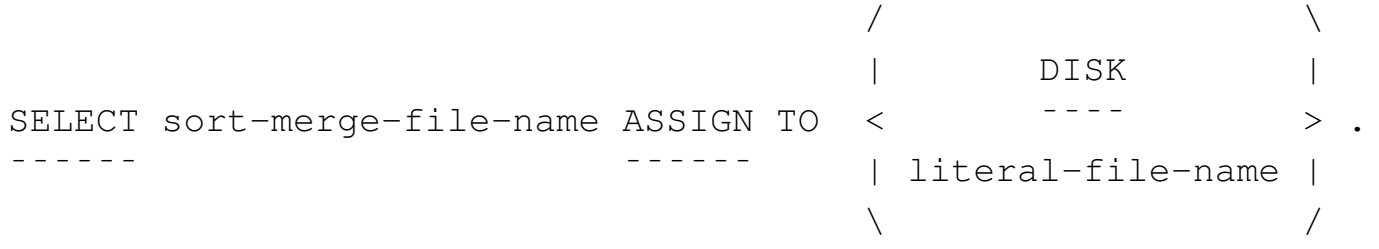
Il file organizzato a indice può disporre di una o più chiavi di accesso e può essere consentita o meno la presenza di record con chiavi uguali.

Rispetto ai file organizzati sequenzialmente o in modo relativo, lo schema sintattico per i file organizzati a indice ha le dichiarazioni '**RECORD KEY**' e '**ALTERNATE RECORD KEY**' per poter specificare la chiave o le chiavi di accesso. Le metavariabili '**data-name-1**' e '**data-name-2**' devono essere nomi di porzioni di record, come dichiarato nella divisione '**DATA DIVISION**', in corrispondenza della descrizione del record stesso. Naturalmente, l'opzione '**WITH DUPLICATES**' serve a dichiarare l'intenzione di gestire chiavi uguali su più record.

72.4.3.6 Riordino e fusione

«

Oltre ai file comuni, per i quali si stabilisce un'organizzazione e un tipo di accesso, sono previsti dei file da usare soltanto per ottenere un riordino o una fusione (*sort*, *merge*). Per questi file occorre una dichiarazione apposita con l'istruzione '**SELECT**', secondo lo schema sintattico seguente:



Viene proposto un esempio di riordino di file, nel quale, in particolare, si dichiarano i nomi dei file su disco, direttamente nell'istruzione '**SELECT**':

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   ORDINA.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-02-25.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 INPUT-OUTPUT SECTION.
000900*
001000 FILE-CONTROL.
001100*
001200     SELECT FILE-DA-ORDINARE     ASSIGN TO "input.seq".
001300     SELECT FILE-ORDINATO        ASSIGN TO "output.seq".
001400     SELECT FILE-PER-IL-RIORDINO ASSIGN TO "sort.tmp".
001500*
001600 DATA DIVISION.
001700*
001800 FILE SECTION.
001900*
002000 FD  FILE-DA-ORDINARE.
002100 01  RECORD-DA-ORDINARE           PIC X(80).
002200*
002300 FD  FILE-ORDINATO.
002400 01  RECORD-ORDINATO              PIC X(80).
002500*
002600 SD  FILE-PER-IL-RIORDINO.
002700*
002800 01  RECORD-PER-IL-RIORDINO.
002900     02  CHIAVE-ORDINAMENTO       PIC X(10).
003000     02  FILLER                   PIC X(70).
003100*
003200 PROCEDURE DIVISION.
003300*
003400 MAIN.
003500     SORT FILE-PER-IL-RIORDINO,
```

```
003600           ON ASCENDING KEY CHIAVE-ORDINAMENTO,
003700           USING FILE-DA-ORDINARE,
003800           GIVING FILE-ORDINATO.
003900*
004000     STOP RUN.
004100*
```

Come si può vedere, si vuole ordinare il file ‘`input.seq`’ per generare il file ‘`output.seq`’, ordinato. Per fare questo, si usa un file intermedio, denominato ‘`sort.tmp`’. Al termine dell’operazione, non dovrebbe rimanere traccia del file intermedio.

Si osservi che non si rende necessaria l’apertura dei file coinvolti per portare a termine l’operazione.

L’esempio seguente riguarda la fusione: si hanno i file ‘`input-1.seq`’ e ‘`input-2.seq`’ ordinati e si vuole ottenere il file ‘`output.seq`’ con la somma dei record, mantenendo l’ordinamento:

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   MERGE.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-02-25.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 INPUT-OUTPUT SECTION.
000900*
001000 FILE-CONTROL.
001100*
001200     SELECT FILE-INPUT-1         ASSIGN TO "input-1.seq".
001300     SELECT FILE-INPUT-2         ASSIGN TO "input-2.seq".
001400     SELECT FILE-OUTPUT          ASSIGN TO "output.seq".
```

```
001500     SELECT FILE-PER-LA-FUSIONE  ASSIGN TO "merge.tmp".
001600*
001700 DATA DIVISION.
001800*
001900 FILE SECTION.
002000*
002100 FD  FILE-INPUT-1
002200 01  RECORD-1                        PIC X(80).
002300*
002400 FD  FILE-INPUT-2
002500 01  RECORD-2                        PIC X(80).
002600*
002700 FD  FILE-OUTPUT
002800 01  RECORD-OUTPUT                   PIC X(80).
002900*
003000 SD  FILE-PER-LA-FUSIONE.
003100*
003200 01  RECORD-PER-LA-FUSIONE.
003300     02  CHIAVE-ORDINAMENTO          PIC X(10).
003400     02  FILLER                      PIC X(70).
003500*
003600 PROCEDURE DIVISION.
003700*
003800 MAIN.
003900     MERGE FILE-PER-LA-FUSIONE
004000           ON ASCENDING KEY CHIAVE-ORDINAMENTO,
004100           USING FILE-INPUT-1,
004200                 FILE-INPUT-2,
004300           GIVING FILE-OUTPUT.
004400*
004500     STOP RUN.
004600*
```

Si osservi che esistono compilatori COBOL, di buona qualità, che però non offrono le funzionalità di riordino e di fusione, oppure

non in modo completo. È frequente l'assenza della funzione per la fusione dei file ordinati.

72.4.3.7 Paragrafo «I-O-CONTROL»

Il paragrafo '**I-O-CONTROL**' è opzionale e il suo scopo è quello di specificare l'utilizzo comune delle aree di memoria centrale adibite alla gestione dei file.

```
I-O-CONTROL.
------------

    .--      .--          --.                                      --.
    |        |  RECORD     |                                        |
    |        |  ------     |                                        |
    |  SAME  |  SORT       |  AREA FOR file-name-1 [file-name-2]...  |... .
    |  ----  |  ----       |                                        |
    |        |  SORT-MERGE |                                        |
    `--      `------------'                                      --'
```

L'utilità dell'utilizzo del paragrafo '**I-O-CONTROL**' dipende molto dal compilatore, che potrebbe anche limitarsi a ignorare l'istruzione '**SAME**...**AREA**', in tutto o solo in parte. Tuttavia, quando l'istruzione '**SAME**...**AREA**' viene presa in considerazione, ci sono delle conseguenze nell'accesso ai file, che bisogna conoscere.

Per cominciare: si intuisce dallo schema sintattico che l'istruzione '**SAME**...**AREA**' inizia nell'area B del modulo di programmazione, si vede che il punto fermo è richiesto solo alla fine del gruppo di istruzioni '**SAME**...**AREA**', inoltre sono evidenti quattro possibilità:

```
SAME AREA FOR file-name-1 [file-name-2]... .
----

SAME RECORD AREA FOR file-name-1 [file-name-2]... .
---- ------

SAME SORT AREA FOR file-name-1 [file-name-2]... .
---- ----

SAME SORT-MERGE AREA FOR file-name-1 [file-name-2]... .
---- ----------
```

Utilizzando la prima forma dell'istruzione '**SAME AREA**', si intende richiedere al compilatore che la gestione dei file elencati sia fatta condividendo tutto quello che si può condividere nella memoria centrale. Così facendo, nell'ambito del gruppo specificato, solo un file può essere aperto simultaneamente; inoltre, se si utilizzano più istruzioni '**SAME AREA**', un file può appartenere soltanto a uno di questi raggruppamenti.

Utilizzando l'istruzione '**SAME RECORD AREA**' si richiede al compilatore di gestire lo spazio della memoria tampone (dei record) di un gruppo di file in modo comune. Così facendo, la lettura di un record di un file del gruppo, comporta il fatto che gli stessi dati siano disponibili come se fossero stati letti da tutti gli altri file del gruppo. I file di un gruppo definito con questa istruzione possono essere aperti simultaneamente, ma le operazioni di accesso ai dati non possono essere simultanee; inoltre, un file può appartenere a un solo raggruppamento di questo tipo.

Teoricamente, i file indicati in un raggruppamento con l'istruzione '**SAME AREA**' possono apparire anche in un raggruppamento con l'istruzione '**SAME RECORD AREA**', ma in tal caso deve trattarsi di tutti quelli che appartengono al primo di questi due (tutti quelli in '**SAME AREA**' devono essere parte di quello in '**SAME RECORD**

AREA’). Inoltre, questo fatto comporta che i file che si trovano anche in ‘**SAME AREA**’ non possono essere aperti simultaneamente.

Nei manuali COBOL classici si sottolinea il fatto che la condivisione dei record offra dei vantaggi in velocità e in risparmio di memoria; in particolare si suggerisce in tali manuali la possibilità di dichiarare nel dettaglio uno solo dei record del gruppo, oppure la possibilità di ridefinire i record cambiando il punto di vista (il record rispetto a quello di un altro). Tuttavia, considerata la potenza elaborativa degli elaboratori attuali, dal momento che esiste comunque la possibilità di ridefinire la suddivisione di un record, l’uso di questo paragrafo diventa sconsigliabile, se non altro per le complicazioni che si creano nell’interpretazione umana del programma sorgente.

Le istruzioni ‘**SAME SORT AREA**’ e ‘**SAME SORT-MERGE AREA**’ sono equivalenti e consentono di condividere la memoria utilizzata per i file che servono specificatamente per il riordino o la fusione. Premesso che in questi raggruppamenti non possono apparire file che appartengono a un gruppo definito come ‘**SAME AREA**’, è invece possibile inserire anche nomi di file che non sono stati dichiarati per l’ordinamento o la fusione, ma la loro presenza fa sì che questi file non possano essere aperti quando invece lo sono quelli che si utilizzano proprio per tale scopo.

I file dichiarati con l'indicatore '**SD**' nella sezione '**FILE SECTION**' servono per portare a termine le operazioni di riordino e di fusione, ma si avvalgono di file in ingresso e di file in uscita, che vengono dichiarati normalmente con l'indicatore '**FD**'. Tutti i file coinvolti in un procedimento di riordino e di fusione, non devono essere aperti esplicitamente durante questa fase.

Tabella 72.48. Codici di due caratteri sullo stato dei file ('**FILE STATUS**'), secondo lo standard del 1985: il significato del primo dei due caratteri.

Codice	Descrizione
0*x*	L'ultimo accesso al file si è concluso sostanzialmente con successo.
1*x*	Si è verificato un tentativo di leggere oltre la fine del file.
2*x*	Si è verificato un errore riferito alla chiave di accesso di un file organizzato a indici.
3*x*	Si è verificato un errore che impedisce di accedere ulteriormente al file.
4*x*	Si è verificato un errore «logico», dovuto a una sequenza errata nelle operazioni o al tentativo di eccedere rispetto ai limiti stabiliti.
9*x*	Si tratta di errori diversi, stabiliti senza uno standard precisato da chi ha realizzato il compilatore.

Tabella 72.49. Codici di due caratteri sullo stato dei file ('**FILE STATUS**'), secondo lo standard del 1985: significato dettagliato.

Codice	Organizzazione sequenziale	Organizzazione relativa	Organizzazione a indici
00	Operazione eseguita con successo.	idem	idem
02	--	--	L'operazione ha avuto successo, ma è stata scoperta una chiave doppia: la lettura di un record evidenzia che è disponibile un altro record con la stessa chiave; la scrittura di un record risulta avere una chiave già presente in altri.
04	La lunghezza del record letto non corrisponde a quella che dovrebbe avere.	idem	idem
05	Il tentativo di aprire un file opzionale mancante è risultato nella sua creazione e apertura successiva.	idem	idem

Codice	Organizzazione sequenziale	Organizzazione relativa	Organizzazione a indici
07	Il file non si trova su nastro e le opzioni specifiche per tale tipo di unità, contenute nei comandi di apertura o di chiusura del file, sono state ignorate.	--	--
10	Si è verificato un tentativo di leggere oltre la fine del file, oppure di leggere un file opzionale che non risulta presente.	--	--

Codice	Organizzazione sequenziale	Organizzazione relativa	Organizzazione a indici
14	--	La dimensione in record del file è più grande della capacità della variabile usata come indice. Questo tipo di errore si può manifestare quando si tenta una lettura sequenziale che dovrebbe incrementare automaticamente il valore dell'indice, ma si trova a non poterlo fare.	--
21	--	--	Si è verificato un errore di sequenza nelle chiavi durante un accesso sequenziale al file.
22	--	Si è verificato un tentativo di scrivere un record già esistente (senza prima averlo cancellato).	Si è verificato un tentativo di scrivere un record con chiave doppia, quando ciò non è consentito.
23	--	Il record richiesto non esiste.	idem

Codice	Organizzazione sequenziale	Organizzazione relativa	Organizzazione a indici
24	--	Tentativo di scrittura oltre il limite della dimensione consentita, oppure tentativo di scrittura sequenziale di un record che ha un numero più grande della capacità della variabile usata come chiave.	Tentativo di scrittura oltre il limite della dimensione consentita.
30	Errore permanente senza altre indicazioni.	idem	idem
34	Si è verificato un errore dovuto a un tentativo di scrittura oltre il limite fisico del file.	--	--
35	Non è stato possibile aprire un file indispensabile.	idem	idem
36	L'operazione richiesta non è gestita dall'unità che contiene il file.	idem	idem

Codice	Organizzazione sequenziale	Organizzazione relativa	Organizzazione a indici
38	È stata tentata l'apertura di un file che risulta essere stato chiuso con l'opzione '**LOCK**'.	idem	idem
39	È stata tentata l'apertura di un file, le cui caratteristiche reali sono incompatibili con quelle dichiarate nel programma.	idem	idem
41	È stato aperto un file che risulta essere già aperto.	idem	idem
42	È stato chiuso un file che non risultava essere aperto.	idem	idem
43	Non è stato eseguito un comando '**READ**' prima del comando '**REWRITE**'.	Durante un accesso sequenziale, non è stato eseguito un comando '**READ**' prima del comando '**REWRITE**' o del comando '**DELETE**'.	Durante un accesso sequenziale, non è stato eseguito un comando '**READ**' prima del comando '**REWRITE**' o del comando '**DELETE**'.

Codice	Organizzazione sequenziale	Organizzazione relativa	Organizzazione a indici
44	Si è verificato un problema legato alla dimensione del record.	idem	idem
46	Durante un accesso sequenziale in lettura, si è verificato un errore, successivo a un altro tentativo fallito di lettura.	idem	idem
47	Tentativo di lettura di un file che non risulta essere aperto per questo tipo di accesso.	idem	idem
48	Tentativo di scrittura di un file che non risulta essere aperto per questo tipo di accesso.	idem	idem
49	Tentativo di riscrittura di un file che non risulta essere aperto per questo tipo di accesso.	idem	idem

72.5 Divisione «DATA DIVISION»

«

La divisione '**DATA DIVISION**' costituisce la terza parte, la più complessa, di un programma COBOL e ha lo scopo di descrivere tutti i dati (variabili e costanti) utilizzati nel programma. Si distinguono in particolare: i record dei file a cui si vuole accedere, altre variabili e valori costanti creati o utilizzati dal programma.

La divisione si articola normalmente in tre sezioni: '**FILE SECTION**', per tutte le informazioni riguardanti i file dichiarati nella divisione '**ENVIRONMENT DIVISION**', soprattutto per quanto riguarda la struttura del record; '**WORKING-STORAGE SECTION**' per tutte le variabili (che possono essere sia scalari, sia strutturate, ma in questo secondo caso vengono chiamate ugualmente record, anche se non sono associate direttamente ad alcun file) e le costanti necessarie per l'elaborazione; '**LINKAGE SECTION**', per la dichiarazione dei dati condivisi con altri programmi.

In questo manuale la descrizione della sezione '**LINKAGE SECTION**' viene omessa del tutto; pertanto, lo schema sintattico seguente non la riporta:

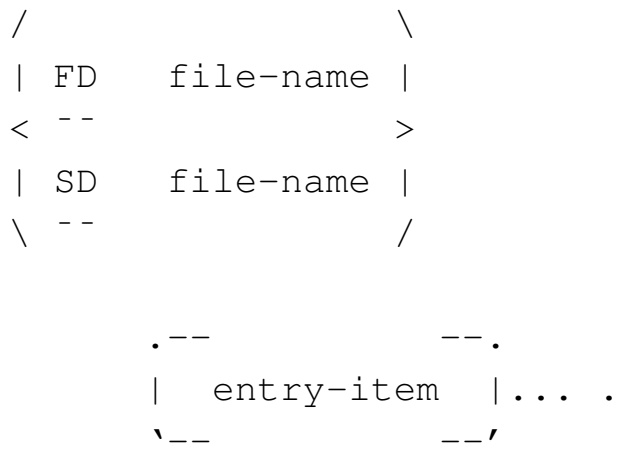
```
   DATA-DIVISION.
   -------------

.--
|
|  FILE SECTION.
|  ------------
`--
       .--                                                          --.  --.
       |  file-description-entry        record-description-entry...  |    |
       |                                                             |... |
       |  sort-merge-description-entry  record-description-entry...  |    |
       `--                                                          --'  --'
.--
|
|  WORKING-STORAGE SECTION.
|  ----------------------
`--
       .--                            --.   --.
       |  77-level-description-entry  |     |
       |                              |...  |
       |  record-description-entry    |     |
       `--                            --'   --'
```

Sulla base della terminologia usata nello schema sintattico, si può intuire il fatto che per il linguaggio COBOL, il termine record ha un significato particolare: si tratta di una variabile strutturata, che pertanto può essere scomposta in campi, in modo più o meno articolato. In questo senso, il contenuto della sezione '**WORKING-STORAGE SECTION**' viene suddiviso in due tipi di dichiarazioni: variabili scalari non suddivisibili (la metavariabile ***77-level-description-entry***) e variabili strutturate, ovvero record. Naturalmente, una variabile strutturata (dichiarata come record) può essere gestita e usata tranquillamente come se fosse uno scalare puro e semplice, ma questo fatto ha eventualmente delle ripercussioni nell'efficienza del programma che si ottiene dalla compilazione.

72.5.1 Sezione «FILE SECTION»

La sezione '**FILE SECTION**' ha lo scopo di definire le caratteristiche fisiche dei file e la struttura dei record. Tradizionalmente sarebbe in questa sezione che si specifica il nome o il percorso dei file in base al sistema operativo in cui si deve utilizzare il programma, salvo il caso in cui il compilatore voglia questa indicazione direttamente nella divisione '**ENVIRONMENT DIVISION**', precisamente nell'istruzione '**SELECT**' della sezione '**FILE CONTROL**'.

La descrizione di un file nella sezione '**FILE SECTION**' inizia con l'***indicatore di livello*** '**FD**' o '**SD**', a seconda che si tratti di un file «normale» o di un file da usare per le operazioni di riordino e fusione. Si osservi che queste due istruzioni iniziano nell'area A del modulo di programmazione, continuando poi nell'area B, ma è importante sottolineare che già il nome del file, evidenziato nello schema sintattico con il nome ***file-name***, deve iniziare nell'area B:



Dopo ogni indicatore di livello '**FD**' o '**SD**' deve apparire la dichiarazione della variabile strutturata che rappresenta il record del file; tale dichiarazione inizia con il livello 01.

72.5.1.1 Indicatore di livello «FD»

I file comuni, ovvero quelli che non sono stati dichiarati esplicitamente per eseguire delle operazioni di riordino o di fusione, si descrivono nella sezione '**FILE SECTION**' con l'indicatore di livello '**FD**' (*File description*), che in pratica è un'istruzione singola. Si ricordi che il nome del file che segue la parola chiave '**FD**' deve iniziare nell'area B del modulo di programmazione:

```
FD  file-name
--

     .--                                                 /              \ --.
     |                                                   |  RECORDS     |   |
     |  BLOCK CONTAINS [integer-1 TO] integer-2    <     |  -------     |   |
     |  -----                          --                |  CHARACTERS  |   |
     `--                                                 \              / --'

      [ RECORD CONTAINS [integer-3 TO] integer-4 CHARACTERS ]
        ------                         --

     .--        /                \   /              \ --.
     |          |  RECORD IS     |   |  OMITTED     |   |
     |  LABEL   <  ------        >   <  -------     >   |
     |  -----   |  RECORDS ARE   |   |  STANDARD    |   |
     `--        \  -------       /   \  --------    / --'

     .--           /                      /                \  \    --.
     |             |                      |  data-name-1   |  |      |
     |  VALUE OF   <  label-info-1  IS     <                >  >...   |
     |  -----      |                      |   literal-1    |  |      |
     `--           \                      \                /  /    --'

     .--       /                \                                  --.
     |         |  RECORD IS     |                                    |
     |  DATA   <  ------        >  data-name-2 [data-name-3]...      |
     |  ----   |  RECORDS ARE   |                                    |
     `--       \  -------       /                                  --'

      [ CODE-SET IS alphabet-name ].
        --------
```

Si osservi che, a seconda del compilatore e del sistema operativo per il quale il programma viene compilato, diverse dichiarazioni inserite nell'indicatore di livello '**FD**' potrebbero essere ignorate in pratica.

72.5.1.2 Indicatore di livello «SD»

I file da usare specificatamente per il riordino o la fusione, si descrivono nella sezione '**FILE SECTION**' con l'indicatore di livello '**SD**' (*Sort description*), che in pratica è un'istruzione singola. Si ricordi che il nome del file che segue la parola chiave '**SD**' deve iniziare nell'area B:

```
SD  file-name
--

     [ RECORD CONTAINS [integer-3 TO] integer-4 CHARACTERS ]
       ------                         --

    .--          /                    /                \  \   --.
    |            |                    |  data-name-1   |  |     |
    |  VALUE OF  <  label-info-1  IS  <                >  >...  |
    |  -----     |                    |   literal-1    |  |     |
    `--          \                    \                /  /   --'

    .--      /               \                                 --.
    |        |  RECORD IS    |                                   |
    |  DATA  <  ------        >  data-name-2 [data-name-3]...   | .
    |  ----  |  RECORDS ARE  |                                   |
    `--      \  -------      /                                 --'
```

72.5.1.3 Dichiarazione «BLOCK CONTAINS»

All'interno dell'indicatore di livello '**FD**' è possibile dichiarare la dimensione di un blocco fisico per l'accesso ai record del file a cui si sta facendo riferimento.

In generale, si può contare sul fatto che il sistema operativo sia in grado di gestire in modo trasparente il problema dei blocchi fisici dei dati, rispetto ai record «logici» utilizzati dai programmi; tuttavia, ci possono essere contesti in cui il programma che si genera deve

provvedere da solo ad accedere all'unità di memorizzazione, pertanto in questi casi conviene dichiarare nel programma la dimensione del blocco di dati da usare per la comunicazione con l'unità stessa. Storicamente la definizione del blocco consente di gestire meglio l'utilizzo di un'unità a nastro; in altre situazioni, come per esempio con un lettore o perforatore di schede, il blocco può contenere un solo record.

```
                                                  /              \
                                                  |  RECORDS     |
BLOCK CONTAINS [integer-1 TO] integer-2           <  -------     >
-----                         --                  |  CHARACTERS  |
                                                  \              /
```

Omettendo questa dichiarazione, si intende lasciare al compilatore o al sistema operativo il compito di determinare un valore predefinito valido.

L'unità di misura del blocco dipende dalla parola usata, o non usata, alla fine della dichiarazione: la parola chiave '**RECORDS**' indica che i valori numerici si riferiscono a quantità di record, mentre diversamente si intendono dei «caratteri». Generalmente è da considerare che per caratteri si intendano byte.

Se viene indicato un valore solo (***integer-2***), si intende che il blocco possa avere soltanto quella dimensione, altrimenti, si intende dire al compilatore che c'è la possibilità di usare blocchi che hanno una dimensione minima (***integer-1***) e una massima (***integer-2***).

72.5.1.4 Dichiarazione «DATA RECORD»

La dichiarazione '**DATA RECORD**', che riguarda sia l'indicatore di livello '**FD**', sia '**SD**', è superata e generalmente viene ignorata dai compilatori. Il suo scopo è quello di dichiarare il nome di una o più variabili strutturate che descrivono il record del file. Questa dichiarazione è superata soprattutto perché il record viene comunque indicato successivamente attraverso la dichiarazione di una variabile strutturata apposita.

```
        /                \
        |  RECORD IS     |
DATA    <  ------        >  data-name-2 [data-name-3]...
----    |  RECORDS ARE   |
        \  -------       /
```

I nomi da inserire al posto delle metavariabili ***data-name*** dello schema sintattico devono corrispondere a nomi di record (variabili strutturate) descritti con il numero di livello 01. La presenza di più di uno di questi nomi nella dichiarazione '**DATA**' implica che i record del file possono avere una struttura e una dimensione differente.

72.5.1.5 Dichiarazione «LABEL RECORD»

A seconda del tipo di unità di memorizzazione dei dati, ci può essere la necessità di aggiungere ai record delle informazioni per poterne poi gestire l'accesso. Il linguaggio COBOL prevede la possibilità di dover gestire direttamente questo meccanismo di etichettatura dei record, ma generalmente i sistemi operativi attuali dovrebbero rendere questo meccanismo trasparente, togliendo al programma COBOL l'onere di doversene occupare.

La dichiarazione '**LABEL RECORD**' servirebbe per stabilire se siano da gestire le «etichette» dei record, oppure se questa funzione non debba essere considerata dal programma. Attualmente, tale dichiarazione è superata e generalmente i compilatori si limitano a ignorarla:

```
        /               \   /             \
        |  RECORD IS    |   |  OMITTED    |
LABEL   <  ------       >   <  -------    >
-----   |  RECORDS ARE  |   |  STANDARD   |
        \  -------      /   \  --------   /
```

Dovendo o volendo inserire questa dichiarazione, in caso di dubbio la forma '**LABEL RECORD IS STANDARD**' dovrebbe essere quella più adatta, anche se non è più compito del programma occuparsi delle etichette. Di per sé, l'omissione di questa dichiarazione comporta, per il compilatore che dovesse volerla, proprio l'utilizzo della forma standard.

72.5.1.6 Dichiarazione «RECORD CONTAINS»

«

La dichiarazione '**RECORD CONTAINS**', che riguarda sia l'indicatore di livello '**FD**', sia '**SD**', permette di specificare la dimensione del record:

```
RECORD CONTAINS [integer-3 TO] integer-4 CHARACTERS
------                         --
```

Come si può intuire, se si indica un valore solo, si intende che il record abbia una dimensione fissa, altrimenti si prevede un intervallo di valori: da un minimo a un massimo.

Generalmente, i compilatori si limitano a ignorare questa dichiarazione, perché le informazioni che porta sono già incluse nella

variabile strutturata che descrive il record stesso, pertanto è sufficiente associare più variabili strutturate nella dichiarazione '**DATA RECORD**'.

72.5.1.7 Dichiarazione «CODE-SET»

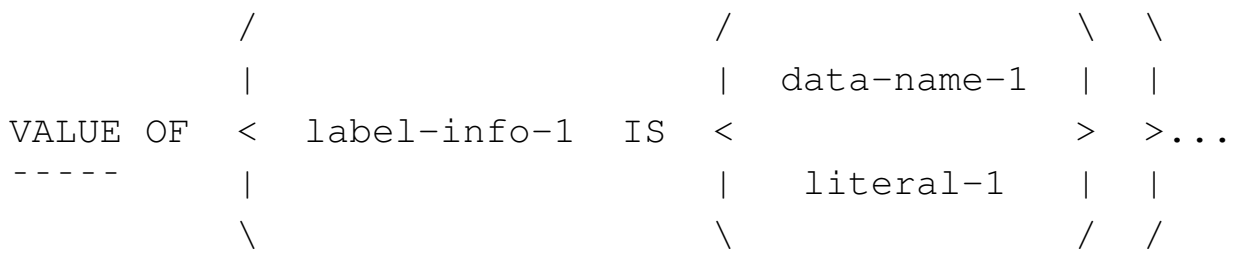
La dichiarazione '**CODE-SET**' riguarda i file a organizzazione sequenziale e serve a specificare l'insieme di caratteri con cui tale file è codificato. Tradizionalmente, questa istruzione è servita per gestire dati in formato EBCDIC, in contrapposizione al codice ASCII, o viceversa.

```
CODE-SET IS alphabet-name
--------
```

Al posto della metavariabile ***alphabet-name*** va inserita una parola che definisce l'insieme di caratteri del file, secondo le aspettative del compilatore utilizzato.

72.5.1.8 Dichiarazione «VALUE OF»

La dichiarazione '**VALUE OF**' consente, in un certo senso, di assegnare dei valori a delle voci legate alle caratteristiche del file. La cosa più importante che si potrebbe fare è di specificare il file da utilizzare secondo ciò che richiede il sistema operativo. Per esempio, se si tratta di un file su disco e il sistema operativo richiede di indicare anche i dischi per nome, il compilatore dovrebbe prevedere qui una voce appropriata.



Le voci che si possono dichiarare qui possono essere di ogni tipo, con la possibilità di abbinare un valore costante (una stringa alfanumerica), oppure una variabile il cui contenuto viene poi modificato in fase elaborativa.

L'estratto seguente di un programma COBOL, scritto per il compilatore TinyCOBOL, mostra l'uso della voce '**FILE-ID**' per dichiarare il nome del file da utilizzare:

```
001000 FILE-CONTROL.
001100     SELECT FILE-NAME ASSIGN TO DISK
001200                      ORGANIZATION IS SEQUENTIAL
001300                      ACCESS MODE IS SEQUENTIAL
001400                      FILE STATUS IS DATA-NAME.
001600 DATA DIVISION.
001700 FILE SECTION.
001800 FD  FILE-NAME
001900     LABEL RECORD IS STANDARD
002000     VALUE OF FILE-ID IS "input.seq".
002100 01  RECORD-NAME  PIC X(20).
002200 WORKING-STORAGE SECTION.
002300 01  DATA-NAME    PIC XX.
```

72.5.1.9 Descrizione del record

«

Dopo ogni indicatore di livello ('**FD**' o '**SD**') si deve descrivere il record attraverso una variabile strutturata, che si dichiara con quelli che sono noti come ***livelli***. I livelli sono in pratica delle dichiarazio-

ni che costituiscono ognuna delle istruzioni singole, ma in tal caso, a differenza delle istruzioni comuni, iniziano con un numero: il numero di livello.

Il livello 01 è obbligatorio e dichiara il nome della variabile strutturata che descrive il record nella sua interezza; qualunque numero superiore serve a descrivere una porzione inferiore del record, con la possibilità di scomposizioni successive. I numeri di livello che possono essere usati per questo scopo sono limitati all'intervallo da 01 a 49, tenendo conto che, a parte l'obbligo di iniziare da 01, i livelli inferiori possono utilizzare incrementi superiori all'unità. Si osservi l'esempio seguente che contiene un estratto dalla sezione '**FILE SECTION**':

```
001600 DATA DIVISION.
001700 FILE SECTION.
001800 FD  SALES-FILE
001830     LABEL RECORD IS STANDARD
001860     VALUE OF FILE-ID IS "sales".
001900 01  SALES-RECORD.
002000     05  SALES-VENDOR-NAME      PIC X(20).
002100     05  SALES-VALUE            PIC S9(6).
002200     05  SALES-NUMBER           PIC X(13).
002300     05  SALES-TYPE             PIC X.
002400     05  SALES-VENDOR-REGION    PIC X(17).
002500     05  SALES-VENDOR-CITY      PIC X(20).
002600     05  SALES-COMMENTS         PIC X(60).
```

Il file individuato dal nome '**SALES-FILE**' si compone di record a cui si può fare riferimento con la variabile strutturata '**SALES-RECORD**'. Il record si suddivide in sette campi con caratteristiche diverse. Il record nella sua interezza corrisponde al livello 01, evidenziato dalla sigla '**01**' che si trova nell'area A del modu-

lo di programmazione. Come si vede nel livello 01 dell'esempio, la variabile strutturata che rappresenta tutto il record viene solo nominata, senza altre indicazioni, perché la sua dimensione si determina dalla somma dei campi che contiene.

I numeri di livello, mano a mano che si annidano in sottolivelli successivi, devono crescere: non è importante se il numero cresce di una o di più unità. Tradizionalmente, i livelli vengono incrementati con un passo maggiore di uno, per facilitare la modifica del sorgente quando dovesse presentarsi l'esigenza di ristrutturare i livelli.

Per comprendere meglio il senso della descrizione del record attraverso il sistema dei livelli, conviene dare un'occhiata allo schema successivo:

registrazione n.	data operazione			carico		scarico		descrizione operazione
	anno	mese	giorno	quantità caricata	costo unitario	quantità scaricata	valore unitario di scarico	

Quello che appare nello schema vuole rappresentare il record di un file da usare per memorizzare carichi e scarichi di un magazzino. Si può osservare inizialmente un campo per numerare le registrazioni (ogni registrazione occupa un record), successivamente, appare la data dell'operazione suddivisa in tre parti (anno, mese e giorno), quindi viene indicato il carico, suddividendo la quantità caricata e il costo unitario di carico, quindi lo scarico, anche questo diviso in quantità scaricata e valore unitario di scarico, infine appare un campo

descrittivo dell'operazione. Un record di questo tipo potrebbe essere descritto utilizzando i livelli nel modo seguente:

```
000000 01  RECORD-MOVIMENTI-DI-MAGAZZINO.
000000     10  MM-NUMERO-REGISTRAZIONE          PIC 99999.
000000     10  MM-DATA-REGISTRAZIONE.
000000         20  MM-DATA-REGISTRAZIONE-ANNO   PIC 9999.
000000         20  MM-DATA-REGISTRAZIONE-MESE   PIC 99.
000000         20  MM-DATA-REGISTRAZIONE-GIORNO PIC 99.
000000     10  MM-CARICO.
000000         20  MM-CARICO-QUANTITA           PIC 9(8)V999.
000000         20  MM-CARICO-COSTO-UNITARIO     PIC 999999V99.
000000     10  MM-SCARICO.
000000         20  MM-SCARICO-QUANTITA          PIC 9(8)V999.
000000         20  MM-SCARICO-VALORE-UNITARIO   PIC 999999V99.
000000     10  MM-DESCRIZIONE                   PIC X(200).
```

Come si può comprendere dall'esempio e come già accennato in precedenza, per le porzioni di record che non si scompongono ulteriormente, si devono specificare le dimensioni, sommando le quali si ottiene la dimensione dei vari raggruppamenti e infine del record complessivo. La sintassi per rappresentare i livelli si potrebbe semplificare in questa fase nel modo seguente, dove però non si usa la notazione standard del linguaggio COBOL:

n n ***nome-campo*** [PIC[TURE] [IS] ***modello_della_variabile*** [***opzioni***]].

Ciò che non è stato descritto fino a questo punto è la parte di dichiarazione successiva al nome del campo, che inizia con la parola chiave '**PICTURE**', spesso abbreviata soltanto con '**PIC**'. Ciò che appare qui serve a definire il modello della variabile, ovvero la sua dimensione e le sue caratteristiche.

Il modello di definizione della variabile è una stringa che va composta seguendo regole precise. Con questo modello si specifica se la variabile è di tipo numerico o alfanumerico, la sua dimensione, la presenza eventuale di una virgola (ovviamente per i valori numerici), il segno ed eventualmente una maschera di trasformazione. Dopo il modello di definizione della variabile possono apparire delle opzioni, in forma di dichiarazioni ulteriori, che servono a precisare la modalità con cui la variabile deve essere rappresentata internamente alla memoria centrale.

Quando si dichiara una variabile numerica, è importante chiarire quale rappresentazione deve avere. A seconda del compilatore, la variabile numerica potrebbe essere gestita in forma binaria oppure in forma BCD (*Binary coded decimal*), che a sua volta può essere «normale», dove ogni cifra occupa un byte, oppure *packed*, dove ogni cifra occupa mezzo byte (4 bit, noto anche come nibble). Questa caratteristica della variabile si definisce con le dichiarazioni opzionali che seguono il modello di definizione della variabile.

Il modo in cui si dichiara il modello di definizione della variabile è descritto nella sezione 72.9, mentre per una visione complessiva del modo in cui si dichiara una variable, si deve consultare la sezione 72.6; tuttavia, in questa fase si può cominciare ugualmente a interpretare l'esempio mostrato in precedenza, osservando in particolare i campi seguenti:

- il campo '**MM-NUMERO-REGISTRAZIONE**' può contenere un numero intero senza segno di cinque cifre: da zero a 99999;

- il campo '**MM-CARICO-QUANTITA**' può contenere un numero senza segno con otto cifre per la parte intera e tre cifre per la parte decimale;
- il campo '**MM-COSTO-UNITARIO**' può contenere un numero senza segno con sei cifre per la parte intera e due cifre per la parte decimale;
- il campo '**MM-DESCRIZIONE**' può contenere caratteri alfanumerici di qualunque tipo (nell'ambito di una rappresentazione in byte), per una dimensione di 200 caratteri.

Nell'esempio del magazzino si può notare che tutti i nomi usati per individuare le varie componenti del record sono unici, ma oltre a questo è stata usata l'accortezza di mettere un prefisso ('**MM-**') per distinguerli rispetto a campi di altri file che potrebbero avere una struttura del record simile. Tuttavia, non è strettamente necessario che tali nomi siano univoci per tutto il programma, perché è prevista la possibilità di qualificarli in modo gerarchico. La qualificazione è descritta nella sezione 72.8.3.

Esiste anche la possibilità di ridefinire la struttura di un record, assegnando un nome alternativo a un certo livello che si vuole descrivere diversamente. Si osservi l'esempio seguente:

```
000000 01  MIO-RECORD.
000000     02  CAMPO-A                              PIC X(20).
000000     02  RIDEFINITO-A REDEFINES CAMPO-A.
000000         03  DATA.
000000             04  ANNO                         PIC 9999.
000000             04  MESE                         PIC 99.
000000             04  GIORNO                       PIC 99.
000000         03  DESCRIZIONE                      PIC X(12).
000000     02  CAMPO-B …
…
```

Nell'esempio si vede un record denominato '**MIO-RECORD**', che inizialmente è composto dal campo '**CAMPO-A**' fatto per contenere 20 caratteri. Questo campo viene ridefinito nella riga successiva con il nome '**RIDEFINITO-A**', che si articola in sottocampi, con i quali si vuole descrivere in modo alternativo la variabile '**CAMPO-A**'. In base al contesto si intende che i primi otto caratteri possano essere interpretati come le cifre numeriche di una data (anno, mese e giorno), individuando il resto come una descrizione non meglio qualificabile.

Generalmente, la ridefinizione di un campo che non è suddiviso è di scarsa utilità, mentre è più interessante quando si applica a campi che hanno già una suddivisione, che però si vuole gestire anche in modo differente:

```
000000 01  MIO-RECORD.
000000     02  A
000000         03  B                                  PIC X(10).
000000         03  C                                  PIC X(10).
000000     02  D REDEFINES A.
000000         03  E                                  PIC X(5).
000000         03  F                                  PIC X(10).
000000         03  G                                  PIC X(5).
000000     02  H …
…
```

In questo caso, il campo ‘**A**’ è composto complessivamente da 20 caratteri, a cui si accede con i campi ‘**B**’ e ‘**C**’ per i primi 10 e gli ultimi 10 rispettivamente. La ridefinizione successiva, consente di accedere a una porzione centrale, a cavallo dei campi ‘**B**’ e ‘**C**’, con il campo ‘**F**’.

72.5.2 Sezione «WORKING-STORAGE SECTION»

La sezione ‘**WORKING-STORAGE SECTION**’ serve a dichiarare le variabili, strutturate o scalari, utilizzate dal programma, che non si riferiscono direttamente alla descrizione dei record dei file:

```
WORKING-STORAGE SECTION.
-----------------------

      .--                            --.
      |  77-level-description-entry    |
      |                                |...
      |  record-description-entry      |
      `--                            --'
```

A differenza della sezione ‘**FILE SECTION**’, oltre alla dichiarazione di variabili strutturate, è possibile dichiarare delle variabili scalari (non suddivisibili), utilizzando il livello speciale numero 77.

```
000000 WORKING-STORAGE SECTION.
000000 01  DATA-DA-SCOMPORRE.
000000     02  ANNO              PIC 9999.
000000     02  MESE              PIC 99.
000000     02  GIORNO            PIC 99.
000000 77  FINE-DEL-FILE         PIC 9.
000000 77  A                     PIC X(10).
000000 77  B                     PIC 9999V99.
```

Il livello 77 viene dichiarato mettendo il numero relativo nella colonna dell'area A del modulo di programmazione, così come si fa per il livello 01; nello stesso modo, il nome della variabile scalare si scrive nell'area B. L'esempio che appare sopra dovrebbe essere sufficiente a comprendere l'uso della sezione '**WORKING-STORAGE SECTION**', tenendo conto che vale quanto descritto a proposito delle variabili strutturate che descrivono i record nella sezione '**FILE SECTION**', compresa la ridefinizione.

La dichiarazione di una variabile scalare con il livello 77 consente di specificare dei tipi numerici binari (come '**USAGE IS INDEX**'), per i quali non si può prevedere la dimensione in modo standard. L'uso di questi tipi numerici speciali non è ammesso nei campi di una variabile scalare descrittiva di un record.

72.5.3 Altri livelli speciali

«

Oltre ai livelli che servono a descrivere le variabili strutturate (da 01 a 49) e le variabili scalari (77), sono previsti due livelli speciali: 66 e 88. Questi livelli speciali servono a definire dei raggruppamenti di

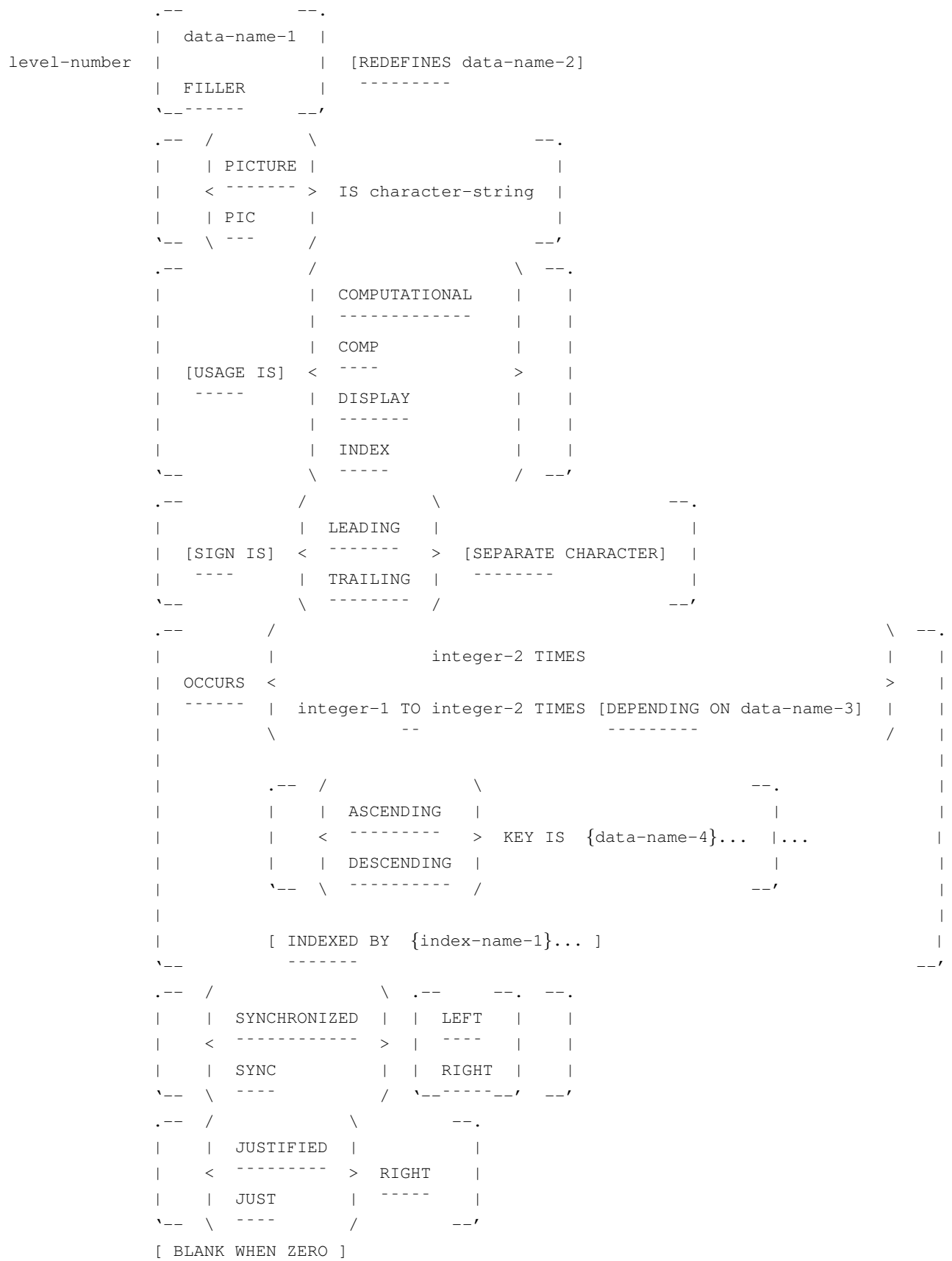
variabili appartenenti alla stessa struttura o a definire dei «nomi di condizione».

La descrizione di questi ulteriori livelli speciali viene fatta nella sezione 72.8.

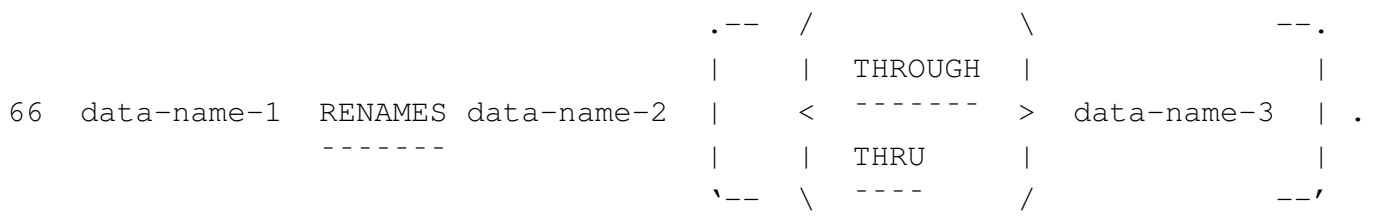
72.6 Descrizione delle variabili

«

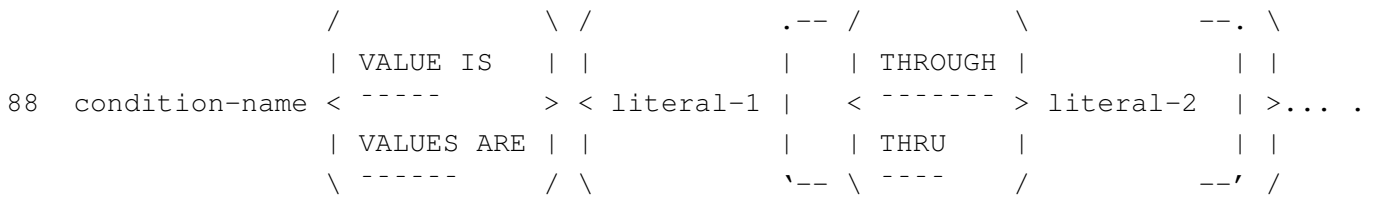
Lo schema sintattico seguente descrive la dichiarazione delle variabili per i livelli da 01 a 49 e per il livello 77:


level-number
data-name-1
FILLER
[REDEFINES data-name-2]
PICTURE
PIC
IS character-string
[USAGE IS]
COMPUTATIONAL
COMP
DISPLAY
INDEX
[SIGN IS]
LEADING
TRAILING
[SEPARATE CHARACTER]
OCCURS
integer-2 TIMES
integer-1 TO integer-2 TIMES [DEPENDING ON data-name-3]
ASCENDING
DESCENDING
KEY IS {data-name-4}...
[INDEXED BY {index-name-1}...]
SYNCHRONIZED
SYNC
LEFT
RIGHT
JUSTIFIED
JUST
[BLANK WHEN ZERO]
[VALUE IS literal-1].

Formato per il livello 66:



Formato per il livello 88:



Nel primo schema, l'indicazione del nome della variabile, che si vede in alternativa alla parola chiave '**FILLER**', deve apparire immediatamente dopo il numero di livello; inoltre, se si usa la parola chiave '**REDEFINES**', questa segue immediatamente il nome della variabile. Il resto delle opzioni può essere inserito con l'ordine che si preferisce, rispettando comunque lo schema e l'eventuale interdipendenza che c'è tra le opzioni stesse.

L'opzione che definisce il modello di definizione della variabile, introdotta dalla parola chiave '**PICTURE**', deve essere usata per ogni variabile che non si scompone ulteriormente e non può essere usata negli altri casi. Nello stesso modo, le opzioni '**SYNCHRONIZED**', '**JUSTIFIED**' e '**BLANK WHEN ZERO**' si possono usare solo per le variabili terminali (che non si scompongono).

Il valore iniziale delle variabili, a eccezione del tipo '**INDEX**', si specifica con l'opzione introdotta dalla parola chiave '**VALUE**'. Quando non si usa o non si può usare questa opzione, il valore iniziale delle variabile non è conosciuto e non può essere previsto.

La descrizione dei nomi di condizione (livello 88) è disponibile nella sezione 72.8.1, mentre per quella riguardante i raggruppamenti alternativi di variabili si può consultare la sezione 72.8.2.

72.6.1 Oggetto della dichiarazione

«

La dichiarazione di una variabile (livelli da 01 a 49 e 77) avviene indicandone il nome subito dopo il numero di livello. Il nome della variable può non essere univoco nell'ambito del programma, purché sia possibile individuare correttamente la variabile attraverso la qualificazione (sezione 72.8.3).

Al posto di indicare il nome di una variabile, è possibile mettere la parola chiave '**`FILLER`**', quando il campo a cui fa riferimento non viene mai utilizzato direttamente. Naturalmente, l'uso di questa parola chiave non è ammissibile in un livello '**`77`**', perché non avrebbe senso dichiarare una variabile scalare, indipendente da qualunque altra, senza avere poi la possibilità di utilizzarla nel programma.

72.6.2 Ridefinizione di una variabile

«

L'opzione '**`REDEFINES`**' può essere usata soltanto quando si dichiara una variabile, nell'ambito dei livelli da 01 a 49 (si esclude quindi l'uso della parola chiave '**`FILLER`**'), allo scopo di ridefinire la struttura di un'altra variabile che si trova allo stesso livello.

```
000000 01  MIO-RECORD.
000000     02  A
000000         03  B                                  PIC X(10).
000000         03  C                                  PIC X(10).
000000     02  D REDEFINES A.
000000         03  E                                  PIC X(5).
000000         03  F                                  PIC X(10).
000000         03  G                                  PIC X(5).
000000     02  H …
…
```

L'esempio mostra che il campo '**A**' è composto complessivamente da 20 caratteri, a cui si accede con i campi '**B**' e '**C**' per i primi 10 e gli ultimi 10 rispettivamente. La ridefinizione successiva, consente di accedere a una porzione centrale, a cavallo dei campi '**B**' e '**C**', con il campo '**F**'.

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   PICTURE-REDEFINES.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-02-25.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 DATA DIVISION.
000900*
001000 WORKING-STORAGE SECTION.
001100 01  RECORD-A.
001200     02  P        PICTURE X(10)      USAGE IS DISPLAY.
001300     02  Q        PICTURE X(10)      USAGE IS DISPLAY.
001400 01  RECORD-B REDEFINES RECORD-A.
001500     02  R        PICTURE X(5)       USAGE IS DISPLAY.
001600     02  S        PICTURE X(10)      USAGE IS DISPLAY.
001700     02  T        PICTURE X(5)       USAGE IS DISPLAY.
```

```
001800*
001900 PROCEDURE DIVISION.
002000*
002100 MAIN.
002200     MOVE "ABCDEFGHIJKLMNOPQRST" TO RECORD-A.
002300     DISPLAY "P: ", P.
002400     DISPLAY "Q: ", Q.
002500     DISPLAY "R: ", R.
002600     DISPLAY "S: ", S.
002700     DISPLAY "T: ", T.
002800*
002900     STOP RUN.
003000*
```

Questo esempio ulteriore mostra un piccolo programma completo, che dimostra il funzionamento della ridefinizione, visualizzando le variabili associate a '**RECORD-A**' e a '**RECORD-B**', che ridefinisce il primo. Se si compila questo programma con TinyCOBOL o con OpenCOBOL, l'avvio dell'eseguibile che si ottiene genera il risultato seguente:

```
P: ABCDEFGHIJ
Q: KLMNOPQRST
R: ABCDE
S: FGHIJKLMNO
T: PQRST
```

72.6.3 Opzione «PICTURE»

«

Attraverso l'opzione '**PICTURE**', che è obbligatoria per le variabili che non si scompongono in livelli inferiori, si definisce precisamente come è fatta la variabile. L'argomento dell'opzione è una stringa che descrive la variabile.

Il modo in cui si rappresenta il modello delle variabili è molto articolato e viene descritto nella sezione 72.9.

72.6.4 Opzione «USAGE»

Per il linguaggio COBOL, il contenuto di una variabile può essere solo di due tipi: alfanumerico o numerico. Una variabile dichiarata per contenere valori alfanumerici, utilizza un formato a «caratteri» (normalmente si tratta della codifica ASCII o EBCDIC), mentre per i valori numerici ci sono delle alternative. Attraverso l'opzione '**USAGE**' si può specificare il modo in cui la variabile deve contenere i dati.

```
                /                    \
                |  COMPUTATIONAL     |
                |  -------------     |
                |  COMP              |
[USAGE IS]      <  ----              >
 -----          |  DISPLAY           |
                |  -------           |
                |  INDEX             |
                \  -----             /
```

Il formato definito dalla parola chiave '**DISPLAY**' corrisponde a un'organizzazione a caratteri. Si tratta evidentemente del formato necessario per le variabili alfanumeriche, ma può essere usato anche per le variabili numeriche.

Il formato '**COMPUTATIONAL**', abbreviabile anche con '**COMP**', può essere usato soltanto per i valori numerici e richiede l'utilizzo di 4 bit per ogni cifra numerica e anche per il segno (se specificato).

Il formato '**INDEX**' serve a dichiarare una variabile speciale, che può rappresentare soltanto numeri interi senza segno. Non si può asso-

ciare un modello di definizione della variabile quando il formato è '**INDEX**', perché la dimensione è fissa e dipende dal compilatore. L'uso di queste variabili è anche limitato a certi contesti (di solito ne è ammesso l'uso negli indici delle tabelle ed eventualmente nei contatori delle iterazioni) e queste variabili si dichiarano soltanto con un livello 77 in modo da essere svincolati da una struttura a record.

L'opzione '**USAGE**' va usata principalmente nella descrizione di variabili terminali; teoricamente si può indicare l'opzione '**USAGE**' anche per variabili che si suddividono in campi e in tal caso i livelli gerarchicamente inferiori ereditano l'opzione da quelli superiori. Tuttavia, è meglio evitare questa pratica e limitarsi a usare l'opzione '**USAGE**' soltanto sui campi terminali, tenendo conto che così i campi composti sono implicitamente di tipo '**DISPLAY**'.

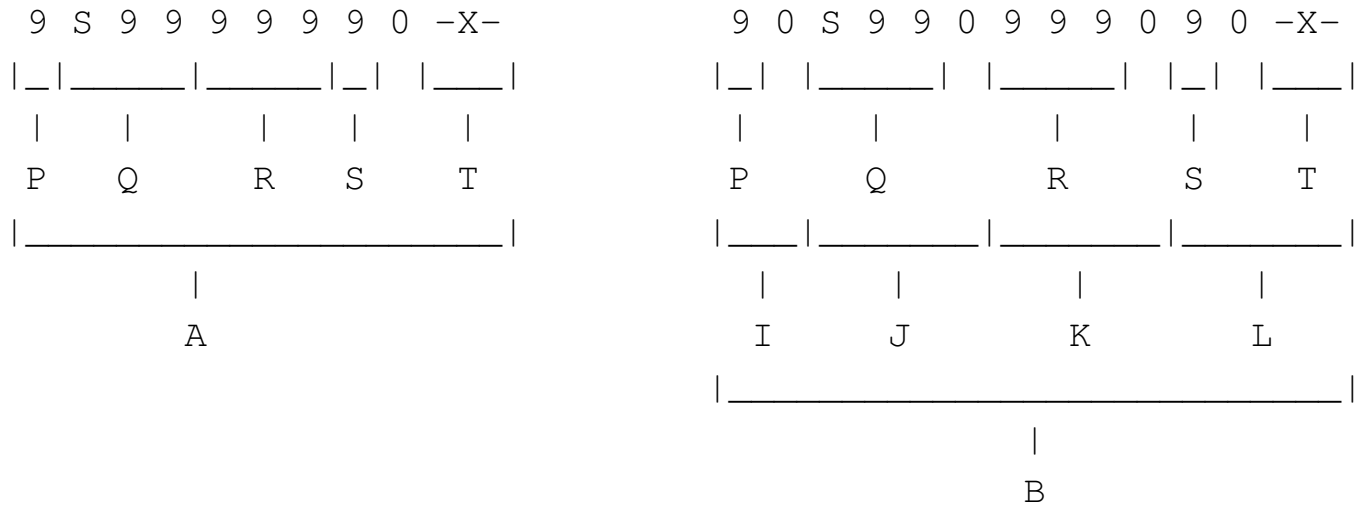
Volendo assegnare a una variabile strutturata un tipo diverso da '**DISPLAY**', si otterrebbe solamente di assegnare questa caratteristica ai livelli inferiori, perché una variabile strutturata, ammesso che la si voglia utilizzare direttamente, nel suo complesso, può funzionare soltanto come '**DISPLAY**'. Come conseguenza, una variabile strutturata può essere formata soltanto da un numero intero di byte.

Quando si dichiarano su uno stesso livello variabili numeriche organizzate secondo il formato '**COMPUTATIONAL**', queste utilizzano regolarmente lo spazio loro assegnato, condividendo eventualmente i byte che dovessero trovarsi a cavallo tra una e l'altra; tuttavia, nell'ambito della variabile composta che contiene questi livelli, i dati devono occupare una quantità di byte intera, pertanto si può perdere

eventualmente un po' di spazio alla fine.

Figura 72.75. Confronto fra due variabili strutturate.

```
01  A.                                  01  B.
                                            02  I.
    03  P PIC 9    COMP.                        03  P PIC 9    COMP.
                                            02  J.
    03  Q PIC S99  COMP.                        03  Q PIC S99  COMP.
                                            02  K.
    03  R PIC 999  COMP.                        03  R PIC 999  COMP.
                                            02  L.
    03  S PIC 9    COMP.                        03  S PIC 9    COMP.
    03  T PIC X.                                03  T PIC X.
```



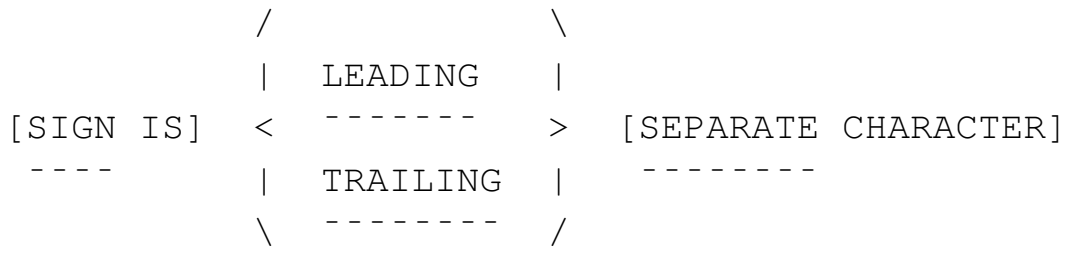
La figura dovrebbe essere sufficiente per capire come viene utilizzata la memoria per la rappresentazione delle variabili '**COMPUTATIONAL**'. Si può osservare che la variabile strutturata '**A**' ha soltanto uno spreco di una cifra '**COMPUTATIONAL**', ovvero 4 bit, mentre la variabile '**B**', avendo un'organizzazione differente nella sua struttura sottostante, spreca più spazio.

Molti compilatori COBOL considerano anche il tipo numerico '**COMPUTATIONAL-3**', o '**COMP-3**'. Si tratta di una variante del tipo '**COMPUTATIONAL**', con l'indicazione del segno o dell'assenza del segno, nel gruppo di 4 bit meno significativo (quello più a destra).

La maggior parte dei manuali COBOL sottolinea il fatto che per eseguire delle elaborazioni numeriche (dei calcoli matematici di qualunque tipo) è bene che le variabili utilizzate siano di tipo '**COMPUTATIONAL**', perché se le variabili sono ti tipo '**DISPLAY**', prima di poter essere utilizzate devono essere convertite.

72.6.5 Opzione «SIGN»

Quando si dichiarano variabili numeriche che prevedono l'indicazione del segno, è possibile stabilire in che modo e in quale posizione deve trovarsi, con l'opzione '**SIGN**':

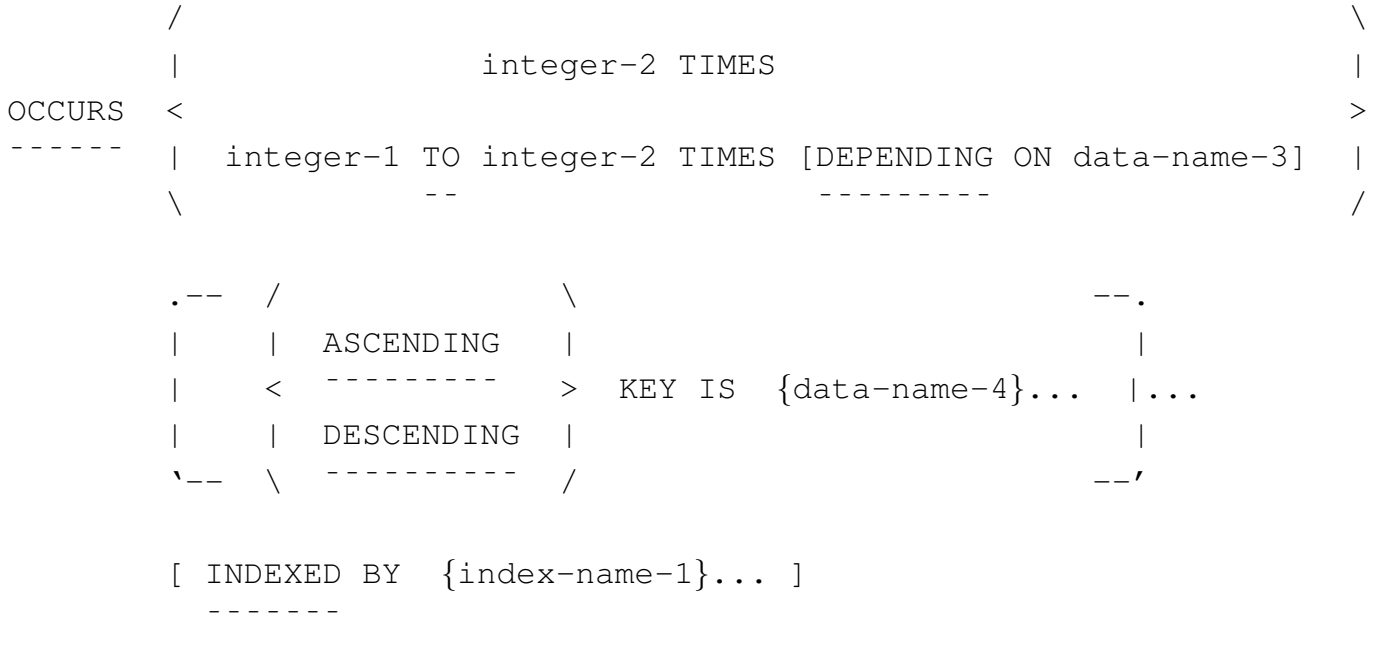


Le variabili numeriche di tipo '**DISPLAY**', in condizioni normali, incorporano il segno nel byte più significativo. Quando si vuole richiedere che il segno occupi un byte tutto per sé, è necessario usare la parola chiave '**SEPARATE**'. Per le variabili di tipo '**COMPUTATIONAL**' il segno occupa sempre uno spazio separato rispetto a quello delle cifre numeriche.

Se si utilizza la parola chiave '**LEADING**', il segno viene collocato a sinistra (e di norma questo è il comportamento predefinito); se invece si usa la parola chiave '**TRAILING**', il segno viene collocato nella posizione più a destra.

72.6.6 Opzione «OCCURS»

La parola chiave '**OCCURS**' introduce un gruppo di informazioni che consentono di indicare che la variabile a cui fanno riferimento viene ripetuta secondo certe modalità. Attraverso questo meccanismo si creano quelle che per il COBOL sono delle tabelle.



La variabile ricorrente di una tabella può ripetersi per un numero fisso di elementi (***integer-2***), oppure per un numero variabile, nell'ambito di un intervallo stabilito (da ***integer-1*** a ***integer-2***), sotto il controllo di un'altra variabile (***data-name-3***).

Se l'insieme degli elementi della tabella dichiarata con l'opzione '**OCCURS**' è ordinato in base a una chiave, questa può essere specificata (***index-name-1***); inoltre, l'indice per accedere agli

elementi può essere dichiarato contestualmente già in questa fase (***index-name-1***).

Per maggiori dettagli, si veda la sezione 72.7, dedicata solo alle tabelle del COBOL.

72.6.7 Opzione «SYNCHRONIZED»

«

L'opzione '**SYNCHRONIZED**' avrebbe lo scopo di allineare una variabile numerica nei limiti di una o più «parole», secondo l'architettura usata nell'elaboratore per il quale è realizzato il compilatore.

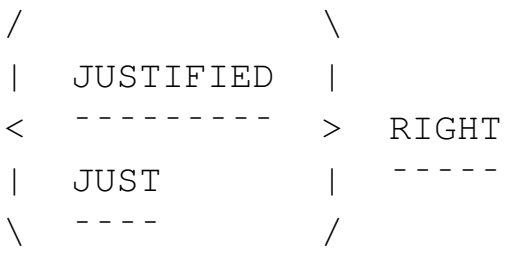
```
/                \   .--      --.
|  SYNCHRONIZED  |   |  LEFT    |
<  ------------  >   |  ----    |
|  SYNC          |   |  RIGHT   |
\  ----          /   `--_____--'
```

Teoricamente, l'uso dell'opzione '**SYNCHRONIZED**' avrebbe lo scopo di facilitare le elaborazioni numeriche, ma si creano delle complicazioni nel modo di determinare la dimensione effettiva delle variabili, soprattutto quando si vogliono ridefinire.

Per scrivere programmi COBOL compatibili tra i vari sistemi operativi e le architetture fisiche degli elaboratori, è meglio evitare l'uso di variabili '**SYNCHRONIZED**'; tuttavia, se si preferisce comunque usare questa funzione, diventa necessario consultare il manuale specifico del proprio compilatore.

72.6.8 Opzione «JUSTIFIED RIGHT»

Le variabili alfabetiche o alfanumeriche possono essere dichiarate con l'opzione '**JUSTIFIED**', per fare in modo che ricevano i dati allineandoli a destra:

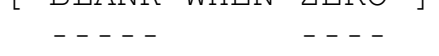


Normalmente, quando si «invia» una stringa in una variabile alfanumerica, la copia inizia da sinistra a destra: se la variabile ricevente è più piccola della stringa, questa viene troncata alla destra; se la variabile ricevente è più grande, si aggiungono alla destra degli spazi.

Quando si usa l'opzione '**JUSTIFIED**' per una variabile (ricevente), la copia di una stringa avviene con un allineamento opposto, pertanto il troncamento avviene a sinistra e così anche l'aggiunta degli spazi ulteriori.

72.6.9 Opzione «BLANK WHEN ZERO»

L'opzione '**BLANK WHEN ZERO**' si può utilizzare solo per le variabili numeriche scalari, dove ogni cifra utilizza un byte intero. L'opzione fa sì che gli zeri anteriori vengano sostituiti da spazi, a meno che il modello di definizione della variabile preveda diversamente.



72.6.10 Opzione «VALUE»

«

Convenzionalmente, una variabile che viene dichiarata nei livelli da 01 a 49 e 77, non ha inizialmente un valore prestabilito, ma solo casuale. Per stabilire un valore da attribuire a una variabile nel momento della sua creazione, si usa l'opzione '**VALUE**':

```
VALUE IS literal-1
-----
```

La costante che nello schema sintattico è indicata come ***literal-1***, è il valore che viene attribuito inizialmente.

Si osservi che è possibile stabilire un valore iniziale per una variabile, soltanto quando si tratta di qualcosa che viene dichiarato nella sezione '**WORKING-STORAGE SECTION**', perché nella sezione '**FILE SECTION**' ciò non è possibile e non avrebbe senso.

L'opzione '**VALUE**' si usa anche per la dichiarazione dei nomi di condizione, ma in tal caso la funzione di questa opzione ha un valore differente e non c'è più la discriminazione tra le sezioni in cui si può utilizzare.

72.6.11 Opzione «RENAMES»

«

L'opzione '**RENAMES**', che si usa nel livello 66, permette di dichiarare delle variabili che rappresentano un raggruppamento di altre variabili, appartenenti alla stessa struttura, purché queste siano adiacenti. Nella sezione 72.8.2 viene mostrata la dichiarazione dei raggruppamenti.

72.7 Tabelle

Il linguaggio COBOL offre la gestione di array attraverso la definizione di variabili multiple, all'interno di variabili strutturate (record); tuttavia, la denominazione usata nel COBOL per queste rappresentazioni dei dati è di ***tabella***.

```
level-number  data-name-1

              [omissis]

                        /                                                   \
                        |               integer-2 TIMES                     |
                 OCCURS <                                                    >
                 ------ |  integer-1 TO integer-2 TIMES [DEPENDING ON data-name-3]  |
                        \             --                  ---------          /

                        .--  /              \                          --.
                        |    |  ASCENDING   |                            |
                        |    <  ----------  >  KEY IS  {data-name-4}...  |...
                        |    |  DESCENDING  |                            |
                        '--  \  ----------  /                          --'

                        [ INDEXED BY  {index-name-1}... ]
                          -------

              [omissis] .
```

72.7.1 Dichiarazione di una tabella

Si dichiara che un campo è composto da più elementi dello stesso tipo aggiungendo in coda l'opzione '**`OCCURS` *n* `TIMES`**'. Lo schema sintattico completo dell'opzione è il seguente:

```
         /                                                   \
         |               integer-2 TIMES                     |
OCCURS   <                                                    >
------   |  integer-1 TO integer-2 TIMES [DEPENDING ON data-name-3]  |
         \             --                  ---------          /

         .--  /              \                          --.
```

```
|    |  ASCENDING   |                                |
|    <  ---------   >  KEY IS  {data-name-4}...  |...
|    |  DESCENDING  |                                |
`--  \  ----------  /                              --'

[ INDEXED BY  {index-name-1}... ]
  -------
```

Le tabelle più semplici sono quelle che hanno un numero fisso di elementi. Si osservi l'esempio seguente:

```
000000 01  A.
000000     02  B         PIC 9999.
000000     02  C                          OCCURS 10 TIMES.
000000         03  D     PIC X(10).
000000         03  E     PIC 99           OCCURS  7 TIMES.
000000     02  F         PIC X(10).
```

Nell'esempio viene dichiarata una variabile strutturata denominata '**A**', che si articola nelle variabili '**B**', '**C**' e '**F**'. La variabile '**C**' è ripetuta per 10 volte e si articola ogni volta nella variabile '**D**' e nella variabile '**E**'. La variabile '**E**' si ripete per sette volte.

La variabile '**E**' è una tabella a due dimensioni, perché è inclusa nelle ripetizioni della variabile '**C**', mentre la variabile '**C**' è una tabella a una sola dimensione.

È evidente che per fare riferimento ai valori contenuti nelle tabelle sia necessario utilizzare un indice.

L'opzione '**OCCURS**' si può utilizzare per tutte le variabili dichiarate con un numero di livello da 02 a 49. In pratica vengono esclusi i livelli 01 (record), 66 (usato per il raggruppamento delle variabili), 77 (usato esclusivamente per le variabili scalari) e 88 (nomi di condizione).

Lo standard del 1974 del linguaggio COBOL pone come limite un massimo di tre dimensioni per le tabelle.

72.7.2 Riferimento al contenuto di una tabella

Per fare riferimento a un elemento di una tabella, nelle istruzioni della divisione '**PROCEDURE DIVISION**' si usa una forma descritta dallo schema sintattico seguente:

```
data-name (subscript-1 [subscript-2 [subscript-3...]])
```

In pratica, si scrive il nome della variabile ripetuta, seguita dall'indice o dagli indici tra parentesi tonde. Il primo indice riguarda la prima dimensione, intesa come quella più esterna; l'ultimo riguarda l'annidamento più interno.

L'indice è un numero intero positivo che va da uno fino al massimo della dimensione che lo riguarda. Seguendo l'esempio apparso nella sezione precedente, '**E (1 7)**' rappresenta la settima occorrenza della variabile '**E**' nell'ambito della prima della variabile '**C**'. Pertanto, il nome da usare per indicare l'elemento è quello della variabile più interna che si vuole individuare, mentre gli indici partono dalla posizione più esterna.

Si noti che è convenzione comune inserire delle virgole per separare gli indici, anche se si tratta di una forma di rappresentazione facoltativa.

Viene mostrato un altro esempio di tabella a tre dimensioni:

```
000000 01  ENCICLOPEDIA.
000000     05  VOLUME                    OCCURS 10 TIMES.
000000         10  TITOLO-VOLUME             PIC X(30).
000000         10  PARTE                 OCCURS 20 TIMES.
000000             15  TITOLO-PARTE          PIC X(30).
000000             15  CAPITOLO          OCCURS 30 TIMES.
000000                 20  TITOLO-CAPITOLO  PIC (30).
000000                 20  TESTO            PIC (200).
```

Si tratta di una variabile strutturata che serve a contenere delle informazioni su un'enciclopedia. L'elemento '**VOLUME (5)**' contiene le informazioni su tutto il volume quinto; l'elemento '**TITOLO-VOLUME (5)**' contiene il titolo del volume quinto; l'elemento '**TITOLO-PARTE (5, 3)**' contiene il titolo della terza parte del volume quinto; l'elemento '**TESTO (5, 3, 25)**' contiene il testo del venticinquesimo capitolo contenuto nella terza parte del quinto volume. Naturalmente, in questo esempio si intende che la numerazione delle parti ricominci da uno all'inizio di ogni volume; così si intende che all'inizio di ogni parte la numerazione dei capitoli riprenda da uno.

72.7.3 Indice

«

L'indice di una tabella può essere indicato attraverso una costante numerica, una variabile numerica a cui sia stato attribuito preventiva-

mente un valore appropriato o attraverso un'espressione elementare che risulta in un numero intero appropriato.

Quando si usa una variabile per la gestione di un indice, è possibile ed è consigliabile che il tipo numerico di tale variabile sia '**INDEX**'. In pratica, nella sezione '**WORKING-STORAGE SECTION**' un indice potrebbe essere dichiarato come nell'esempio seguente, dove se ne vedono due, il primo, denominato '**INDICE**', è dichiarato come variabile scalare di livello 77, il secondo, denominato '**INDICE-C**', è sempre costituito da una variabile scalare, che però fa parte di una variabile strutturata:

```
000000 77  INDICE           USAGE IS INDEX.
000000
000000 01  A.
000000     02  B             PIC X(30).
000000     02  INDICE-C      USAGE IS INDEX.
```

Si può osservare che questo tipo di variabile numerica non prevede la definizione della sua dimensione che è stabilita invece dal compilatore, in base alle caratteristiche dell'elaboratore e del sistema operativo.

In alternativa, **l'indice può essere dichiarato contestualmente alla dichiarazione della variabile ricorrente**; in tal caso, il compilatore può aggiungere dei controlli tali da impedire che si possa assegnare alla variabile un valore al di fuori dell'ambito della dimensione della tabella:

```
000000 01  ENCICLOPEDIA.
000000     05  VOLUME                      OCCURS 10 TIMES
000000                                     INDEX IS IND-VOLUME.
000000         10  TITOLO-VOLUME              PIC X(30).
000000         10  PARTE                   OCCURS 20 TIMES
000000                                     INDEX IS IND-PARTE.
000000             15  TITOLO-PARTE           PIC X(30).
000000             15  CAPITOLO            OCCURS 30 TIMES
000000                                     INDEX IS IND-CAPITOLO.
000000                 20  TITOLO-CAPITOLO  PIC (30).
000000                 20  TESTO            PIC (200).
```

Qui viene ripreso e modificato un esempio già apparso in una sezione precedente. La differenza consiste nell'assegnare l'indice a ogni variabile ricorrente. Pertanto si hanno gli indici: '**IND-VOLUME**', '**IND-PARTE**' e '**IND-CAPITOLO**'.

Come accennato, si può fare riferimento a un elemento di una tabella indicando un indice costituito da un'espressione matematica elementare. Si parla in questo caso di indici relativi, perché si possono sommare o sottrarre dei valori a partire da una posizione di partenza. Per esempio, supponendo che la variabile '**I**' contenga il numero 15, l'elemento indicato come '**ELE (I - 4)**' corrisponderebbe alla notazione '**ELE (11)**'; nello stesso modo, l'elemento indicato come '**ELE (I + 4)**' corrisponderebbe alla notazione '**ELE (19)**'.

72.7.4 Tabelle di dimensione variabile

«

Teoricamente, è possibile dichiarare l'occorrenza di una variabile per una quantità variabile di elementi; si usa in tal caso la forma '**OCCURS *m* TO *n* TIMES**'. A seconda del compilatore, può essere obbligatorio, o facoltativo, specificare il nome di una variabile che controlla dinamicamente la quantità massima di elementi:

```
OCCURS  integer-1 TO integer-2 TIMES [DEPENDING ON data-name-3]
------              --                 ---------
```

Viene mostrato l'esempio di un programma completo, che serve ad accumulare in una tabella alcuni dati personali. Sono previsti un massimo di 60 elementi e la quantità effettiva di elementi è controllata dalla variabile '**`UTENTI-MAX`**':

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   ELM1150.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-02-24.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 DATA DIVISION.
000900*
001000 WORKING-STORAGE SECTION.
001100 01  RECORD-UTENTI.
001200     02  UTENTE           OCCURS 1 TO 60 TIMES
001300                          DEPENDING ON UTENTI-MAX
001400                          INDEXED BY IND-UTENTE.
001500         03  COGNOME      PIC X(30).
001600         03  NOME         PIC X(30).
001700         03  NOTA         PIC X(200).
001800 77  UTENTI-MAX           USAGE IS INDEX.
001900 77  EOJ                  PIC 9   VALUE ZERO.
002000 77  RISPOSTA             PIC XX.
002100*
002200 PROCEDURE DIVISION.
002300*------------------------ LIVELLO 0 -----------------
002400 MAIN.
002500     PERFORM INSERIMENTO-DATI
002600             VARYING IND-UTENTE FROM 1 BY 1
002700             UNTIL EOJ = 1.
```

```
002800     PERFORM SCANSIONE
002900             VARYING IND-UTENTE FROM 1 BY 1
003000             UNTIL IND-UTENTE > UTENTI-MAX.
003100*
003200     STOP RUN.
003300*------------------------ LIVELLO 1 -----------------
003400 INSERIMENTO-DATI.
003500     MOVE IND-UTENTE TO UTENTI-MAX.
003600     DISPLAY "INSERISCI IL COGNOME: ".
003700     ACCEPT COGNOME (IND-UTENTE).
003800     DISPLAY "INSERISCI IL NOME: ".
003900     ACCEPT NOME (IND-UTENTE).
004000     DISPLAY "INSERISCI UNA NOTA DESCRITTIVA: ".
004100     ACCEPT NOTA (IND-UTENTE).
004200*
004300     IF IND-UTENTE >= 60
004400       THEN
004500           MOVE 1 TO EOJ;
004600       ELSE
004700           DISPLAY "VUOI CONTINUARE? SI O NO",
004800           ACCEPT RISPOSTA;
004900           IF RISPOSTA = "SI"
005000             THEN
005100                 DISPLAY "OTTIMO!";
005200             ELSE
005300                 MOVE 1 TO EOJ.
005400*-----------------------------------------------
005500 SCANSIONE.
005600     DISPLAY COGNOME (IND-UTENTE), " ",
005700             NOME (IND-UTENTE), " ",
005800             NOTA (IND-UTENTE).
005900*
```

72.7.5 Tabelle ordinate

Se si devono utilizzare i dati in una tabella per eseguire una ricerca al suo interno (utilizzando l'istruzione '**SEARCH**' nella divisione '**PROCEDURE DIVISION**'), se si può essere certi che le informazioni contenute siano ordinate secondo una certa chiave, lo si può specificare nella dichiarazione:

```
          /                                                          \
          |               integer-2 TIMES                            |
OCCURS    <                                                          >
------    |  integer-1 TO integer-2 TIMES [DEPENDING ON data-name-3]  |
          \             --                   ---------               /

          .--  /              \                            --.
          |    |  ASCENDING   |                              |
          |    <  ---------   >  KEY IS  {data-name-4}...   |...
          |    |  DESCENDING  |                              |
          `--  \  ----------  /                            --'

          [ INDEXED BY  {index-name-1}... ]
            -------
```

La metavariabile ***data-name-4*** dello schema sintattico rappresenta una variabile contenuta nell'elemento ricorrente; attraverso la parola chiave '**ASCENDING**' si intende dichiarare che la tabella è ordinata, lessicograficamente, in modo ascendente, secondo il contenuto di quella variabile, se invece si usa la parola chiave '**DESCENDING**', si intende un ordinamento decrescente.

È possibile specificare più chiavi di ordinamento successive, nel caso si vogliano abbinare chiavi secondarie di ordinamento.

Sia chiaro che la tabella deve già risultare ordinata secondo le chiavi specificate, altrimenti le istruzioni '**SEARCH**' della divisione '**PROCEDURE DIVISION**' danno risultati errati o falliscono semplicemente. Naturalmente, all'interno del programma è possibile prevedere un procedimento di riordino, da eseguire prima di utilizzare delle istruzioni '**SEARCH**'.

L'esempio seguente mostra l'indicazione della chiave di ordinamento, costituita precisamente dalla variabile '**COGNOME**', che deve risultare ascendente in fase di ricerca:

```
001000 WORKING-STORAGE SECTION.
001100 01  RECORD-UTENTI.
001200     02  UTENTE              OCCURS 1 TO 60 TIMES
001300                             DEPENDING ON UTENTI-MAX
001350                             ASCENDING KEY IS COGNOME
001400                             INDEXED BY IND-UTENTE.
001500         03  COGNOME         PIC X(30).
001600         03  NOME            PIC X(30).
001700         03  NOTA            PIC X(200).
001800 77  UTENTI-MAX              USAGE IS INDEX.
```

72.7.6 Scansione delle tabelle

«

Il linguaggio COBOL prevede un'istruzione apposita per facilitare la scansione delle tabelle. Si tratta di '**SEARCH**', che ha due modalità di funzionamento, a seconda che si voglia eseguire una ricerca sequenziale o una ricerca binaria. Naturalmente, la ricerca sequenziale si presta alla scansione di una tabella i cui dati non sono ordinati, mentre nel secondo caso devono esserlo.

Viene mostrato l'esempio di un programma completo che inizia con l'inserimento di dati all'interno di una tabella, quindi esegue una ricerca sequenziale al suo interno:

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   TEST-SEARCH.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 INSTALLATION. NANOLINUX IV,
000500               OPENCOBOL 0.31,
000600 DATE-WRITTEN. 2005-03-12.
000700*
000800 ENVIRONMENT DIVISION.
000900*
001000 DATA DIVISION.
001100*
001200 WORKING-STORAGE SECTION.
001300 01  RECORD-UTENTI.
001400     02  UTENTE              OCCURS 60 TIMES
001500                              INDEXED BY IND-UTENTE.
001600         03  COGNOME         PIC X(30).
001700         03  NOME            PIC X(30).
001800         03  NOTA            PIC X(200).
001900 77  EOJ                     PIC 9   VALUE ZERO.
002000 77  RISPOSTA                PIC XX.
002100 77  RICERCA                 PIC X(30).
002200*
002300 PROCEDURE DIVISION.
002400*------------------------ LIVELLO 0 -----------------
002500 MAIN.
002600     PERFORM INSERIMENTO-DATI
002700             VARYING IND-UTENTE FROM 1 BY 1
002800             UNTIL EOJ = 1.
002900     MOVE 0 TO EOJ.
003000     PERFORM SCANSIONE UNTIL EOJ = 1.
003100*
```

```
003200     STOP RUN.
003300*------------------------ LIVELLO 1 -----------------
003400 INSERIMENTO-DATI.
003500     DISPLAY IND-UTENTE, " INSERISCI IL COGNOME: ".
003600     ACCEPT COGNOME (IND-UTENTE).
003700     DISPLAY IND-UTENTE, " INSERISCI IL NOME: ".
003800     ACCEPT NOME (IND-UTENTE).
003900     DISPLAY IND-UTENTE,
003950             " INSERISCI UNA NOTA DESCRITTIVA: ".
004000     ACCEPT NOTA (IND-UTENTE).
004100*
004200     IF IND-UTENTE >= 60
004300       THEN
004400           MOVE 1 TO EOJ;
004500       ELSE
004600           DISPLAY "VUOI CONTINUARE? SI O NO",
004700           ACCEPT RISPOSTA;
004800           IF RISPOSTA = "SI"
004900             THEN
005000                 NEXT SENTENCE;
005100             ELSE
005200                 MOVE 1 TO EOJ.
005300*-------------------------------------------------
005400 SCANSIONE.
005500     DISPLAY "INSERISCI IL COGNOME DA CERCARE:".
005600     ACCEPT RICERCA.
005700     IF RICERCA = SPACES
005800       THEN
005900           MOVE 1 TO EOJ;
006000       ELSE
006100           SET IND-UTENTE TO 1,
006200           SEARCH UTENTE
006300                AT END
006400                     DISPLAY "IL COGNOME CERCATO ",
```

```
006500                                   "NON SI TROVA NELLA ",
006550                                   "TABELLA: ",
006600                                   QUOTE RICERCA QUOTE;
006700                 WHEN COGNOME (IND-UTENTE) = RICERCA
006800                     DISPLAY "IL COGNOME ", RICERCA,
006900                             "SI TROVA NELLA ",
006950                             "POSIZIONE ",
007000                             IND-UTENTE.
007100*
```

Nell'esempio sono evidenziate le righe in cui si dichiara la tabella e quelle che eseguono la scansione. Si deve osservare che prima dell'istruzione '**SEARCH**', l'indice deve essere collocato manualmente nella posizione iniziale.

L'esempio seguente mostra una variante del programma già descritto, in cui si vuole eseguire una ricerca binaria. Perché la ricerca possa avere successo, la tabella deve essere dichiarata con una dimensione variabile di elementi, inoltre non è più necessario impostare il valore iniziale dell'indice, prima della scansione.

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   TEST-SEARCH-KEY.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 INSTALLATION. NANOLINUX IV,
000500               OPENCOBOL 0.31,
000600 DATE-WRITTEN. 2005-03-12.
000700*
000800 ENVIRONMENT DIVISION.
000900*
001000 DATA DIVISION.
001100*
001200 WORKING-STORAGE SECTION.
001300 01  RECORD-UTENTI.
```

```
001400     02  UTENTE              OCCURS 1 TO 60 TIMES
001500                             DEPENDING ON UTENTI-MAX
001600                             ASCENDING KEY IS COGNOME
001700                             INDEXED BY IND-UTENTE.
001800         03  COGNOME         PIC X(30).
001900         03  NOME            PIC X(30).
002000         03  NOTA            PIC X(200).
002100 77  UTENTI-MAX              USAGE IS INDEX.
002200 77  EOJ                     PIC 9   VALUE ZERO.
002300 77  RISPOSTA                PIC XX.
002400 77  RICERCA                 PIC X(30).
002500*
002600 PROCEDURE DIVISION.
002700*------------------------ LIVELLO 0 -----------------
002800 MAIN.
002900     PERFORM INSERIMENTO-DATI
003000             VARYING IND-UTENTE FROM 1 BY 1
003100             UNTIL EOJ = 1.
003200     MOVE 0 TO EOJ.
003300     PERFORM SCANSIONE UNTIL EOJ = 1.
003400*
003500     STOP RUN.
003600*------------------------ LIVELLO 1 -----------------
003700 INSERIMENTO-DATI.
003800     MOVE IND-UTENTE TO UTENTI-MAX.
003900     DISPLAY IND-UTENTE, " INSERISCI IL COGNOME: ".
004000     ACCEPT COGNOME (IND-UTENTE).
004100     DISPLAY IND-UTENTE, " INSERISCI IL NOME: ".
004200     ACCEPT NOME (IND-UTENTE).
004300     DISPLAY IND-UTENTE,
004350             " INSERISCI UNA NOTA DESCRITTIVA: ".
004400     ACCEPT NOTA (IND-UTENTE).
004500*
004600     IF IND-UTENTE >= 60
```

```
004700       THEN
004800           MOVE 1 TO EOJ;
004900       ELSE
005000           DISPLAY "VUOI CONTINUARE? SI O NO",
005100           ACCEPT RISPOSTA;
005200           IF RISPOSTA = "SI"
005300             THEN
005400                 NEXT SENTENCE;
005500             ELSE
005600                 MOVE 1 TO EOJ.
005700*-------------------------------------------------
005800 SCANSIONE.
005900     DISPLAY "INSERISCI IL COGNOME DA CERCARE:".
006000     ACCEPT RICERCA.
006100     IF RICERCA = SPACES
006200       THEN
006300           MOVE 1 TO EOJ;
006400       ELSE
006600           SEARCH ALL UTENTE
006700                AT END
006800                     DISPLAY "IL COGNOME CERCATO ",
006900                             "NON SI TROVA NELLA ",
006950                             "TABELLA: ",
007000                             QUOTE RICERCA QUOTE;
007100                WHEN COGNOME (IND-UTENTE) = RICERCA
007200                    DISPLAY "IL COGNOME ", RICERCA,
007300                            "SI TROVA NELLA ",
007350                            "POSIZIONE ",
007400                            IND-UTENTE.
007500*
```

La ricerca binaria richiede che gli elementi della tabella siano ordinati in base alla chiave primaria; pertanto, si presume che l'inserimento dei cognomi avvenga tenendo conto dell'ordine

lessicografico.

72.8 Nomi di condizione, raggruppamento e qualificazione

«

Per rappresentare in modo immediato una condizione che verifichi la corrispondenza tra il contenuto di una variabile e un insieme di valori prestabiliti, si possono utilizzare i ***nomi di condizione***. I nomi di condizione sono sostanzialmente delle costanti che descrivono un'espressione condizionale di questo tipo e si dichiarano con il numero di livello 88.

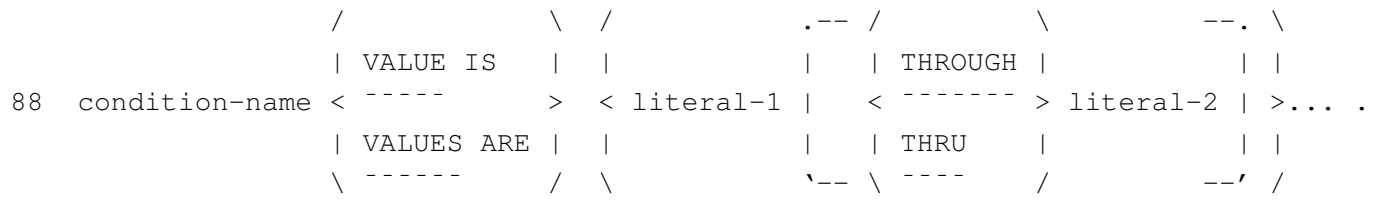
Le variabili possono essere raggruppate diversamente, purché si trovino all'interno di una stessa variabile strutturata e siano adiacenti.

I nomi dati alle variabili e ad altri oggetti del programma, in certe situazioni possono richiedere la «qualificazione», che consiste nello specificare il contesto al quale si riferiscono.

72.8.1 Nomi di condizione

«

Attraverso il livello 88 è possibile definire una costante speciale, con lo scopo di rappresentare la possibilità che una variabile contenga certi valori. In pratica, si dichiara una variabile, con i livelli da 01 a 49 o anche con il livello 77. A questa variabile si abbinano una o più costanti dichiarate con il livello 88, che descrivono la possibilità che la variabile a cui si riferiscono contenga un certo insieme di valori. Le costanti di livello 88, dichiarate in questo modo, si usano poi nelle condizioni.



Nello schema sintattico si vede soltanto la dichiarazione del livello 88, ma si deve tenere conto che si tratta di istruzioni che vanno collocate a partire dalla riga successiva a quella di dichiarazione della variabile a cui si riferiscono.

La parola chiave '**THROUGH**' o '**THRU**' si usa per specificare un intervallo di valori (dalla costante ***literal-1*** a ***literal-2***. Il valore singolo o l'intervallo di valori, può essere seguito da altri.

Nell'esempio seguente, si vede la dichiarazione della variabile '**VERNICIATURA**' che può contenere una stringa di 30 caratteri (alfanumerica). A questa variabile si associano due nomi di condizione, '**VERNICE-CHIARA**' e '**VERNICE-SCURA**', che servono a definire due gruppi di colori, descritti per nome. Da questo si intuisce che nella parte procedurale del programma venga attribuito alla variabile '**VERNICIATURA**' il nome di un colore (scritto con lettere maiuscole); poi, per verificare il tipo di colore si può usare uno di questi nomi di condizione, per esprimere il fatto che la variabile contenga uno dei nomi del gruppo a cui quel nome fa riferimento.

```
000000 01  VERNICIATURA     PIC X(30).
000000     88  VERNICE-CHIARA   "ARANCIO", "GIALLO", "VERDE",
000000                          "AZZURRO",  "GRIGIO",  "BIANCO".
000000     88  VERNICE-SCURA    "NERO", "MARRONE", "ROSSO",
000000                          "BLU",  "VIOLA".
```

L'esempio seguente descrive la variabile '**CODICE**' che può contenere una sola cifra numerica. A questa variabile si associano dei nomi

di condizione, che descrivono raggruppamenti diversi delle cifre che possono essere contenute nella variabile.

```
000000     02  CODICE           PIC 9.
000000         88  PARI         0, 2, 4, 6, 8.
000000         88  DISPARI      1, 3, 5, 7, 9.
000000         88  BASSO        0 THRU 4.
000000         88  ALTO         5 THRU 9.
```

Nella divisione '**PROCEDURE DIVISION**' potrebbero apparire righe come quelle successive, per verificare che la variabile '**CODICE**' contenga certi valori:

```
000000     IF PARI
000000     THEN
000000         PERFORM …;
000000     ELSE
000000         …
```

In questo modo si evita di scrivere un'espressione condizionale complessa come nell'esempio seguente:

```
000000     IF CODICE IS EQUAL TO 0
000000        OR CODICE IS EQUAL TO 2
000000        OR CODICE IS EQUAL TO 4
000000        OR CODICE IS EQUAL TO 6
000000        OR CODICE IS EQUAL TO 8
000000     THEN
000000         PERFORM …
000000     ELSE
000000         …
```

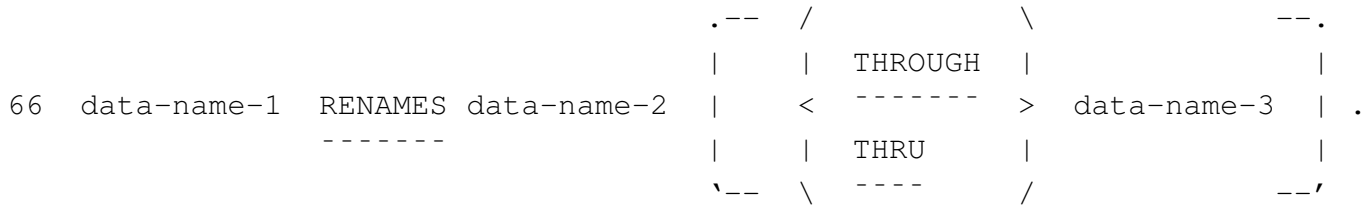
I nomi di condizione si possono associare a variabili che hanno un contenuto alfabetico, alfanumerico e numerico, ma nell'ultimo caso, deve trattarsi di valori rappresentati in forma di stringhe di carat-

teri (pertanto sono escluse le rappresentazioni compatte che usano quattro bit per cifra e quelle binarie).

Un nome di condizione può essere associato a una variabile che costituisce una tabella, oppure che è contenuta in una struttura tabellare. Per fare riferimento al nome di condizione nella cella giusta, occorre utilizzare gli indici, così come si fa normalmente in queste situazioni.

72.8.2 Raggruppamento

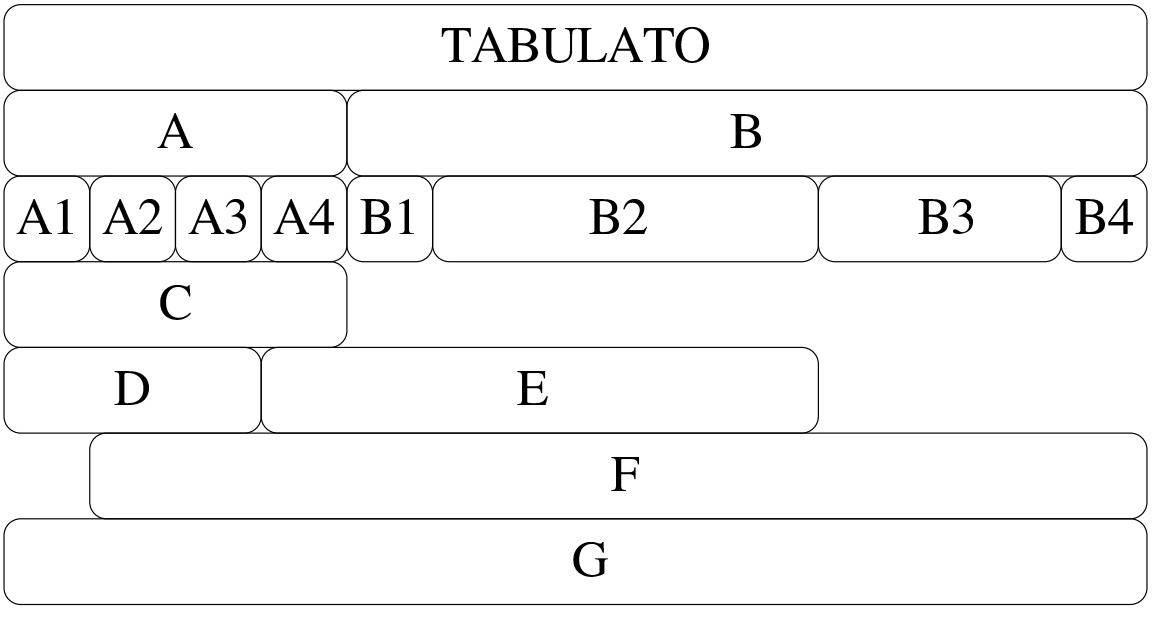
Il livello 66 permette di raggruppare alcune variabili adiacenti di una stessa struttura:



Dopo il numero di livello 66 viene dichiarato il nome del raggruppamento, quindi, dopo la parola chiave '**RENAMES**', si indica il nome della prima variabile; eventualmente, dopo la parola chiave '**THROUGH**' o '**THRU**', si indica l'ultima variabile da riunire. Si osservi l'esempio seguente:

```
000000 01  TABULATO.
000000     05  A.
000000         10  A1  PIC X.
000000         10  A2  PIC XXX.
000000         10  A3  PIC XX.
000000         10  A4  PIC XX.
000000     05  B.
000000         10  B1  PIC XX.
000000         10  B2  PIC X(16).
000000         10  B3  PIC X(8).
000000         10  B4  PIC X.
000000 66  C RENAMES A.
000000 66  D RENAMES A1 THRU A3.
000000 66  E RENAMES A4 THRU B2.
000000 66  F RENAMES A2 THRU B.
000000 66  G RENAMES A  THRU B.
```

Il disegno successivo, anche se non proporzionato rispetto alla dimensione delle variabili, mostra a cosa si riferiscono i vari raggruppamenti:



72.8.3 Qualificazione

I nomi usati nel linguaggio COBOL, per le variabili e per la delimitazione di porzioni del contenuto della divisione '**PROCEDURE DIVISION**' hanno valore per tutta l'estensione del programma; pertanto, in generale, non possono esserci nomi doppi e non si creano ambiti di funzionamento ristretti.

Dal momento che i nomi usati per dichiarare le variabili, oltre che le sezioni e i paragrafi della divisione '**PROCEDURE DIVISION**', hanno una lunghezza limitata normalmente a soli 30 caratteri, in un programma abbastanza grande si può porre il problema di riuscire a scrivere nomi abbastanza mnemonici e univoci. Per questo, è possibile riutilizzare certi nomi, purché poi questi siano indicati attraverso la qualificazione del loro contesto. Il contesto che consente di qualificare un nome è costituito da una gerarchia di nomi; si osservi l'esempio seguente:

```
001700 FILE SECTION.
001800 FD  SALES-FILE
001830     LABEL RECORD IS STANDARD
001860     VALUE OF FILE-ID IS "sales".
001900 01  SALES-RECORD.
002000     05  VENDOR-NAME      PIC X(20).
002100     05  VALUE            PIC S9(6).
002200     05  NUMBER           PIC X(13).
002300     05  TYPE             PIC X.
002400     05  VENDOR-REGION    PIC X(17).
002500     05  VENDOR-CITY      PIC X(20).
002600     05  COMMENTS         PIC X(60).
```

Si vede la dichiarazione del file '**SALES-FILE**', dove la variabile '**SALES-RECORD**' ne rappresenta il record, che a sua volta è suddi-

viso in una serie di campi. La variabile '**TYPE**', di questo esempio, appartiene gerarchicamente alla variabile '**SALES-RECORD**', che a sua volta appartiene al file '**SALES-FILE**'.

Supponendo che in questo programma, per qualche ragione, ci sia un'altra variabile con il nome '**TYPE**', si potrebbe individuare quella abbinata all'esempio specificando che si tratta della variabile '**TYPE**' di '**SALES-RECORD**'; ma volendo supporre che ci siano anche diverse variabili '**SALES-RECORD**', contenenti un campo denominato '**TYPE**', occorrerebbe indicare la variabile '**TYPE**' di '**SALES-RECORD**' di '**SALES-FILE**'.

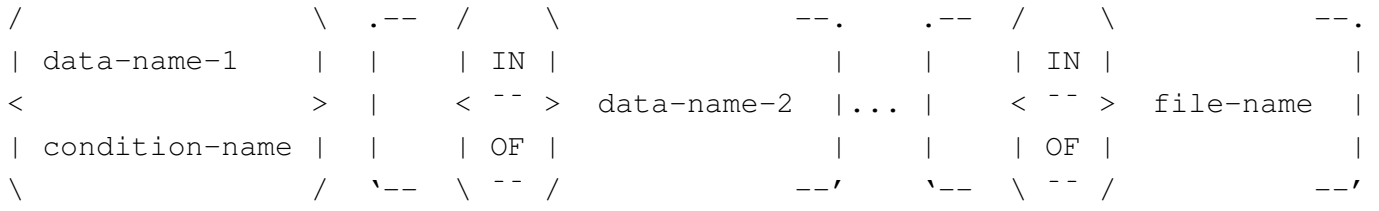
Nella divisione '**PROCEDURE DIVISION**' può succedere qualcosa di simile, quando si usa una suddivisione delle procedure in sezioni e paragrafi. In questo modo, i nomi dei paragrafi potrebbero richiedere una qualificazione, specificando a quale sezione appartengono.

Ciò che consente di qualificare un nome, in modo sufficiente a renderlo univoco, è il ***qualificatore***.

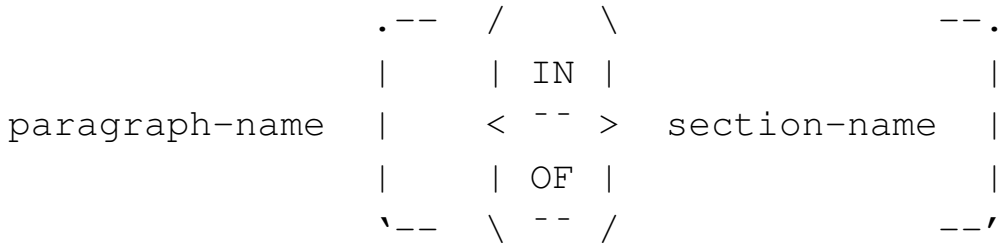
Quando nella divisione '**PROCEDURE DIVISION**' si usa un nome che richiede una qualificazione, si usa indifferentemente la parola chiave '**IN**' oppure '**OF**':

```
… TYPE IN SALES-RECORDS …
… TYPE IN SALES-RECORDS OF SALES-FILE …
```

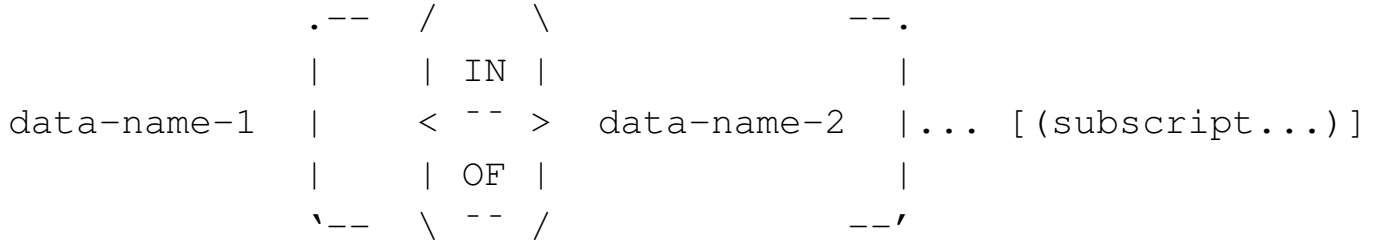
Segue lo schema sintattico per qualificare una variabile:

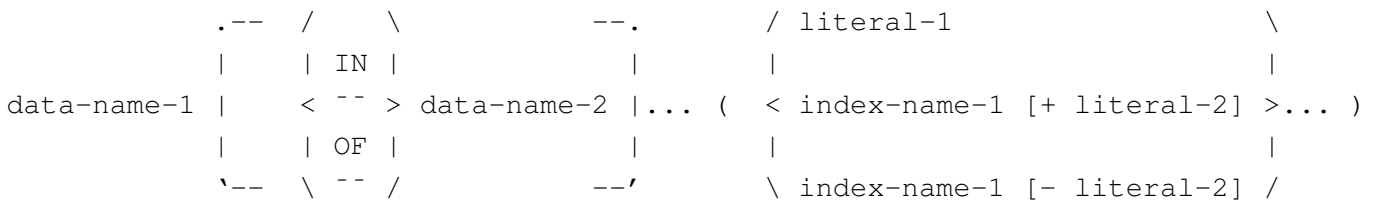


Segue lo schema sintattico per qualificare un paragrafo all'interno della divisione '**PROCEDURE DIVISION**':



Quando si deve usare la qualificazione per individuare un elemento di una tabella, gli indici tra parentesi tonde appaiono alla fine, dopo la qualificazione stessa. Seguono due modelli sintattici alternativi che descrivono il modo di rappresentare un elemento di una tabella, tenendo conto anche della qualificazione:





Le regole che stabiliscono la possibilità o meno di usare la qualificazione, sono intuitive e non vengono descritte qui nel dettaglio. La regola generale da seguire è quella della leggibilità del programma

sorgente, che richiede di usare la qualificazione solo quando questa è utile per migliorare la chiarezza dello stesso.

72.9 Modello di definizione della variabile

«

Il COBOL gestisce soltanto due tipi di dati: numerici e alfanumerici. Tuttavia, nel momento in cui si dichiara il modello, le variabili scalari (ovvero quelle che non sono scomposte) si distinguono in classi e in categorie, secondo lo schema che si vede nella tabella successiva.

Tabella 72.109. Classi e categorie delle variabili COBOL.

Livello di suddivisione	**Classe**	**Categoria**
variabile scalare (non suddivisibile)	numerica	numerica
	alfanumerica	numerica *edited*
		alfanumerica *edited*
		alfanumerica
variabile strutturata (*record*)	alfanumerica	alfanumerica

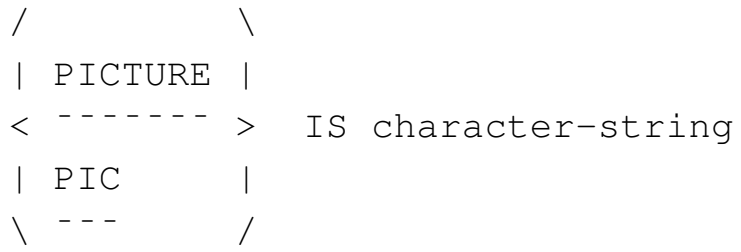
Le variabili che appartengono alla ***classe numerica*** contengono dei valori che si possono utilizzare per eseguire delle espressioni numeriche; mentre quelle che appartengono alla ***classe alfanumerica*** si possono utilizzare soltanto come sequenze di caratteri.

Si osservi che una variabile strutturata è sempre solo alfanumerica, ma ciò non toglie che i campi che contiene possano essere di una classe e di una categoria differente.

Le variabili che appartengono alle categorie *edited*, «modificate», servono per ricevere un valore, numerico o alfanumerico, da modificare nell'atto stesso della ricezione. Una variabile di questo tipo fa sempre parte della classe alfanumerica, perché, una volta ricevuti i dati e modificati in base al modello di definizione della variabile, questi sono semplicemente una sequenza di caratteri senza più alcun valore numerico.

72.9.1 Dichiarazione del modello di definizione della variabile

Il modello di definizione della variabile è introdotto dalla parola chiave '**PICTURE**', o '**PIC**':



La metavariabile ***character-string*** costituisce il modello vero e proprio, che può essere composto da un massimo di 30 caratteri, anche se in questo modo si possono comunque rappresentare modelli di variabili di dimensioni molto più grandi.

Si osservi che l'indicazione di un modello di definizione della variabile è obbligatorio per tutte le variabili scalari (pertanto riguarda potenzialmente i livelli da 01 a 49 e 77), mentre non è consentito per le variabili strutturate.

La stringa che costituisce il modello di definizione della variabile è composta di simboli che descrivono ognuno le caratteristiche di

un carattere, di una cifra, di un segno, oppure inseriscono qualcosa che nel dato originale è assente, ma spesso possono essere ripetuti mettendo un numero intero tra parentesi:

```
x(n)
```

Una notazione del genere indica che il simbolo ***x*** va inteso come se fosse ripetuto ***n*** volte; per esempio, '`9(3)V9(5)`' è equivalente a '`999V99999`'. È attraverso questo meccanismo che si possono descrivere variabili molto grandi, pur avendo un modello di definizione limitato a soli 30 caratteri.

Nelle sezioni successive vengono descritti i simboli utilizzabili per i modelli di definizione delle variabili scalari. Lo standard del linguaggio COBOL annovera più possibilità di quelle indicate, ma i compilatori non si comportano sempre allo stesso modo e spesso ignorano l'uso di alcuni simboli, oppure li interpretano in modo inesatto. Questo problema riguarda soprattutto i simboli che applicano delle modifiche al valore ricevuto nella variabile, che quindi vanno usati con estrema parsimonia se si vuole scrivere un programma abbastanza compatibile.

72.9.2 Variabili alfanumeriche

«

Sono variabili alfanumeriche pure e semplici quelle che possono rappresentare qualunque carattere delle codifica prevista, limitatamente al fatto che la rappresentazione è in forma di byte (ASCII o EBCDIC). Il modello di definizione di una variabile alfanumerica prevede simboli speciali che limitano l'inserimento di soli caratteri alfa-

betici o di sole cifre numeriche, ma in pratica, succede spesso che i compilatori non applichino alcuna restrizione.

Tabella 72.111. Simboli del modello di definizione di una variabile alfanumerica.

Simbolo	Descrizione
`9`	Rappresenta una cifra numerica.
`A`	Rappresenta una lettera dell'alfabeto latino, non accentata, maiuscola o minuscola.
`X`	Rappresenta un carattere qualsiasi.

Si osservi che il modello di definizione di una variabile alfanumerica deve contenere almeno un simbolo '**`X`**' o '**`A`**', altrimenti, un modello che contenga soltanto il simbolo '**`9`**' verrebbe interpretato come numerico.

A titolo di esempio, viene mostrato un piccolo programma con tre variabili scalari alfanumeriche, aventi modelli diversi, abbinate ognuna a una variabile strutturata. Alle variabili scalari viene assegnato lo stesso valore, in modo da poter confrontare come questo valore viene inteso e rappresentato.

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   PICTURE-ALPHANUMERIC.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-02-23.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 DATA DIVISION.
000900*
001000 WORKING-STORAGE SECTION.
001100 01  RECORD-A.
```

```
001200     02  A        PICTURE X(15)        USAGE IS DISPLAY.
001300 01  RECORD-B.
001400     02  B        PICTURE 9(5)A(5)X(5) USAGE IS DISPLAY.
001500 01  RECORD-C.
001600     02  C        PICTURE A(5)X(5)9(5) USAGE IS DISPLAY.
001700*
001800 PROCEDURE DIVISION.
001900*
002000 MAIN.
002100     MOVE " 1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ" TO A.
002200     MOVE " 1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ" TO B.
002300     MOVE " 1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ" TO C.
002400     DISPLAY "SOURCE VALUE IS: ", QUOTE,
002500             " 1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ", QUOTE.
002600     DISPLAY "PICTURE: X(15)         VALUE: ", QUOTE, A, QUOTE,
002700             "  DATA: ", QUOTE, RECORD-A, QUOTE.
002800     DISPLAY "PICTURE: 9(5)A(5)X(5)  VALUE: ", QUOTE, B, QUOTE,
002900             "  DATA: ", QUOTE, RECORD-B, QUOTE.
003000     DISPLAY "PICTURE: A(5)X(5)9(5)  VALUE: ", QUOTE, C, QUOTE,
003100             "  DATA: ", QUOTE, RECORD-C, QUOTE.
003200*
```

Compilando il programma con TinyCOBOL, l'avvio dell'eseguibile che si ottiene genera il risultato seguente, dove si può vedere che l'uso dei simboli '**A**' e '**9**' non comporta alcuna differenza di funzionamento rispetto a '**X**'; tuttavia, un compilatore più sofisticato potrebbe segnalare qualche tipo di errore:

```
SOURCE VALUE IS: " 1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ"
PICTURE: X(15)         VALUE: " 1234567890ABCD"  DATA: " 1234567890ABCD"
PICTURE: 9(5)A(5)X(5)  VALUE: " 1234567890ABCD"  DATA: " 1234567890ABCD"
PICTURE: A(5)X(5)9(5)  VALUE: " 1234567890ABCD"  DATA: " 1234567890ABCD"
```

72.9.3 Variabili alfanumeriche modificate

Il modello di definizione di una variabile alfanumerica può contenere simboli che applicano una modifica al valore stesso. La modifica avviene nel momento in cui il valore viene ricevuto.

Tabella 72.114. Simboli del modello di definizione di una variabile alfanumerica che descrivono una modifica del valore ricevuto.

Simbolo	Descrizione
`B`	Richiede l'inserimento di uno spazio.
`0`	Richiede l'inserimento di uno zero.
`/`	Se il compilatore lo permette, inserisce una barra obliqua.
`,`	Se il compilatore lo permette, inserisce una virgola. Si osservi comunque che non può apparire la virgola come simbolo conclusivo del modello, perché in tal caso assumerebbe il significato di un delimitatore.
	Teoricamente, ogni altro simbolo che non abbia un significato preciso per la realizzazione dei modelli di definizione delle variabili, dovrebbe essere aggiunto tale e quale; in pratica, molto dipende dalle caratteristiche del compilatore.

A titolo di esempio, viene mostrato un piccolo programma con due variabili scalari alfanumeriche, aventi modelli diversi, abbinate ognuna a una variabile strutturata. Alle variabili scalari viene assegnato lo stesso valore, in modo da poter confrontare come questo viene inteso e rappresentato. Nell'esempio si tenta in particolare di inserire in un modello una barra obliqua e una virgola.

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   PICTURE-ALPHANUMERIC-EDITED.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-02-23.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 DATA DIVISION.
000900*
001000 WORKING-STORAGE SECTION.
001100 01  RECORD-A.
001200     02  A        PICTURE X(15)           USAGE IS DISPLAY.
001300 01  RECORD-B.
001400     02  B        PICTURE ABX09,/X(8) USAGE IS DISPLAY.
001500*
001600 PROCEDURE DIVISION.
001700*
001800 MAIN.
001900     MOVE "ABCDEFGHIJKLMNOPQRSTUVWXYZ" TO A.
002000     MOVE "ABCDEFGHIJKLMNOPQRSTUVWXYZ" TO B.
002100     DISPLAY "SOURCE VALUE IS: ", QUOTE,
002200             "ABCDEFGHIJKLMNOPQRSTUVWXYZ", QUOTE.
002300     DISPLAY "PICTURE: X(15)        VALUE: ", QUOTE, A, QUOTE,
002400             "  DATA: ", QUOTE, RECORD-A, QUOTE.
002500     DISPLAY "PICTURE: ABX09,/X(8)  VALUE: ", QUOTE, B, QUOTE,
002600             "  DATA: ", QUOTE, RECORD-B, QUOTE.
002700*
```

Compilando il programma con TinyCOBOL, l'avvio dell'eseguibile che si ottiene genera il risultato seguente:

```
SOURCE VALUE IS: "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
PICTURE: X(15)        VALUE: "ABCDEFGHIJKLMNO"  DATA: "ABCDEFGHIJKLMNO"
PICTURE: ABX09,/X(8)  VALUE: "A B0C,/DEFGHIJK"  DATA: "A B0C,/DEFGHIJK"
```

72.9.4 Variabili numeriche

Le variabili numeriche pure e semplici, sono quelle il cui valore può essere usato per calcolare delle espressioni numeriche. Nel modello di definizione di una variabile di questo tipo, possono apparire solo simboli che descrivono la dimensione del valore rappresentabile, distinguendo la parte intera da quella decimale e specificando eventualmente la presenza del segno.

Tabella 72.117. Simboli del modello di definizione di una variabile numerica.

Simbolo	Descrizione
`9`	Rappresenta una cifra numerica singola.
`V`	Rappresenta la separazione tra la parte intera e la parte decimale del numero e di conseguenza può apparire una sola volta nel modello. Questo simbolo non incrementa la dimensione della variabile, in quanto serve solo a sapere dove si intende che debba trovarsi la separazione tra la parte intera e quella decimale, ai fini dei calcoli che si possono eseguire. Se questo simbolo appare alla fine del modello, è come se non fosse inserito, perché la sua presenza non aggiunge alcuna informazione.
`S`	Se si utilizza questo simbolo, può essere collocato soltanto all'estrema sinistra del modello, allo scopo di specificare che il numero è provvisto di segno. A seconda del modo in cui l'informazione numerica viene codificata nella variabile (‘**`DISPLAY`**’, ‘**`COMPUTATIONAL`**’ e altre opzioni eventuali), il segno può occupare una posizione separata oppure no.

Simbolo	Descrizione
P	Il simbolo '**P**' può essere usato soltanto alla sinistra o alla destra del modello e può apparire ripetuto, tenendo conto che se appare alla sinistra è incompatibile con il simbolo '**V**'. Questo simbolo non incrementa la dimensione della variabile, ma serve a far sì che il valore contenuto nella variabile sia interpretato in modo differente. **L'uso di questo simbolo è sconsigliabile**, perché altera il comportamento della variabile in un modo che non è facile da interpretare correttamente.

A titolo di esempio, viene mostrato un piccolo programma con cinque variabili scalari numeriche, aventi modelli diversi, abbinate ognuna a una variabile strutturata. Alle variabili scalari viene assegnato lo stesso valore, in modo da poter confrontare come questo valore viene inteso e rappresentato. Nell'esempio appare anche l'uso del simbolo '**P**' a dimostrazione della difficoltà che questo comporta nell'interpretare l'esito degli assegnamenti alle variabili.

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   PICTURE-NUMERIC.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-02-22.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 DATA DIVISION.
000900*
001000 WORKING-STORAGE SECTION.
001100 01  RECORD-A.
001200     02  A        PICTURE 99999       USAGE IS DISPLAY.
001300 01  RECORD-B.
001400     02  B        PICTURE 999V99      USAGE IS DISPLAY.
```

```
001500 01  RECORD-C.
001600     02  C        PICTURE S999V99     USAGE IS DISPLAY.
001700 01  RECORD-D.
001800     02  D        PICTURE 999V99PP    USAGE IS DISPLAY.
001900 01  RECORD-E.
002000     02  E        PICTURE PP99999     USAGE IS DISPLAY.
002100*
002200 PROCEDURE DIVISION.
002300*
002400 MAIN.
002500     MOVE -1234.5678 TO A.
002600     MOVE -1234.5678 TO B.
002700     MOVE -1234.5678 TO C.
002800     MOVE -1234.5678 TO D.
002900     MOVE -1234.5678 TO E.
003000     DISPLAY "SOURCE VALUE IS -1234.5678".
003100     DISPLAY "PICTURE: 99999     VALUE: ", A,
003200             "     DATA: ", RECORD-A.
003300     DISPLAY "PICTURE: 999V99    VALUE: ", B,
003400             "    DATA: ", RECORD-B.
003500     DISPLAY "PICTURE: S999V99   VALUE: ", C,
003600             "   DATA: ", RECORD-C.
003700     DISPLAY "PICTURE: 999V99PP  VALUE: ", D,
003800             "  DATA: ", RECORD-D.
003900     DISPLAY "PICTURE: PP99999   VALUE: ", E,
004000             "  DATA: ", RECORD-E.
004100     STOP RUN.
004200*
```

Compilando il programma con TinyCOBOL, l'avvio dell'eseguibile che si ottiene genera il risultato seguente:

```
SOURCE VALUE IS -1234.5678
PICTURE: 99999     VALUE: 01234     DATA: 01234
PICTURE: 999V99    VALUE: 234.56    DATA: 23456
PICTURE: S999V99   VALUE: -234.56   DATA: 2345O
PICTURE: 999V99PP  VALUE: 004.5678  DATA: 45678
PICTURE: PP99999   VALUE: .0078000  DATA: 78000
```

Facendo la stessa cosa con OpenCOBOL:

```
SOURCE VALUE IS -1234.5678
PICTURE: 99999     VALUE: 01234     DATA: 01234
PICTURE: 999V99    VALUE: 234.56    DATA: 23456
PICTURE: S999V99   VALUE: -234.56   DATA: 2345v
PICTURE: 999V99PP  VALUE: 004.5678  DATA: 45678
PICTURE: PP99999   VALUE: .0078000  DATA: 78000
```

Si osservi che nell'esempio le variabili scalari numeriche sono state dichiarate con l'opzione '**`USAGE IS DISPLAY`**', che comunque sarebbe stata implicita, in modo da assicurare la visibilità del contenuto leggendo il livello 01.

72.9.5 Variabili numeriche modificate

Il modello di definizione di una variabile fatta per ricevere un valore numerico, può contenere simboli che applicano una modifica all'apparenza del valore. Nel momento in cui una variabile del genere riceve un valore, ciò che la variabile fornisce è un'informazione alfanumerica, che non può più essere usata per elaborazioni matematiche; al massimo, una variabile del genere può ricevere il risultato di un'espressione che generi un valore numerico.

Tabella 72.121. Alcuni simboli del modello di definizione di una variabile numerica che descrivono una trasformazione in formato alfanumerico.

Simbolo	Descrizione
B	Richiede l'inserimento di uno spazio.
0	Richiede l'inserimento di uno zero.
Z	Si usa soltanto nella parte sinistra del modello e richiede di indicare uno spazio al posto di una cifra zero.
*	Si usa soltanto nella parte sinistra del modello e richiede di indicare un asterisco al posto di una cifra zero.
,	In condizioni normali inserisce una virgola letterale. Si osservi che non può apparire la virgola come simbolo conclusivo del modello, perché in tal caso assumerebbe il valore di un delimitatore.
.	In condizioni normali rappresenta il punto di separazione tra la parte intera e la parte decimale. Si osservi che in un modello che prevede la modifica, non si può usare il simbolo '**V**'.
+ –	Il segno '**+**' o il segno '**–**' indicano la posizione in cui si vuole appaia il segno del numero; se il segno viene ripetuto, le occorrenze successive alla prima rappresentano implicitamente la posizione di una cifra numerica, con soppressione degli zeri anteriori. Usando il segno '**–**' si dovrebbe ottenere la rappresentazione di un numero senza segno, quando questo ha un valore positivo.

Simbolo	Descrizione
`CR` `DB`	Questi simboli, che occupano due caratteri, vengono inseriti tali e quali nella rappresentazione della variabile. Sono utili queste sigle per la contabilità dei paesi di lingua inglese (*credit*, *debit*); a ogni modo è bene ricordare che non sono influenzate dal segno del valore numerico che viene ricevuto nella variabile.
`$`	Si usa per mostrare il simbolo del dollaro, quando la valuta prevista nel sorgente del programma è quella predefinita.
`/`	Se il compilatore lo permette, inserisce una barra obliqua.
	Teoricamente, ogni altro simbolo che non abbia un significato preciso per la realizzazione dei modelli di definizione delle variabili, dovrebbe essere aggiunto tale e quale; in pratica, molto dipende dalle caratteristiche del compilatore.

Come esempio viene mostrato un piccolo programma con alcune variabili scalari numeriche modificate (*edited*), aventi modelli diversi, abbinate ognuna a una variabile strutturata. Alle variabili scalari viene assegnato lo stesso valore, in modo da poter confrontare come questo valore viene inteso e rappresentato.

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   PICTURE-NUMERIC-EDITED.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-02-25.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 DATA DIVISION.
000900*
```

```
001000 WORKING-STORAGE SECTION.
001100 01  RECORD-A.
001200     02  A        PICTURE S9(10)V9(5)   USAGE IS DISPLAY.
001300 01  RECORD-B.
001400     02  B        PICTURE +Z(9)9.9(5)   USAGE IS DISPLAY.
001500 01  RECORD-C.
001600     02  C        PICTURE CR+Z(7)9.9(5) USAGE IS DISPLAY.
001700 01  RECORD-D.
001800     02  D        PICTURE +Z(7)9.9(5)DB USAGE IS DISPLAY.
001900 01  RECORD-E.
002000     02  E        PICTURE *(9)9.9(5)+   USAGE IS DISPLAY.
002100 01  RECORD-F.
002200     02  F        PICTURE +*(9)9.9(4)$  USAGE IS DISPLAY.
002300 01  RECORD-G.
002400     02  G        PICTURE +*(9)9,9(4)$  USAGE IS DISPLAY.
002500 01  RECORD-H.
002600     02  H        PICTURE -(10)9,9(4)$  USAGE IS DISPLAY.
002700 01  RECORD-I.
002800     02  I        PICTURE +(10)9,9(4)$  USAGE IS DISPLAY.
002900*
003000 PROCEDURE DIVISION.
003100*
003200 MAIN.
003300     MOVE +123456.789 TO A.
003400     MOVE +123456.789 TO B.
003500     MOVE +123456.789 TO C.
003600     MOVE +123456.789 TO D.
003700     MOVE +123456.789 TO E.
003800     MOVE +123456.789 TO F.
003900     MOVE +123456.789 TO G.
004000     MOVE +123456.789 TO H.
004100     MOVE +123456.789 TO I.
004200     DISPLAY "SOURCE VALUE IS: +123456.789".
004300     DISPLAY "PICTURE: S9(10)V9(5)    VALUE: ", A,
004400             "  DATA: ", RECORD-A.
004500     DISPLAY "PICTURE: +Z(9)9.9(5)    VALUE: ", B,
```

```
004600              "  DATA: ", RECORD-B.
004700     DISPLAY "PICTURE: CR+Z(7)9.9(5)  VALUE: ", C,
004800              "  DATA: ", RECORD-C.
004900     DISPLAY "PICTURE: +Z(7)9.9(5)DB  VALUE: ", D,
005000              "  DATA: ", RECORD-D.
005100     DISPLAY "PICTURE: *(9)9.9(5)+    VALUE: ", E,
005200              "  DATA: ", RECORD-E.
005300     DISPLAY "PICTURE: +*(9)9.9(4)$   VALUE: ", F,
005400              "  DATA: ", RECORD-F.
005500     DISPLAY "PICTURE: +*(9)9,9(4)$   VALUE: ", G,
005600              "  DATA: ", RECORD-G.
005700     DISPLAY "PICTURE: -(10)9,9(4)$   VALUE: ", H,
005800              "  DATA: ", RECORD-H.
005900     DISPLAY "PICTURE: +(10)9,9(4)$   VALUE: ", I,
006000              "  DATA: ", RECORD-I.
006100     STOP RUN.
006200*
```

Compilando il programma con TinyCOBOL, l'avvio dell'eseguibile che si ottiene genera il risultato seguente:

```
SOURCE VALUE IS: +123456.789
PICTURE: S9(10)V9(5)    VALUE:  0000123456.78900  DATA: 00001234567890{
PICTURE: +Z(9)9.9(5)    VALUE: +    123456.78900  DATA: +    123456.78900
PICTURE: CR+Z(7)9.9(5)  VALUE: CR+  123456.78900  DATA: CR+  123456.78900
PICTURE: +Z(7)9.9(5)DB  VALUE: +  123456.78900DB  DATA: +  123456.78900DB
PICTURE: *(9)9.9(5)+    VALUE: ****123456.78900+  DATA: ****123456.78900+
PICTURE: +*(9)9.9(4)$   VALUE: +****123456.7890$  DATA: +****123456.7890$
PICTURE: +*(9)9,9(4)$   VALUE: +********12,3456$  DATA: +********12,3456$
PICTURE: -(10)9,9(4)$   VALUE:          12,3456$  DATA:          12,3456$
PICTURE: +(10)9,9(4)$   VALUE:         +12,3456$  DATA:         +12,3456$
```

Tra i vari risultati, si può osservare che la virgola è stata interpretata come un segno senza un ruolo preciso, pertanto si colloca semplicemente prima delle ultime quattro cifre, secondo la previsione del modello.

Intervenendo nella sezione '**CONFIGURATION SECTION**' è possibile invertire il ruolo del punto e della virgola, nella rappresentazione dei numeri; nello stesso modo, è possibile attribuire un simbolo differente per la valuta. L'esempio seguente è una variante di quello appena mostrato, con le modifiche necessarie per questo scopo. Si osservi che come simbolo di valuta è stata scelta la lettera «E».

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.    PICTURE-NUMERIC-EDITED-BIS.
000300 AUTHOR.        DANIELE GIACOMINI.
000400 DATE-WRITTEN.  2005-02-25.
000500*
000600 ENVIRONMENT DIVISION.
000700 CONFIGURATION SECTION.
000800 SPECIAL-NAMES. DECIMAL-POINT IS COMMA
000900                CURRENCY SIGN IS "E".
001000*
001100 DATA DIVISION.
001200*
001300 WORKING-STORAGE SECTION.
001400 01  RECORD-A.
001500     02  A        PICTURE S9(10)V9(5)   USAGE IS DISPLAY.
001600 01  RECORD-B.
001700     02  B        PICTURE +Z(9)9.9(5)   USAGE IS DISPLAY.
001800 01  RECORD-C.
001900     02  C        PICTURE CR+Z(7)9.9(5) USAGE IS DISPLAY.
002000 01  RECORD-D.
002100     02  D        PICTURE +Z(7)9.9(5)DB USAGE IS DISPLAY.
002200 01  RECORD-E.
002300     02  E        PICTURE *(9)9.9(5)+   USAGE IS DISPLAY.
002400 01  RECORD-F.
002500     02  F        PICTURE +*(9)9.9(4)E  USAGE IS DISPLAY.
002600 01  RECORD-G.
002700     02  G        PICTURE +*(9)9,9(4)E  USAGE IS DISPLAY.
002800 01  RECORD-H.
```

```
002900     02  H         PICTURE -(10)9,9(4)E  USAGE IS DISPLAY.
003000 01  RECORD-I.
003100     02  I         PICTURE +(10)9,9(4)E  USAGE IS DISPLAY.
003200*
003300 PROCEDURE DIVISION.
003400*
003500 MAIN.
003600     MOVE +123456,789 TO A.
003700     MOVE +123456,789 TO B.
003800     MOVE +123456,789 TO C.
003900     MOVE +123456,789 TO D.
004000     MOVE +123456,789 TO E.
004100     MOVE +123456,789 TO F.
004200     MOVE +123456,789 TO G.
004300     MOVE +123456,789 TO H.
004400     MOVE +123456,789 TO I.
004500     DISPLAY "SOURCE VALUE IS: +123456.789".
004600     DISPLAY "PICTURE: S9(10)V9(5)    VALUE: ", A,
004700             "  DATA: ", RECORD-A.
004800     DISPLAY "PICTURE: +Z(9)9.9(5)    VALUE: ", B,
004900             "  DATA: ", RECORD-B.
005000     DISPLAY "PICTURE: CR+Z(7)9.9(5)  VALUE: ", C,
005100             "  DATA: ", RECORD-C.
005200     DISPLAY "PICTURE: +Z(7)9.9(5)DB  VALUE: ", D,
005300             "  DATA: ", RECORD-D.
005400     DISPLAY "PICTURE: *(9)9.9(5)+    VALUE: ", E,
005500             "  DATA: ", RECORD-E.
005600     DISPLAY "PICTURE: +*(9)9.9(4)E   VALUE: ", F,
005700             "  DATA: ", RECORD-F.
005800     DISPLAY "PICTURE: +*(9)9,9(4)E   VALUE: ", G,
005900             "  DATA: ", RECORD-G.
006000     DISPLAY "PICTURE: -(10)9,9(4)E   VALUE: ", H,
006100             "  DATA: ", RECORD-H.
006200     DISPLAY "PICTURE: +(10)9,9(4)E   VALUE: ", I,
006300             "  DATA: ", RECORD-I.
006400     STOP RUN.
```

```
006500*
```

Compilando il programma con TinyCOBOL, l'avvio dell'eseguibile che si ottiene genera il risultato seguente:

```
SOURCE VALUE IS: +123456.789
PICTURE: S9(10)V9(5)    VALUE:  0000123456,78900  DATA: 00001234567890{
PICTURE: +Z(9)9.9(5)    VALUE: +         1.23456  DATA: +         1.23456
PICTURE: CR+Z(7)9.9(5)  VALUE: CR+       1.23456  DATA: CR+       1.23456
PICTURE: +Z(7)9.9(5)DB  VALUE: +       1.23456DB  DATA: +       1.23456DB
PICTURE: *(9)9.9(5)+    VALUE: *********1.23456+  DATA: *********1.23456+
PICTURE: +*(9)9.9(4)E   VALUE: +********12.3456E  DATA: +********12.3456E
PICTURE: +*(9)9,9(4)E   VALUE: +****123456,7890E  DATA: +****123456,7890E
PICTURE: -(10)9,9(4)E   VALUE:      123456,7890E  DATA:      123456,7890E
PICTURE: +(10)9,9(4)E   VALUE:     +123456,7890E  DATA:     +123456,7890E
```

Questa volta si può osservare che nel modello è il punto che perde il suo significato, apparendo nel risultato soltanto nella posizione prevista, allineando le cifre numeriche originali alla destra.

72.10 Note sull'utilizzo dell'insieme di caratteri universale con il COBOL

Lo standard COBOL del 1985 prevede sostanzialmente che si possano gestire informazioni alfanumeriche composte di simboli rappresentabili in byte, appartenenti pertanto al codice ASCII o EBCDIC. Con l'introduzione dell'insieme di caratteri universale, si pone il problema di gestire codifiche di tipo diverso, ben più grandi del solito byte. Purtroppo, però, le soluzioni adottate non sono semplici e lineari come nel passato.

72.10.1 Stringhe letterali

«

Le stringhe letterali che devono contenere simboli al di fuori del codice ASCII o EBCDIC, devono essere delimitate in modo differente. Generalmente sono disponibili due forme:

```
N"stringa_letterale"
```

```
NX"stringa_esadecimale"
```

La prima forma riguarda una stringa letterale composta secondo la forma codificata del carattere prevista dal compilatore (UTF-8, UTF-16 o altro); la seconda, ammesso che sia disponibile, viene espressa attraverso cifre esadecimali. Si osservi, però, che per poter esprimere una stringa del genere in forma esadecimale, occorre sapere in che modo il compilatore la interpreta, dato che dipende dalla forma codificata del carattere adottata.

72.10.2 modello di definizione delle variabili

«

Nel modello di definizione di una variabile, la lettera '**N**' rappresenta un carattere espresso secondo la codifica universale; la lettera «N» sta per *National*. Pertanto, si aggiunge anche una voce nuova all'opzione '**USAGE**': '**USAGE IS NATIONAL**'.

A seconda della forma codificata del carattere adottata dal compilatore, cambia la dimensione di una variabile del genere. Se si utilizzano codifiche del tipo UTF-8, che hanno una lunghezza variabile, può diventare impossibile stabilire in anticipo la dimensione in byte

corrispondente. Anche per questo motivo, è improbabile che si possa usare lo standard UTF-8 con il COBOL.

72.10.3 Costanti figurative

Tra le costanti figurative, '**HIGH-VALUES**' e '**LOW-VALUES**' perdono di significato, se associate a una variabile dichiarata come '**USAGE IS NATIONAL**'.

72.11 Divisione «PROCEDURE DIVISION»

La divisione '**PROCEDURE DIVISION**' costituisce la quarta e ultima parte di un programma sorgente COBOL. La divisione si può suddividere in paragrafi, oppure in sezioni contenenti eventualmente dei paragrafi. All'interno delle sezioni o dei paragrafi, si inseriscono le istruzioni che descrivono la procedura del programma.

Le istruzioni sono inserite a gruppi, terminanti con un punto fermo, seguito da uno spazio; le istruzioni singole, che non costituiscono un gruppo autonomo, possono essere separate graficamente attraverso dei separatori (la virgola, il punto e virgola, la parola '**THEN**').

Alcune istruzioni, quando non costituiscono un gruppo autonomo, possono collocarsi solo alla fine del gruppo. Si tratta precisamente di '**GO TO**' e di '**STOP RUN**'.

La divisione può articolarsi in tre modi diversi; quello che si vede descritto nello schema segue è il più semplice, perché non fa uso delle sezioni:

```
[PROCEDURE DIVISION.
 --------- --------
```

```
{paragraph-name.
     [sentence]...}...
```

Se si usano le sezioni, i paragrafi devono essere contenuti tutti all'interno di sezioni:

```
[PROCEDURE DIVISION.
 --------- --------

/ section-name SECTION [segment-number]. \
|              -------                   |
< {paragraph-name.                       >...
|      [sentence]...}...                 |
\                                        /
```

Eventualmente ci può essere un gruppo iniziale di sezioni speciali; in tal caso, è obbligatorio suddividere il resto del programma in sezioni:

```
[PROCEDURE DIVISION.
 --------- --------

 DECLARATIVES.
 ------------

 / section-name SECTION [segment-number]. \
 |              -------                   |
 |     USE statement                      |
 <     ---                                >...
 | {paragraph-name.                       |
 |      [sentence]...}...                 |
 \                                        /
 END DECLARATIVES.
 --- ------------

/ section-name SECTION [segment-number]. \
|              -------                   |
< {paragraph-name.                       >...
|      [sentence]...}...                 |
\                                        /
```

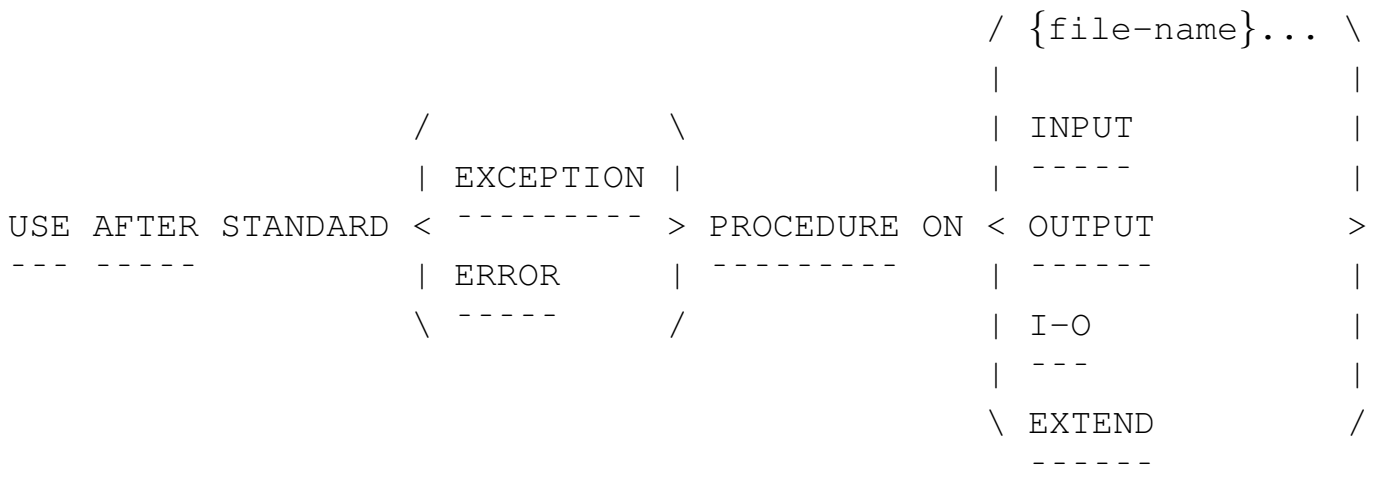
Il primo gruppo di istruzioni a essere eseguito è quello che si trova nel primo paragrafo della prima sezione; escludendo quelli inseriti in un blocco '**DECLARATIVES**'. In condizioni normali, la sequenza dei gruppi di istruzioni eseguiti prosegue con quelli successivi, salvo quando si incontrano istruzioni speciali che richiedono esplicitamente di modificare questo tipo di flusso.

72.11.1 Gruppo di sezioni «DECLARATIVES»

Quando all'inizio della divisione '**PROCEDURE DIVISION**' appare la parola chiave '**DECLARATIVES**', che inizia dall'area A del modulo di programmazione, le sezioni dichiarate fino alla riga dove appare '**END DECLARATIVES**' (sempre a partire dall'area A), non vengono eseguite normalmente, ma solo al verificarsi di certe condizioni.

```
DECLARATIVES.
------------
/ section-name SECTION [segment-number]. \
|                -------                 |
|     USE statement                      |
<     ---                               >...
| {paragraph-name.                       |
|       [sentence]...}...                |
\                                        /
END DECLARATIVES.
--- ------------
```

Ogni sezione di questo gruppo speciale, inizia con una o più istruzioni '**USE**', prima di procedere con dei paragrafi contenenti altre istruzioni. L'istruzione '**USE**' serve ad abbinare l'esecuzione della sezione (a partire dal primo dei suoi paragrafi), a condizione che si verifichi una certa condizione:



Tenendo conto che le parole chiave '**EXCEPTION**' e '**ERROR**' del modello sono equivalenti, si intende che questa istruzione serve ad attivare la sezione che la contiene se si verifica una condizione di errore, che non sia stato gestito diversamente all'interno del programma, riguardante: un certo file (***file-name***), un file qualunque aperto in lettura ('**INPUT**'), scrittura ('**OUTPUT**'), lettura e scrittura ('**I-O**') o in estensione ('**EXTEND**').

Viene mostrato l'esempio di un piccolo programma completo, che ha lo scopo di leggere un file ('`input.txt`') e di mostrarne il contenuto sullo schermo:

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   TEST-DECLARATIVES.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-02-26.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 INPUT-OUTPUT SECTION.
000900*
001000 FILE-CONTROL.
001100*
```

```
001200     SELECT FILE-DA-LEGGERE ASSIGN TO "input.txt"
001300                            ORGANIZATION IS
001350                                LINE SEQUENTIAL
001400                            FILE STATUS IS
001450                                STATO-FILE-DA-LEGGERE.
001500*
001600 DATA DIVISION.
001700*
001800 FILE SECTION.
001900*
002000 FD  FILE-DA-LEGGERE.
002100 01  RECORD-DA-LEGGERE            PIC X(79).
002200*
002300 WORKING-STORAGE SECTION.
002400 77  STATO-FILE-DA-LEGGERE        PIC XX.
002500*
002600 PROCEDURE DIVISION.
002700*
002800 DECLARATIVES.
002900 FILE-ACCESS-ERROR SECTION.
003000     USE AFTER STANDARD ERROR PROCEDURE ON INPUT.
003100 FILE-ACCESS-ERROR-RECOVERY.
003200     DISPLAY "FILE ACCESS ERROR: ",
003250             STATO-FILE-DA-LEGGERE.
003300     STOP RUN.
003400 END DECLARATIVES.
003500*
003550 MAIN-PROCEDURE SECTION.
003600 MAIN.
003700     OPEN INPUT FILE-DA-LEGGERE.
003800     PERFORM LETTURA UNTIL 0 = 1.
003900     CLOSE FILE-DA-LEGGERE.
004000*
004100     STOP RUN.
```

```
004200*
004300 LETTURA.
004400     READ FILE-DA-LEGGERE.
004500     DISPLAY RECORD-DA-LEGGERE.
004600*
```

Si può osservare nel programma che il ciclo di lettura non termina mai, perché la condizione '**0 = 1**' non si può avverare. Così facendo, dato che la lettura non prevede alcun controllo del superamento della fine del file, si verifica un errore che viene preso in considerazione dalla sezione '**FILE-ACCESS-ERROR**'.

Compilando il programma con OpenCOBOL, l'avvio dell'eseguibile che si ottiene genera un risultato simile a quello seguente:

```
aaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbb
cccccccccccccccccc
FILE ACCESS ERROR: 10
```

In pratica, alla fine del file termina la visualizzazione del suo contenuto e si ottiene un messaggio di errore, come organizzato nella sezione '**FILE-ACCESS-ERROR**'.

72.11.2 Sezioni e segmenti

«

Le sezioni della divisione '**PROCEDURE DIVISION**', oltre al nome possono indicare un numero di segmento, che può andare da zero a 99.

```
section-name SECTION [segment-number].
```

Il numero di segmento serve a raggruppare tutte le sezioni con lo stesso numero in uno stesso segmento, allo scopo di sapere, quale parte del programma deve rimanere simultaneamente nella memoria centrale durante il funzionamento.

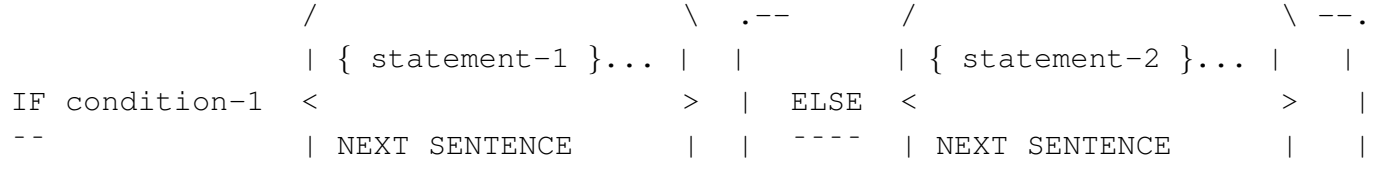
Si dividono precisamente due tipi di segmenti: quelli fissi, con numeri da 00 a 49, e quelli indipendenti, da 50 a 99. I segmenti numerati fino al numero 49 devono rimanere sempre in memoria, mentre gli altri devono esserci solo per il tempo necessario al loro funzionamento. Per questa ragione, le sezioni dichiarate nella zona '**DECLARATIVES**', possono essere associate soltanto a segmenti fissi (da 00 a 49).

Naturalmente, questa possibilità di segmentare il programma dipende dal compilatore, che potrebbe limitarsi semplicemente a ignorare il numero di segmento.

72.11.3 Gruppi di istruzioni e istruzioni condizionali

Un gruppo di istruzioni si evidenzia per la presenza del punto fermo conclusivo (seguito da uno spazio). Le istruzioni che non costituiscono gruppi singoli possono essere separate, oltre che con lo spazio, con la virgola, il punto e virgola, e con la parola '**THEN**'.

Le istruzioni condizionali sono quelle che alterano la sequenza normale dell'esecuzione delle istruzioni, sulla base della verifica di una condizione. L'istruzione condizionale tipica è '**IF**', ma molte altre istruzioni prevedono delle parole opzionali per descrivere un'azione da compiere al verificarsi di una certa condizione.



\ ---- -------- / `-- \ ---- -------- / --'

Quello che si vede sopra è lo schema sintattico dell'istruzione '**IF**', che incorpora a sua volta altre istruzioni. Naturalmente, le istruzioni incorporate possono contenere altre istruzioni condizionali annidate; in ogni caso, non è possibile suddividere una struttura del genere in gruppi di istruzioni più piccoli, pertanto il punto fermo finale può apparire solo alla fine della struttura più esterna.

```
000000 IF ALTEZZA IS GREATER THAN 190
000000   THEN
000000       DISPLAY "LA PERSONA E` MOLTO ALTA!",
000000       PERFORM PERSONA-MOLTO-ALTA;
000000   ELSE
000000       IF ALTEZZA IS GREATER THAN 170
000000         THEN
000000             DISPLAY "LA PERSONA E` ABBASTANZA ALTA.",
000000             PERFORM PERSONA-ALTA;
000000         ELSE
000000             DISPLAY "LA PERSONA HA UN'ALTEZZA ",
000000                     "MEDIA O BASSA".
```

L'esempio mostra un'istruzione '**IF**' annidata, dove sono stati usati i vari separatori disponibili, per facilitare la lettura: la parola '**THEN**' non fa parte dell'istruzione, ma introduce qui le istruzioni da eseguire nel caso la condizione si avveri; la virgola viene usata per terminare le istruzioni singole, mentre il punto e virgola si usa per concludere quelle istruzioni dopo le quali si passa all'alternativa (introdotta dalla parola chiave '**ELSE**').

Il punto fermo finale è molto importante, perché rappresenta l'unico modo per stabilire dove finisca tutta la struttura, dal momento che

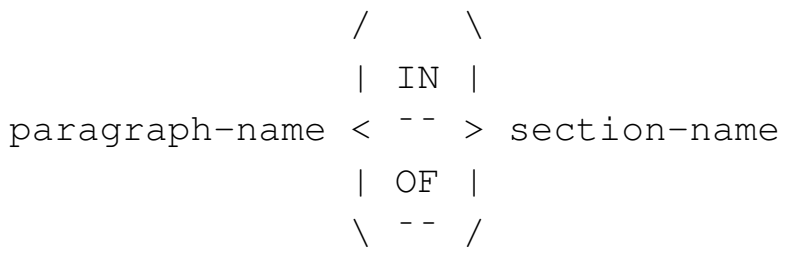
nel linguaggio non è previsto l'uso di parole come «end if».

72.11.4 Sezioni, paragrafi e qualificazione

Quando la parte procedurale del programma si suddivide in sezioni, i nomi dei paragrafi devono essere univoci soltanto nell'ambito della sezione in cui vengono dichiarati.

Quando si deve fare riferimento al nome di un paragrafo che non è unico nel programma, si deve usare la qualificazione per distinguere a quale sezione si sta facendo riferimento; eccezionalmente, se si tratta della sezione in cui ci si trova già, la qualificazione è implicita.

La qualificazione si ottiene aggiungendo la parola '**OF**', oppure '**IN**', seguita dal nome della sezione.



72.11.5 Espressioni aritmetiche

L'espressione aritmetica è ciò che si traduce in un valore numerico, eventualmente attraverso l'uso di operatori. Gli operatori aritmetici disponibili nel linguaggio COBOL sono molto pochi, limitando le possibilità alle quattro operazioni.

È importante osservare che gli operatori aritmetici, tranne nel caso delle parentesi, vanno separati dai loro argomenti; diversamente, il segno '**-**' verrebbe confuso come carattere che compone una parola. Per esempio, '**A - B**' è un'espressione che rappresenta una sottrazione, mentre '**A-B**' è una parola.

Tabella 72.137. Espressioni aritmetiche.

Espressione	Descrizione
+ *x*	Non modifica il segno di *x*.
- *x*	Inverte il segno di *x*.
x + *y*	Somma i due operandi.
x - *y*	Sottrae da *x* il valore di *y*.
x * *y*	Moltiplica i due operandi.
x / *y*	Divide il primo operando per il secondo.
(...)	Cambia la precedenza stabilendo che quanto contenuto tra parentesi va calcolato prima di ciò che si trova all'esterno.

L'ordine di precedenza nelle espressioni aritmetiche è quello consueto: prima gli operatori unari, che si applicano a un operando singolo, poi la moltiplicazione e la divisione, quindi la somma e la sottrazione.

72.11.6 Espressioni condizionali

Nel linguaggio COBOL si distinguono diversi tipi di espressioni condizionali elementari, che vengono descritte nelle sezioni successive. Le espressioni elementari, a loro volta, si possono combinare in espressioni composte, con l'uso di operatori booleani ed eventualmente con l'aiuto di parentesi tonde per modificare l'ordine di valutazione.

72.11.6.1 Condizioni di relazione

Le condizioni di relazione stabiliscono un confronto tra due valori, che possono essere rappresentati da variabili, costanti o da espressioni aritmetiche. Segue lo schema sintattico:

```
                        / IS [NOT] GREATER THAN \
                        |      ---   -------     |
                        | IS [NOT] LESS THAN      |
/ identifier-1       \  |      ---   ----        |  / identifier-2       \
|                    |  | IS [NOT] EQUAL TO       |  |                    |
< literal-1          >  <      ---   -----       >  < literal-2          >
|                    |  | IS [NOT] >              |  |                    |
\ arith-expression-1 /  |      ---   -            |  \ arith-expression-2 /
                        | IS [NOT] <              |
                        |      ---   -            |
                        \ IS [NOT] =              /
                               ---   -
```

Tabella 72.139. Significato degli operatori di relazione.

Operatore	Descrizione
`IS [NOT] GREATER THEN` `    ---   -------`	maggiore di, non maggiore di
`IS [NOT] >` `    ---   -`	maggiore di, non maggiore di

Operatore	Descrizione
`IS [NOT] LESS THEN` `    ---  ----`	minore di, non minore di
`IS [NOT] <` `    ---  -`	minore di, non minore di
`IS [NOT] EQUAL TO` `    ---  -----`	uguale a, diverso da
`IS [NOT] =` `    ---  -`	uguale a, diverso da
`IS GREATER THAN OR EQUAL TO` `   -------      --------`	maggiore o uguale a
`IS >=` `   --`	maggiore o uguale a
`IS LESS THAN OR EQUAL TO` `   ----      --------`	minore o uguale a
`IS <=` `   --`	minore o uguale a

Quando gli operandi sono entrambi numerici, indipendentemente dal fatto che la loro rappresentazione sia in forma di «indice» (‘**INDEX**’), compatta (‘**COMPUTATIONAL**’) o in forma di byte (‘**DISPLAY**’), il confronto si basa sul valore numerico che esprimono, tenendo conto del segno, se c’è, considerando positivi i valori senza segno.

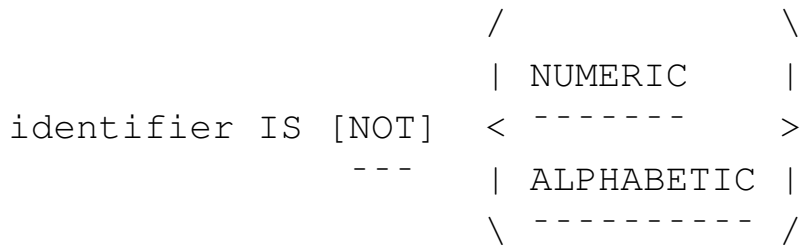
Quando si confrontano operandi alfanumerici, o quando anche uno solo è di tipo alfanumerico, il confronto avviene in modo lessicografico (in base all’ordinamento previsto dalla codifica adottata).

72.11.6.2 Condizioni di classe

La condizione di classe serve a stabilire se l'operando a cui si applica è numerico o alfabetico. È numerico un operando che è composto soltanto di cifre da '**0**' a '**9**', con il segno eventuale; è alfabetico un operando composto soltanto dal lettere alfabetiche ed eventualmente da spazi.

La condizione di classe si utilizza solo per verificare il contenuto di variabili che sono state dichiarate con una rappresentazione in byte ('**USAGE IS DISPLAY**').

Segue lo schema sintattico per esprimere la condizione di classe:



Naturalmente, se si usa la parola chiave '**NOT**', si intende invertire il significato della condizione.

72.11.6.3 Nomi di condizione

I nomi di condizione, che si dichiarano nella divisione '**DATA DIVISION**' con il numero di livello 88, servono a descrivere il confronto della variabile a cui si riferiscono con i valori che rappresentano.

Supponendo di avere dichiarato il nome di condizione '**PARI**' nel modo seguente:

```
000000     02  CODICE           PIC 9.
000000         88  PARI         0, 2, 4, 6, 8.
000000         88  DISPARI      1, 3, 5, 7, 9.
000000         88  BASSO        0 THRU 4.
000000         88  ALTO         5 THRU 9.
```

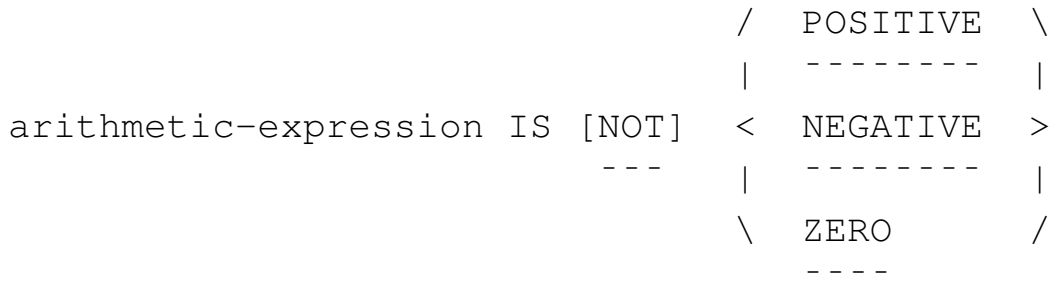
Nella divisione '**PROCEDURE DIVISION**' potrebbero apparire righe come quelle successive, per verificare che la variabile '**CODICE**' contenga un valore pari:

```
000000     IF PARI
000000     THEN
000000         PERFORM …;
000000     ELSE
000000         …
```

72.11.6.4 Condizioni di segno

«

La condizione di segno permette di stabilire se un'espressione aritmetica (e può essere anche solo una costante o una variabile numerica) è positiva, negativa o se vale esattamente zero:



72.11.6.5 Condizioni composte e negate

«

Attraverso gli operatori booleani comuni, si possono definire delle condizioni composte, oppure negate. Si utilizzano le parole chiave '**AND**', '**OR**' e '**NOT**' per esprimere gli operatori booleani noti

con lo stesso nome. Con l'ausilio delle parentesi tonde si possono modificare le precedenze nella valutazione delle espressioni.

Il linguaggio COBOL prevede una forma abbreviata per esprimere delle condizioni di relazione composte. Si osservi l'espressione seguente:

```
A > B OR A > C OR A < D
```

Questa potrebbe essere abbreviata così:

```
A > B OR > C OR < D
```

Tuttavia, si comprende che l'abbreviazione comporta maggiore difficoltà interpretativa nella lettura umana del programma sorgente.

72.11.7 Avverbi comuni

Per «avverbi comuni» qui si intendono delle parole chiave che possono far parte di varie istruzioni, fornendo però lo stesso tipo di funzionalità.

Tabella 72.144. Alcuni avverbi comuni.

Avverbio	Descrizione
`ROUNDED`	Quando una variabile deve ricevere il risultato di un'espressione aritmetica e non ha la possibilità di rappresentare in modo esatto i decimali, con l'uso dell'avverbio '**`ROUNDED`**' si chiede di assegnare un valore arrotondato nella cifra meno significativa.

Avverbio	Descrizione
`SIZE ERROR`	Quando una variabile deve ricevere il risultato di un'espressione aritmetica e non ha la possibilità di rappresentare alcune cifre più significative, si verifica un errore, che può essere espresso dalla condizione '**`SIZE ERROR`**'. La verifica di questa condizione implica l'indicazione di un'istruzione da eseguire se si verifica questo tipo di errore.
`CORRESPONDING`	Alcune istruzioni prevedono l'uso dell'avverbio '**`CORRESPONDING`**' per fare riferimento a variabili strutturate che contengono campi con lo stesso nome, pur non avendo la stessa struttura. Generalmente si usa per assegnare un gruppo di variabili a un altro gruppo, con lo stesso nome, in un'altra struttura.

72.12 Istruzioni della divisione «PROCEDURE DIVISION»

«

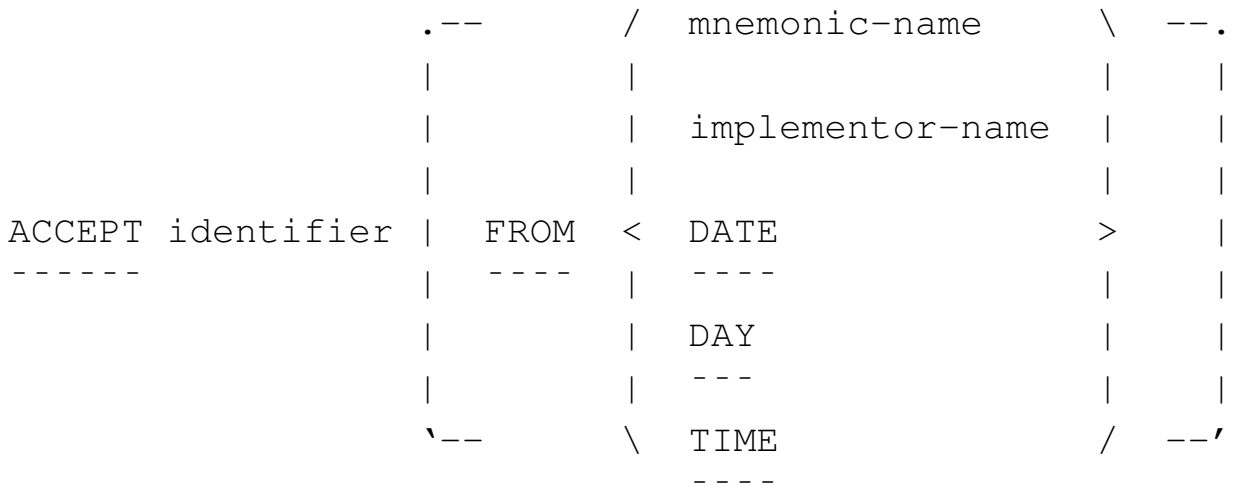
Nelle sezioni successive sono raccolti e descritti, in ordine alfabetico, i modelli sintattici delle istruzioni principali del linguaggio COBOL, da usare nella divisione '**`PROCEDURE DIVISION`**'.

72.12.1 Istruzione «ACCEPT»

«

L'istruzione '**`ACCEPT`**' permette di ricevere un'informazione da una fonte esterna al programma e di assegnarla a una variabile. Generalmente si usa questa istruzione per consentire all'utente l'inserimento

di un valore attraverso il terminale che sta utilizzando, ma, a seconda delle possibilità offerte dal compilatore, può servire anche per l'acquisizione di altri dati.



La fonte di dati per l'istruzione '**ACCEPT**' può essere dichiarata attraverso un nome mnemonico, definito nel paragrafo '**SPECIAL-NAMES**' della sezione '**INPUT-OUTPUT SECTION**' (sezione 72.4.3), un nome particolare che dipende da funzionalità speciali del compilatore, oppure un nome che fa riferimento alla data o all'orario attuale (le parole chiave '**DATE**', '**DAY**', '**TIME**').

Tabella 72.146. Parole chiave standard per definire l'accesso alle informazioni data-orario.

Parola chiave	Descrizione
`DATE`	Fornisce la data attuale, formata da sei cifre numeriche, secondo la forma ***yymmdd*** (due cifre per l'anno, due per il mese e due per il giorno).

Parola chiave	Descrizione
`DAY`	Fornisce il giorno attuale, formato da cinque cifre numeriche, secondo la forma ***yyddd*** (due cifre per l'anno e tre cifre per il giorno, espresso in quantità di giorni trascorsi dall'inizio dell'anno).
`TIME`	Fornisce l'orario attuale, formato da otto cifre numeriche, secondo la forma ***hhmmsscc*** (due cifre per l'ora, due per i minuti, due per i secondi e due per i centesimi di secondo).

L'esempio seguente dimostra il funzionamento e l'utilizzo di queste parole chiave standard:

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   TEST-ACCEPT-DATE-TIME.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-02-27.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 DATA DIVISION.
000900*
001000 WORKING-STORAGE SECTION.
001100 77  MY-DATE      PIC X(30).
001200 77  MY-DAY       PIC X(30).
001300 77  MY-TIME      PIC X(30).
001400*
001500 PROCEDURE DIVISION.
001600*
001700 MAIN.
001800     ACCEPT MY-DATE FROM DATE.
001900     ACCEPT MY-DAY  FROM DAY.
```

```
002000     ACCEPT MY-TIME FROM TIME.
002100     DISPLAY "DATE: ", MY-DATE.
002200     DISPLAY "DAY:  ", MY-DAY.
002300     DISPLAY "TIME: ", MY-TIME.
002400*
002500     STOP RUN.
002600*
```

Avviando questo programma il giorno 27 gennaio 2005, alle ore 13:30.45, si dovrebbe ottenere il risultato seguente:

```
DATE: 050227
DAY:  05058
TIME: 13304500
```

Tabella 72.149. Parole chiave non standard.

Parola chiave	Descrizione
`CONSOLE` `SYSIN`	Quando non si specifica la fonte dei dati per l'istruzione '**`ACCEPT`**', si intende il terminale dal quale il programma è stato avviato; spesso, i compilatori considerano l'uso della parola chiave '**`CONSOLE`**', o di '**`SYSIN`**', come sinonimo di questo comportamento, anche se è quello predefinito.
`COMMAND-LINE`	I compilatori per i sistemi Unix consentono spesso di accedere al contenuto della riga di comando usata per avviare il programma, attraverso l'uso di questa parola chiave.

L'esempio successivo dimostra l'uso di un nome mnemonico

per dichiarare l'origine dei dati. Sono evidenziate le righe più significative:

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   TEST-ACCEPT.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-02-27.
000500*
000600 ENVIRONMENT DIVISION.
000700 CONFIGURATION SECTION.
000800 SOURCE-COMPUTER.
000900     OPENCOBOL.
001000 SPECIAL-NAMES.
001100     CONSOLE IS STANDARD-INPUT.
001200*
001300 DATA DIVISION.
001400*
001500 WORKING-STORAGE SECTION.
001600 77  MESSAGGIO            PIC X(30).
001700*
001800 PROCEDURE DIVISION.
001900*
002000 MAIN.
002100     DISPLAY "INSERISCI IL MESSAGGIO".
002200     ACCEPT MESSAGGIO FROM STANDARD-INPUT.
002300     DISPLAY "HAI INSERITO: ", MESSAGGIO.
002400*
002500     STOP RUN.
002600*
```

72.12.2 Istruzione «ADD»

L'istruzione '**ADD**' consente di eseguire delle somme. Sono previsti diversi formati per l'utilizzo di questa istruzione.

```
      /               \
      | identifier-1   |
ADD   <               >...  TO { identifier-n  [ROUNDED] }...
---   | literal-1     |      --                   -------
      \               /

     [ ON SIZE ERROR  imperative-statement ]
          ---- -----
```

Nello schema sintattico appena mostrato, si vede che dopo la parola chiave '**ADD**' si elencano delle costanti o variabili con valore numerico, da sommare assieme, sommando poi quanto ottenuto al contenuto delle variabili specificate dopo la parola chiave '**TO**'. L'opzione '**ROUNDED**' richiede di eseguire un arrotondamento se la variabile ricevente non può rappresentare in modo esatto il valore; l'opzione '**SIZE ERROR**' serve a eseguire un'istruzione nel caso una delle variabili riceventi non possa accogliere la porzione più significativa del valore ottenuto dalla somma. Si osservi l'esempio seguente:

```
000000     ADD 1, 2, 3, TO A.
```

Supponendo che la variabile '**A**', prima della somma contenga il valore 10, dopo la somma contiene il valore 16 (1+2+3+10).

```
      /               \  /               \
      | identifier-1   |  | identifier-2   |
ADD   <                 >  <                >...
---   | literal-1      |  | literal-2      |
      \               /  \               /

    GIVING { identifier-n  [ROUNDED] }...
    ------                  -------

    [ ON SIZE ERROR  imperative-statement ]
         ---- -----
```

Quando al posto della parola chiave '**TO**', si usa '**GIVING**', la somma dei valori che precede tale parola chiave viene assegnata alle variabili indicate dopo, senza tenere in considerazione il loro valore iniziale nella somma. Valgono le stesse considerazioni già fatte a proposito delle opzioni '**ROUNDED**' e '**SIZE ERROR**'. Si osservi l'esempio seguente:

```
000000     ADD 1, 2, 3, GIVING A.
```

Qualunque sia il valore iniziale della variabile '**A**', dopo la somma questa contiene il valore 6 (1+2+3).

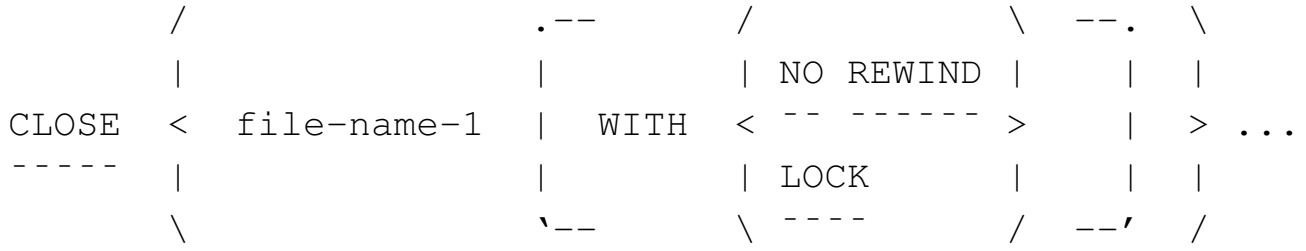
```
      /               \
      | CORR          |
ADD   < ----          >  identifier-1  TO  identifier-2  [ROUNDED]
---   | CORRESPONDING |                --                  -------
      \ ------------- /

    [ ON SIZE ERROR  imperative-statement ]
         ---- -----
```

In questo ultimo caso, la somma fa riferimento a variabili strutturate, dove i campi della prima variabile devono essere sommati ai

campi della seconda variabile che hanno lo stesso nome della prima. Valgono le stesse considerazioni già fatte a proposito delle opzioni '**ROUNDED**' e '**SIZE ERROR**'.

72.12.3 Istruzione «CLOSE»

Attraverso l'istruzione '**CLOSE**' si può chiudere un file aperto. Questa istruzione non riguarda i file definiti esplicitamente per le funzionalità di riordino e fusione del COBOL, perché questi non vengono aperti. La sintassi dell'istruzione può essere più o meno ricca, a seconda delle estensioni che offre il compilatore; tuttavia, lo schema seguente si adatta alla maggior parte delle situazioni:



Il file indicato viene chiuso, eventualmente con delle opzioni. Se si tratta di un file sequenziale a nastro, si può utilizzare l'opzione '**NO REWIND**', con la quale si vuole evitare che il nastro venga riavvolto automaticamente dopo la chiusura, così da poter accedere eventualmente a un file successivo, già esistente o da creare sullo stesso nastro. L'opzione '**LOCK**' serve a impedire che il file possa essere riaperto nel corso del funzionamento del programma.

Nel caso si utilizzino dei nastri, quelli che il programma ha chiuso senza riavvolgere, vengono comunque riavvolti alla conclusione del programma stesso; inoltre, alla conclusione del programma vengono chiusi automaticamente i file che sono rimasti ancora aperti.

72.12.4 Istruzione «COMPUTE»

L'istruzione '**COMPUTE**' consente di calcolare un'espressione aritmetica, assegnando il risultato a una o più variabili:

```
COMPUTE  { identifier  [ROUNDED] }... = arithmetic-expression
-------                 -------

    [ ON SIZE ERROR  imperative-statement ]
         ---- -----
```

La variabile che nello schema sintattico appare con il nome ***identifier*** deve essere scalare e di tipo numerico, anche se può contenere una maschera di modifica. Possono essere indicate più variabili a sinistra del segno '**=**' e ognuna riceve una copia del risultato dell'espressione alla destra.

L'opzione '**ROUNDED**' serve a richiedere un arrotondamento se la variabile ricevente non può rappresentare il risultato con la stessa precisione ottenuta dal calcolo dell'espressione; l'opzione '**SIZE ERROR**' consente di richiamare un'istruzione nel caso una delle variabili riceventi non fosse in grado di contenere la parte più significativa del valore ottenuto calcolando l'espressione.

```
000000     COMPUTE D = A * B + C.
```

L'esempio mostra che si vuole assegnare alla variabile '**D**' il risultato dell'espressione '**A * B + C**' (A moltiplicato B, sommato a C).

72.12.5 Istruzione «DELETE»

L'istruzione '**DELETE**' cancella un record logico da un file organizzato in modo relativo o a indice (sono esclusi i file organizzati in modo sequenziale).

```
DELETE  file-name  RECORD  [INVALID KEY imperative-statement]
------                      -------
```

Per poter cancellare un record è necessario che il file sia stato aperto in lettura e scrittura (‘**I-O**’).

Se il file viene utilizzato con un accesso sequenziale, l’opzione ‘**INVALID KEY**’ non è applicabile e non deve essere scritta nell’istruzione. Inoltre, utilizzando un accesso sequenziale, prima di eseguire un’istruzione ‘**DELETE**’ è necessario che il puntatore del record sia stato posizionato attraverso un’istruzione ‘**READ**’. L’istruzione ‘**READ**’ deve precedere immediatamente l’istruzione ‘**DELETE**’, che così può cancellare il record appena letto.

Quando il file viene utilizzato con un accesso diretto (‘**RANDOM**’) o dinamico (‘**DYNAMIC**’), l’opzione ‘**INVALID KEY**’ è obbligatoria, a meno di avere dichiarato un’azione alternativa, in caso di errore, nella zona di istruzioni definite come ‘**DECLARATIVES**’, all’inizio della divisione ‘**PROCEDURE DIVISION**’. Per individuare il record da cancellare, si fa riferimento alla chiave, come specificato dalla dichiarazione ‘**RECORD KEY**’, associata al file in questione. Se si tenta di cancellare un record indicando una chiave che non esiste, si ottiene l’errore che fa scattare l’esecuzione dell’istruzione associata all’opzione ‘**INVALID KEY**’.

Dipende dal compilatore il modo in cui viene trattato effettivamente il record da cancellare: questo potrebbe essere sovrascritto con un valore prestabilito, oppure potrebbe essere semplicemente segnato per la cancellazione; in ogni caso, il record non viene cancellato fisicamente dal file.

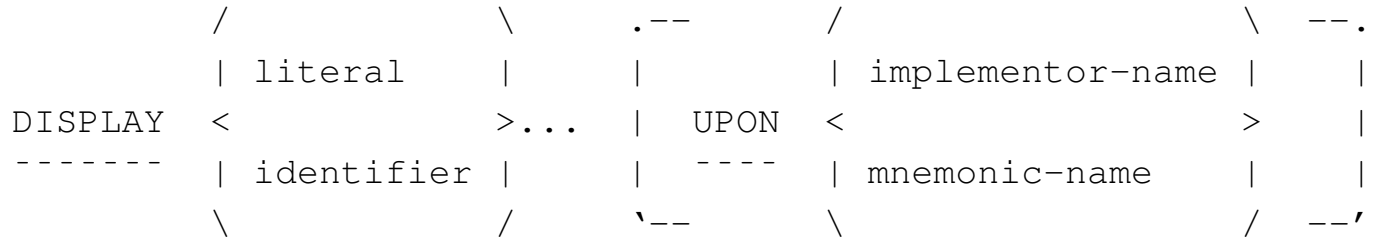
Quando si accede al file attraverso un indice, bisogna considerare

che la cancellazione può provocare la comparsa di record con chiavi doppie, se la cancellazione implica la sovrascrittura del record con un certo valore; inoltre, se il file contiene record con chiavi doppie, la cancellazione di un record specificando la sua chiave, può portare a cancellare quello sbagliato. Pertanto, in presenza di file a indice con chiavi doppie, conviene usare un accesso sequenziale per individuare in modo esatto il record da cancellare.

72.12.6 Istruzione «DISPLAY»

«

L'istruzione '**DISPLAY**' consente di emettere un messaggio attraverso un dispositivo che consenta di farlo. Generalmente, se usata senza opzioni, la visualizzazione avviene attraverso il terminale dal quale è stato avviato il programma.



Osservando lo schema sintattico si vede che dopo la parola chiave '**DISPLAY**' si possono mettere delle costanti letterali o dei nomi di variabile. Questi elementi possono rappresentare sia valori alfanumerici, sia numerici (tuttavia, il compilatore potrebbe rifiutarsi di accettare delle variabili di tipo '**INDEX**'): è il compilatore che provvede a eseguire le conversioni necessarie. L'elenco di costanti o di variabili viene concatenato prima della visualizzazione.

L'aggiunta dell'opzione '**UPON**' consente di specificare dove deve essere emesso il messaggio. Si può indicare una parola chiave definita dal compilatore, che identifica qualche tipo di di-

spositivo, oppure un nome mnemonico, da specificare nel paragrafo '**SPECIAL-NAMES**' della sezione '**INPUT-OUTPUT SECTION**' (sezione 72.4.3).

Tabella 72.161. Parole chiave non standard.

Parola chiave	Descrizione
`CONSOLE` `SYSOUT`	Quando non si specifica la fonte dei dati per l'istruzione '**ACCEPT**', si intende il terminale dal quale il programma è stato avviato; spesso, i compilatori considerano l'uso della parola chiave '**CONSOLE**', o di '**SYSOUT**', come sinonimo di questo comportamento, anche se è quello predefinito.

L'esempio successivo mostra un uso abbastanza comune dell'istruzione '**DISPLAY**':

```
000000     DISPLAY "ATTENZIONE: ", A, " + ", B, " = ", C.
```

L'esempio mostra in particolare il concatenamento che si vuole ottenere. Si ricorda che non è importante se le variabili utilizzate nell'istruzione sono alfanumeriche o numeriche, perché è il compilatore che provvede a convertire tutto nel modo più appropriato al tipo di dispositivo che deve emettere il messaggio.

72.12.7 Istruzione «DIVIDE»

L'istruzione '**DIVIDE**' consente di eseguire delle divisioni, fornendone il risultato ed eventualmente il resto. Sono previsti diversi formati per l'utilizzo di questa istruzione.

```
          /               \
          | identifier-1 |
DIVIDE    <               >  INTO  { identifier-2 [ROUNDED] }...
------    | literal-1     |  ----                  -------
          \               /

     [ ON SIZE ERROR  imperative-statement ]
          ---- -----
```

Nello schema sintattico appena mostrato, si vede che dopo la parola chiave '**DIVIDE**' viene indicato un valore, in forma costante o attraverso una variabile; questo valore viene diviso per la variabile indicata dopo la parola chiave '**INTO**' e il risultato viene assegnato alla stessa variabile che funge da divisore. Se appaiono più variabili dopo la parola '**INTO**', la divisione viene ripetuta per ognuna di quelle, assegnando rispettivamente il risultato.

L'opzione '**ROUNDED**' richiede di eseguire un arrotondamento se la variabile ricevente non può rappresentare in modo esatto il valore; l'opzione '**SIZE ERROR**' serve a eseguire un'istruzione nel caso una delle variabili riceventi non possa accogliere la porzione più significativa del valore ottenuto dalla somma. Si osservi l'esempio seguente:

```
000000     DIVIDE 100 INTO A.
```

Supponendo che la variabile '**A**', prima della divisione contenga il valore 5, dopo l'operazione contiene il valore 20 (100/5). Si potrebbe scrivere la stessa cosa utilizzando l'istruzione '**COMPUTE**':

```
000000     COMPUTE A = 100 / A.
```

Lo schema sintattico successivo mostra l'utilizzo di '**DIVIDE**' in modo da non alterare i valori utilizzati come divisori:

```
          /               \  /      \  /               \
          | identifier-1 |  | INTO |  | identifier-2 |
DIVIDE    <               >  < ---- >  <               >
------    | literal-1    |  | BY   |  | literal-2    |
          \               /  \ --   /  \               /

     GIVING identifier-3  [ROUNDED]
     ------                -------

     [ REMAINDER  identifier-4  [ROUNDED] ]
       ---------

     [ ON SIZE ERROR  imperative-statement ]
          ---- -----
```

Nella forma appena mostrata, dove le parole '**INTO**' e '**BY**' sono equivalenti, la divisione avviene immettendo il risultato dell'operazione nella variabile indicata dopo la parola '**GIVING**'. Valgono le stesse considerazioni già fatte a proposito delle opzioni '**ROUNDED**' e '**SIZE ERROR**'. Si osservi l'esempio seguente che ripete sostanzialmente l'esempio già mostrato in precedenza:

```
000000     DIVIDE 100 BY 5 GIVING A.
```

Utilizzando l'opzione '**REMAINDER**', si fa in modo che il resto della divisione venga inserito nella variabile che segue tale parola. Tuttavia, si osservi che per resto si intende ciò che rimane moltiplicando il quoziente ottenuto (***identifier-3***) per il divisore (***identifier-2*** o ***literal-2***), sottraendo poi questo valore ottenuto dal dividendo (***identifier-1*** o ***literal-1***). Si osservi l'esempio che segue:

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   TEST-DIVIDE.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-02-27.
000500*
```

```
000600 ENVIRONMENT DIVISION.
000700*
000800 DATA DIVISION.
000900*
001000 WORKING-STORAGE SECTION.
001100 77  A            PIC 9(10)V99.
001200 77  B            PIC 9(10)V99.
001400*
001500 PROCEDURE DIVISION.
001600*
001700 MAIN.
001800     DIVIDE 100 BY 3 GIVING A REMAINDER B.
001900     DISPLAY "100 / 3 = ", A, " CON IL RESTO DI ", B.
002000*
002100     STOP RUN.
002200*
```

Una volta compilato questo programma, se viene messo in funzione si dovrebbe ottenere il risultato seguente, che dovrebbe chiarire di che tipo di resto si parla con questa istruzione:

```
100 / 3 = 0000000033.33 CON IL RESTO DI 0000000000.01
```

72.12.8 Istruzione «EXIT»

L'istruzione '**EXIT**' serve a concludere anticipatamente l'esecuzione di un gruppo di paragrafi, attraverso un'istruzione '**PERFORM**'. L'istruzione '**EXIT**' deve essere usata da sola, all'interno di un paragrafo tutto per sé:

```
paragraph-name

    EXIT.
    ----
```

Si osservi che un programma COBOL scritto in modo ordinato non dovrebbe avere bisogno di questa istruzione.

```
000000     PERFORM UNO THRU TRE.
000000     …
000000 UNO.
000000     …
000000 DUE.
000000     …
000000     IF …
000000       THEN
000000           GO TO TRE.
000000     …
000000 TRE.
000000     EXIT.
000000 QUATTRO.
000000     …
```

L'esempio appena mostrato serve a dare un'idea del significato dell'istruzione '**EXIT**': la chiamata iniziale con l'istruzione '**PERFORM**' richiede l'esecuzione sequenziale dei paragrafi da '**UNO**' a '**TRE**', ma nel paragrafo '**DUE**' si verifica una condizione e al suo avverarsi si esegue un salto ('**GO TO**') al paragrafo '**TRE**', che conclude comunque la chiamata principale.

Come già accennato, dal momento che l'uso dell'istruzione '**EXIT**' implica l'utilizzo di '**GO TO**', che notoriamente complica la comprensibilità di un programma in modo eccessivo, entrambe queste istruzioni sono da evitare accuratamente.

72.12.9 Istruzione «GO TO»

«

L'istruzione '**GO TO**' consente di saltare all'inizio di un paragrafo specificato, senza ritorno. Sono previsti due modi di utilizzo:

```
GO TO  procedure-name
--
```

Oppure:

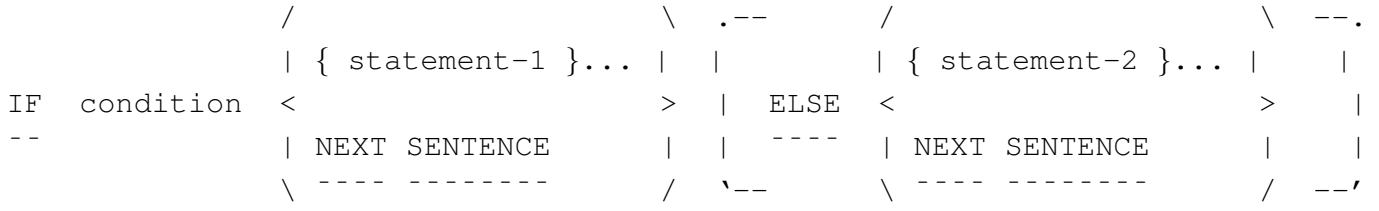
```
GO TO  { procedure-name }... DEPENDING ON identifier
--                           ---------
```

Nel primo caso, l'esecuzione dell'istruzione passa il controllo al paragrafo indicato; nel secondo, viene scelto il paragrafo a cui passare il controllo in base al valore indicato dopo la parola '**DEPENDING**'. Il valore in questione deve essere un numero intero, rappresentato attraverso una variabile (altrimenti non ci sarebbe motivo di usarlo), dove il valore uno rappresenta il primo paragrafo nominato dopo le parole '**GO TO**' e il valore ***n*** rappresenta il paragrafo ***n***-esimo dello stesso elenco.

L'utilizzo dell'istruzione '**GO TO**' complica la lettura di un programma sorgente COBOL e, secondo il parere di molti, andrebbe abolita. Si veda a questo proposito: Edsger W. Dijkstra, *Go To Statement Considered Harmful*, 1968, *http://www.acm.org/classics/oct95/* , *http://www.cs.utsa.edu/~wagner/CS3723/nogoto/harm2.html* e altri indirizzi.

72.12.10 Istruzione «IF»

L'istruzione '**IF**' consente di eseguire un gruppo di istruzioni solo se si verifica una condizione, o se questa non si verifica. Il formato di questa istruzione è visibile nello schema seguente:



Le istruzioni che seguono immediatamente la condizione (***statement-1***), vengono eseguite se la condizione si avvera; le istruzioni del gruppo che segue la parola '**ELSE**' vengono eseguite se la condizione non si avvera. Le istruzioni del primo e del secondo gruppo, possono contenere altre istruzioni '**IF**'.

Si osservi che la parola '**THEN**' è un separatore, ma viene usata spesso per migliorare la lettura di un'istruzione '**IF**':

```
000000 IF ALTEZZA IS GREATER THAN 190
000000   THEN
000000       DISPLAY "LA PERSONA E` MOLTO ALTA!",
000000       PERFORM PERSONA-MOLTO-ALTA;
000000   ELSE
000000       IF ALTEZZA IS GREATER THAN 170
000000         THEN
000000             DISPLAY "LA PERSONA E` ABBASTANZA ALTA.",
000000             PERFORM PERSONA-ALTA;
000000         ELSE
000000             DISPLAY "LA PERSONA HA UN'ALTEZZA ",
000000                     "MEDIA O BASSA".
```

L'esempio mostra un'istruzione '**IF**' che ne contiene un'altra dopo

la parola '**ELSE**'. Si può osservare che il punto fermo che conclude il gruppo di istruzioni appare solo alla fine della prima istruzione '**IF**' e costituisce l'unico modo per poter comprendere dove finisce tutta la struttura. Si osservi che la rappresentazione della struttura con dei rientri appropriati serve per individuare facilmente i livelli di annidamento esistenti.

Data la particolarità di questo esempio, i rientri potrebbero essere gestiti in modo diverso, per sottolineare la presenza di una serie di condizioni alternative ('**ELSE IF**'):

```
000000 IF ALTEZZA IS GREATER THAN 190
000000 THEN
000000     DISPLAY "LA PERSONA E` MOLTO ALTA!",
000000     PERFORM PERSONA-MOLTO-ALTA;
000000 ELSE IF ALTEZZA IS GREATER THAN 170
000000 THEN
000000     DISPLAY "LA PERSONA E` ABBASTANZA ALTA.",
000000     PERFORM PERSONA-ALTA;
000000 ELSE
000000     DISPLAY "LA PERSONA HA UN'ALTEZZA MEDIA O BASSA".
```

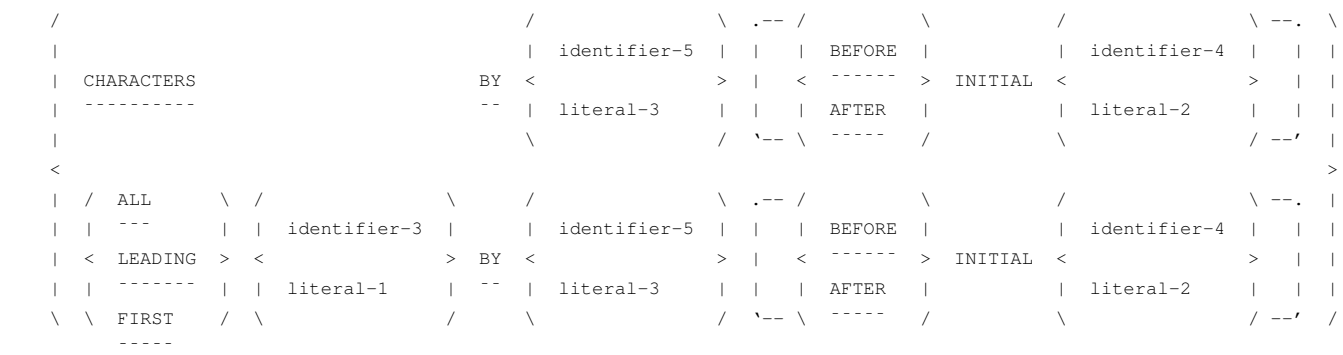
72.12.11 Istruzione «INSPECT»

«

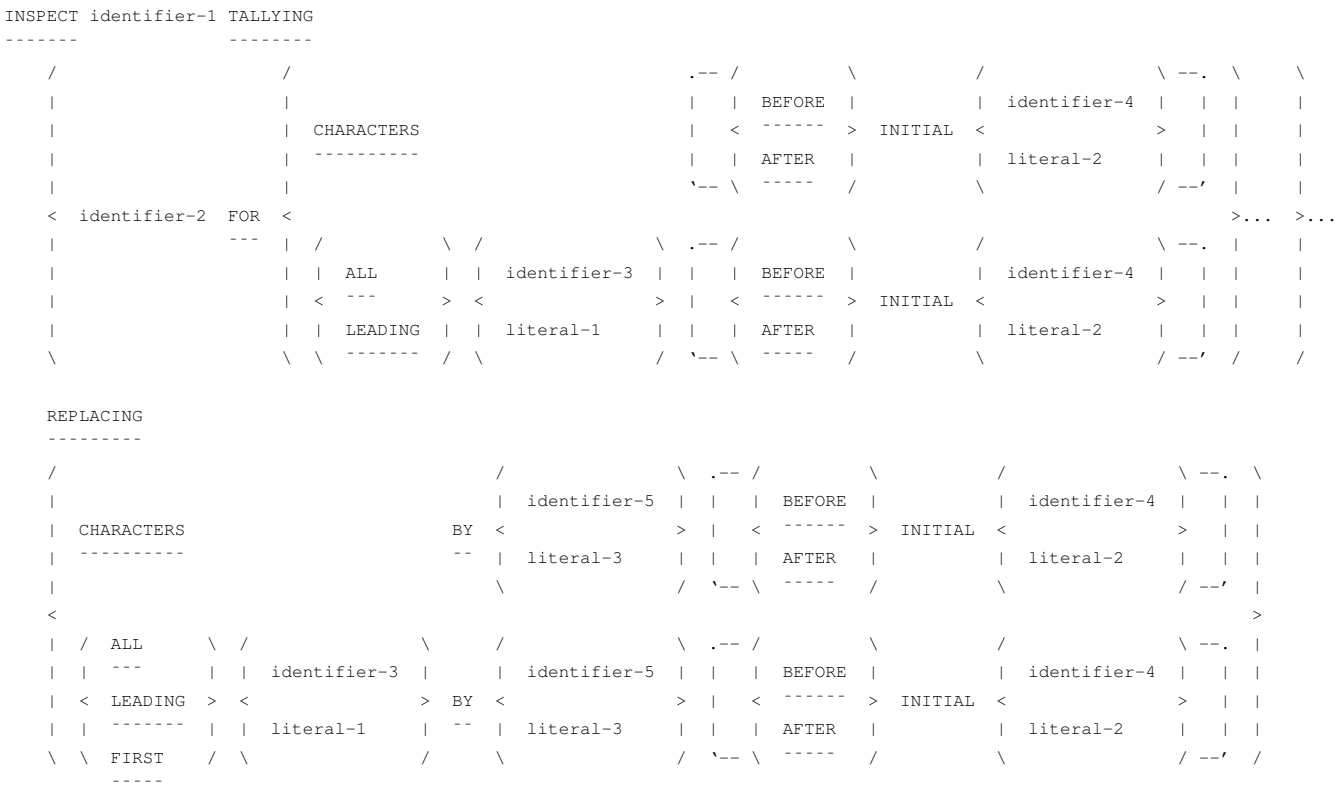
L'istruzione '**INSPECT**' consente di scandire una variabile contenente una stringa alfanumerica, allo scopo di contare alcuni caratteri o di rimpiazzare alcuni dei caratteri della stringa. Sono previsti tre schemi sintattici per l'uso di questa istruzione, per il conteggio, la sostituzione, oppure per entrambe le cose simultaneamente.

Conteggio dei caratteri

Sostituzione di caratteri



Conteggio e sostituzione



In tutti gli schemi sintattici, la variabile indicata dopo la parola '**INSPECT**', che viene annotata come ***identifier-1***, deve contenere una stringa di caratteri, da scandire.

L'opzione '**BEFORE**' o '**AFTER**', permette di individuare una posizione nella stringa, da prendere come limite finale, o come punto iniziale, per l'elaborazione. In pratica, la variabile ***identifier-4***, o la costante letterale ***literal-2***, serve a rappresentare una sottostringa (anche un solo carattere), che all'interno della stringa complessiva si trova per prima (a partire da sinistra); se si usa la parola '**BEFORE**', l'elaborazione deve avvenire nella parta iniziale della stringa, fino a quella sottostringa di riferimento esclusa; se si usa la parola '**AFTER**', l'elaborazione deve avvenire nella parta finale della stringa, subito dopo quella sottostringa. Naturalmente, se la sottostringa indicata non esiste nella stringa, è come se l'opzione '**BEFORE**' o '**AFTER**' non fosse

stata aggiunta.

Con il primo schema sintattico, si vogliono contare i caratteri della stringa che soddisfano certe condizioni. Il conteggio viene eseguito incrementando il valore contenuto nella variabile indicata nello schema come ***identifier-2***, che deve essere numerica. Si osservi che la variabile non viene azzerata automaticamente, pertanto il suo valore iniziale viene sommato al conteggio eseguito.

Il conteggio può riguardare tutti i caratteri della stringa o della porzione iniziale o finale selezionata, utilizzando la parola '**CHARACTERS**'. Si osservi l'esempio successivo che utilizza solo questo tipo di conteggio.

Listato 72.180. Programma elementare che scandisce una stringa e conta i caratteri contenuti.

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   TEST-INSPECT-TALLYING-1.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 INSTALLATION. NANOLINUX IV,
000500               OPENCOBOL 0.31,
000600 DATE-WRITTEN. 2005-03-15.
000700*
000800 ENVIRONMENT DIVISION.
000900*
001000 DATA DIVISION.
001100*
001200 WORKING-STORAGE SECTION.
001300 77  STRINGA-DI-CARATTERI PIC X(30).
001400 77  CONTATORE-1          PIC 99 VALUE IS 0.
001500 77  CONTATORE-2          PIC 99 VALUE IS 0.
001600 77  CONTATORE-3          PIC 99 VALUE IS 0.
001700*
001800 PROCEDURE DIVISION.
```

```
001900*------------------------ LIVELLO 0 ----------------
002000 MAIN.
002100     MOVE "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123"
002200          TO STRINGA-DI-CARATTERI.
002300     INSPECT STRINGA-DI-CARATTERI
002400             TALLYING CONTATORE-1
002500             FOR CHARACTERS,
002600             TALLYING CONTATORE-2
002700             FOR CHARACTERS BEFORE INITIAL "H",
002800             TALLYING CONTATORE-3
002900             FOR CHARACTERS AFTER INITIAL "H".
003000     DISPLAY "CONTATORI: ", CONTATORE-1, " ",
003100             CONTATORE-2, " ", CONTATORE-3.
003200     STOP RUN.
```

L'esempio appena mostrato utilizza un'istruzione '**INSPECT**' per contare tre cose in una stringa, con una sola scansione: i caratteri contenuti in tutta la stringa; i caratteri fino alla comparsa della prima lettera «H»; i caratteri che si trovano dopo la lettera «H»:

```
              30 caratteri
     <--------------------------->
     ABCDEFGHIJKLMNOPQRSTUVWXYZ0123
     <-----> <-------------------->
  7 caratteri      22 caratteri
```

Compilando l'esempio e avviando il programma eseguibile che si ottiene, si dovrebbe vedere il risultato seguente:

```
CONTATORI: 30 07 22
```

Con la parola '**ALL**' si intendono contare tutte le corrispondenze con una certa sottostringa (***identifier-3*** o ***literal-1***), contenuta nella stringa complessiva o nella porzione specificata successivamente. Con la

parola '**LEADING**', si vogliono contare solo le corrispondenze che avvengono in modo contiguo, purché inizino dal principio della zona di interesse.

Listato 72.183. Programma elementare che scandisce una stringa e conta i caratteri che corrispondono a delle sottostringhe.

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   TEST-INSPECT-TALLYING-2.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 INSTALLATION. NANOLINUX IV,
000500               OPENCOBOL 0.31,
000600 DATE-WRITTEN. 2005-03-15.
000700*
000800 ENVIRONMENT DIVISION.
000900*
001000 DATA DIVISION.
001100*
001200 WORKING-STORAGE SECTION.
001300 77  STRINGA-DI-CARATTERI PIC X(30).
001400 77  CONTATORE-1          PIC 99 VALUE IS 0.
001500 77  CONTATORE-2          PIC 99 VALUE IS 0.
001600 77  CONTATORE-3          PIC 99 VALUE IS 0.
001700*
001800 PROCEDURE DIVISION.
001900*------------------------ LIVELLO 0 -----------------
002000 MAIN.
002100     MOVE "ABCDEFGHIAAAABBBBCCCCDDDDEEEEF"
002200          TO STRINGA-DI-CARATTERI.
002300     INSPECT STRINGA-DI-CARATTERI
002400             TALLYING CONTATORE-1
002500             FOR ALL "E",
002600             TALLYING CONTATORE-2
002700             FOR LEADING "A" AFTER INITIAL "I",
002800             TALLYING CONTATORE-3
```

```
002900              FOR LEADING "B" BEFORE INITIAL "I".
003000     DISPLAY "CONTATORI: ", CONTATORE-1, " ",
003100             CONTATORE-2, " ", CONTATORE-3.
003200     STOP RUN.
```

In questo esempio viene cercata la corrispondenza con tutte le lettere «E»; le lettere «A» adiacenti che iniziano a partire dalla prima apparizione della lettera «I»; le lettere «B» adiacenti e iniziali, che si trovano prima di quella stessa lettera «I».

```
          5 lettere «E»
       -------------------------
       |                     ||||
    ABCDEFGHIAAAABBBBCCCCDDDDEEEEF
          |\\\\
          | ------4 lettere «A» adiacenti e iniziali
          |
  lettera «I» di riferimento

Non ci sono lettere «B» adiacenti e iniziali
prima del riferimento.
```

Compilando l'esempio e avviando il programma eseguibile che si ottiene, si dovrebbe vedere il risultato seguente:

```
CONTATORI: 05 04 00
```

Il secondo schema sintattico mostra l'uso di '**INSPECT**' per rimpiazzare delle sottostringhe. L'interpretazione dello schema è simile a quella del conteggio, con la differenza che si aggiunge la parola chiave '**BY**', che ha alla sinistra la sottostringa da rimpiazzare e alla destra il suo nuovo valore. Quando si usa la parola '**CHARACTERS**', si intende rimpiazzare tutta la stringa (o tutta la porzione prima o dopo un

certo riferimento), con qualcosa con un carattere; le parole '**ALL**' e '**LEADING**' funzionano sostanzialmente come nel conteggio, riferendosi a tutte le sottostringhe di un certo tipo o a tutte le sottostringhe iniziali e adiacenti, dello stesso tipo. In questo schema, si aggiunge la parola '**FIRST**', che identifica solo una prima corrispondenza, non ripetuta.

Listato 72.186. Programma che scandisce una stringa e sostituisce alcuni suoi contenuti. Il programma sfrutta un'estensione al linguaggio standard, che permette di eseguire più sostituzioni in una sola istruzione '**INSPECT**'.

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   TEST-INSPECT-REPLACING.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 INSTALLATION. NANOLINUX IV,
000500               OPENCOBOL 0.31,
000600 DATE-WRITTEN. 2005-03-15.
000700*
000800 ENVIRONMENT DIVISION.
000900*
001000 DATA DIVISION.
001100*
001200 WORKING-STORAGE SECTION.
001300 77  STRINGA-DI-CARATTERI PIC X(30).
001400*
001500 PROCEDURE DIVISION.
001600*------------------------ LIVELLO 0 ----------------
001700 MAIN.
001800     MOVE "AAAAAABBBBBBCCCCCCDDDDDDEEEEEE"
001900          TO STRINGA-DI-CARATTERI.
002000     INSPECT STRINGA-DI-CARATTERI REPLACING
002100             CHARACTERS BY "X" AFTER INITIAL "DDD",
002200             LEADING "BB" BY "YZ"
```

```
002250                          AFTER INITIAL "AAAAAA",
002300                    FIRST "C" BY "W",
002400                    ALL "C" BY "P".
002500           DISPLAY STRINGA-DI-CARATTERI.
002600           STOP RUN.
```

L'esempio appena mostrato sfrutta un'estensione al linguaggio tradizionale, in modo da ottenere più sostituzioni con una sola passata. L'esempio fatto in questo modo permette di capire cosa succede in queste situazioni particolari.

```
AAAAAABBBBBBCCCCCCDDDDDDEEEEEE
                      XXXXXXXXX  CHARACTERS BY "X" AFTER INITIAL "DDD"
      YZYZYZ                     LEADING "BB" BY "YZ" AFTER INITIAL "AAAAAA"
            W                    FIRST "C" BY "W"
             PPPPP               ALL "C" BY "P"
AAAAAAYZYZYZWPPPPPDDDXXXXXXXXX
```

Compilando l'esempio e avviando il programma eseguibile che si ottiene, si dovrebbe vedere il risultato seguente che rappresenta soltanto il contenuto finale della variabile elaborata:

```
AAAAAAYZYZYZWPPPPPDDDXXXXXXXXX
```

72.12.12 Istruzione «MOVE»

«

L'istruzione '**MOVE**' copia o assegna un valore in una o più variabili di destinazione. Sono disponibili due modi di usare questa istruzione:

```
       /              \
       | identifier-1  |
MOVE   <              >  TO  { identifier-2 }...
----   | literal-1    |  --
       \              /
```

Oppure:

```
       /                \
       | CORRESPONDING |
MOVE   < ------------- >  identifier-1  TO  { identifier-2 }...
----   | CORR          |                  --
       \ ----          /
```

Nel primo caso, ciò che appare dopo la parola chiave '**MOVE**' può essere il nome di una variabile, oppure una costante. Il valore contenuto nella variabile o rappresentato dalla costante, viene copiato in tutte le variabili indicate dopo la parola '**TO**', rispettando eventualmente le regole di modifica stabilite dai modelli di definizione delle variabili.

Nel secondo caso, avendo aggiunto la parola '**CORRESPONDING**' (o soltanto '**CORR**'), si copia il contenuto di una variabile strutturata in una o più variabili strutturate, abbinando però i campi aventi lo stesso nome. In pratica, con il secondo schema si vogliono copiare i campi della prima variabile strutturata che hanno gli stessi nomi di quelli contenuti nella seconda variabile strutturata. Diversamente, per una copia di una variabile strutturata in altre variabili, mantenendo inalterata la struttura originale dei dati, si usa il primo schema sintattico.

È bene ricordare che in alcuni casi la copia dei dati non può essere eseguita; per esempio non si può assegnare a una variabile numerica un'informazione alfanumerica (tenendo conto che una variabile numerica che contiene delle regole di modifica, all'atto della sua lettura offre un'informazione alfanumerica).

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   TEST-MOVE.
```

```
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-02-28.
000500
000600 ENVIRONMENT DIVISION.
000700
000800 DATA DIVISION.
000900
001000 WORKING-STORAGE SECTION.
001100 01  RECORD-1.
001200     02 A          PIC 999V99.
001300     02 B          PIC X(10).
001400     02 C          PIC 99999.
001500
001600 01  RECORD-2.
001700     02 C          PIC 9999999.
001800     02 B          PIC X(12).
001900     02 A          PIC 9999V999.
002000
002100 PROCEDURE DIVISION.
002200
002300 MAIN.
002400     MOVE 123.45       TO A OF RECORD-1.
002500     MOVE "ABCDEFGHIJ" TO B OF RECORD-1.
002600     MOVE 12345        TO C OF RECORD-1.
002700     DISPLAY "RECORD-1: ", RECORD-1.
002800     DISPLAY "    A:    ", A OF RECORD-1.
002900     DISPLAY "    B:    ", B OF RECORD-1.
003000     DISPLAY "    C:    ", C OF RECORD-1.
003100
003200     MOVE RECORD-1 TO RECORD-2.
003300     DISPLAY "RECORD-2: ", RECORD-2
003400     DISPLAY "    A:    ", A OF RECORD-2.
003500     DISPLAY "    B:    ", B OF RECORD-2.
003600     DISPLAY "    C:    ", C OF RECORD-2.
```

```
003700
003800     MOVE CORRESPONDING RECORD-1 TO RECORD-2.
003900     DISPLAY "RECORD-2: ", RECORD-2
004000     DISPLAY "    A:    ", A OF RECORD-2.
004100     DISPLAY "    B:    ", B OF RECORD-2.
004200     DISPLAY "    C:    ", C OF RECORD-2.
004300
004400     STOP RUN.
```

L'esempio mostra un programma in cui ci sono due variabili strutturate, contenenti campi, simili, con lo stesso nome, ordinati in modo differente. Dopo aver assegnato dei valori ai campi della prima variabile, il contenuto della variabile viene copiato nella seconda; successivamente, viene ripetuta la copia in modo corrispondente.

Se si compila il programma con OpenCOBOL e si avvia ciò che si ottiene, si dovrebbe vedere un risultato simile a quello seguente, dove si può notare la differenza tra un tipo di copia e l'altra:

```
RECORD-1: 12345ABCDEFGHIJ12345
    A:    123.45
    B:    ABCDEFGHIJ
    C:    12345
RECORD-2: 12345ABCDEFGHIJ12345
    A:    5   .000
    B:    CDEFGHIJ1234
    C:    12345A
RECORD-2: 0012345ABCDEFGHIJ  0123450
    A:    0123.450
    B:    ABCDEFGHIJ
    C:    0012345
```

Si osservi che una variabile di tipo '**INDEX**' non può essere usata con l'istruzione '**MOVE**'. Per assegnare un valore a una tale variabile occorre servirsi dell'istruzione '**SET**'.

72.12.13 Istruzione «MULTIPLY»

«

L'istruzione '**MULTIPLY**' consente di eseguire delle moltiplicazioni. Sono previsti due diversi formati per l'utilizzo di questa istruzione.

```
           /               \
           | identifier-1 |
MULTIPLY   <               >  BY  { identifier-2 [ROUNDED] }...
--------   | literal-1     |  --                  -------
           \               /

     [ ON SIZE ERROR  imperative-statement ]
          ---- -----
```

Nello schema sintattico appena mostrato, si vede che dopo la parola chiave '**MULTIPLY**' viene indicato un valore, in forma costante o attraverso una variabile; questo valore viene moltiplicato per la variabile indicata dopo la parola chiave '**BY**' e il risultato viene assegnato alla stessa variabile che funge da moltiplicatore. Se appaiono più variabili dopo la parola '**BY**', la moltiplicazione viene ripetuta per ognuna di quelle, assegnando rispettivamente il risultato.

L'opzione '**ROUNDED**' richiede di eseguire un arrotondamento se la variabile ricevente non può rappresentare in modo esatto il valore; l'opzione '**SIZE ERROR**' serve a eseguire un'istruzione nel caso una delle variabili riceventi non possa accogliere la porzione più significativa del valore ottenuto dalla somma. Si osservi l'esempio seguente:

```
000000     MULTIPLY 100 BY A.
```

Supponendo che la variabile '**A**', prima della divisione contenga il valore 5, dopo l'operazione contiene il valore 500 (100×5). Si potrebbe scrivere la stessa cosa utilizzando l'istruzione '**COMPUTE**':

```
000000     COMPUTE A = 100 * A.
```

Lo schema sintattico successivo mostra l'utilizzo di '**MULTIPLY**' in modo da non alterare i valori utilizzati come moltiplicatori:

```
          /               \        /               \
          | identifier-1 |        | identifier-2 |
MULTIPLY  <               >   BY   <               >
--------  | literal-1    |   --    | literal-2    |
          \               /        \               /

    GIVING identifier-3  [ROUNDED]
    ------                -------

    [ ON SIZE ERROR  imperative-statement ]
         ---- -----
```

Nella forma appena mostrata, la moltiplicazione avviene immettendo il risultato dell'operazione nella variabile indicata dopo la parola '**GIVING**'. Valgono le stesse considerazioni già fatte a proposito delle opzioni '**ROUNDED**' e '**SIZE ERROR**'. Si osservi l'esempio seguente che ripete sostanzialmente l'esempio già mostrato in precedenza:

```
000000     MULTIPLY 100 BY 5 GIVING A.
```

72.12.14 Istruzione «OPEN»

«

L'istruzione '**OPEN**' serve ad aprire un file, o un gruppo di file, specificando la modalità di accesso. Quando l'accesso a un file richiede l'esecuzione di alcune procedure meccaniche preliminari, questa istruzione serve a eseguirle. L'istruzione '**OPEN**' non riguarda i file dichiarati esplicitamente per il riordino e la fusione.

```
        /  INPUT   { file-name  [ WITH NO REWIND ] }...  \
        |  -----                        -- ------        |
        |  OUTPUT  { file-name  [ WITH NO REWIND ] }...  |
OPEN    <  ------                       -- ------        >...
----    |  I-O     { file-name }...                      |
        |  ---                                           |
        \  EXTEND  { file-name }...                      /
           ------
```

Dopo la parola chiave '**OPEN**' inizia l'elenco dei file che si vogliono aprire, cominciando con la parola chiave che definisce la modalità di accesso desiderata: '**INPUT**' richiede un accesso in lettura; '**OUTPUT**' un accesso in scrittura; '**I-O**' un accesso in lettura e scrittura; '**EXTEND**' un accesso in estensione (scrittura).

Il tipo di accesso consentito dipende dall'organizzazione dei file o dalla modalità di accesso; nelle versioni più vecchie del linguaggio, l'apertura in estensione ('**EXTEND**') può essere usata soltanto per i file sequenziali; l'apertura in lettura e scrittura ('**I-O**') richiede che il file sia collocato in un'unità di memorizzazione ad accesso diretto, come nel caso dei dischi.

L'opzione '**NO REWIND**' si riferisce al riavvolgimento automatico del nastro, che riguarda, evidentemente, solo file sequenziali su unità ad accesso sequenziale, che possono richiedere un'operazione di

riavvolgimento. Se si usa questa opzione, si intende evitare che il nastro venga riavvolto automaticamente alla chiusura del file stesso. Per i file su disco, o comunque su unità ad accesso diretto, anche se si tratta di file con organizzazione sequenziale, questa opzione non deve essere usata.

Quando un file viene aperto (con questa istruzione) è possibile accedervi secondo la modalità prevista, con le istruzioni appropriate. L'apertura va eseguita una sola volta e la chiusura (con l'istruzione '**CLOSE**') dichiara la conclusione delle operazioni con quel file. Se un file deve essere riaperto all'interno del programma, probabilmente perché vi si vuole accedere secondo una modalità differente, o per altri motivi, è necessario che alla chiusura non sia utilizzata l'opzione '**lock**', che altrimenti impedirebbe di farlo.

L'apertura in lettura che si ottiene con la parola chiave '**READ**' serve ad accedere a un file esistente in modo da poter leggere il suo contenuto; l'apertura fa sì che la posizione relativa, iniziale, all'interno del file, corrisponda al primo record logico. Se il file non esiste, si presenta una condizione di errore.

L'apertura in scrittura che si ottiene con la parola chiave '**OUTPUT**' serve a **creare** un file, ma se il file esiste già, questo viene azzerato completamente.

L'apertura in lettura e scrittura che si ottiene con la parola chiave '**I-O**' serve a permettere l'accesso a un file esistente, sia per leggere i dati, sia per modificarli. La posizione relativa iniziale è quella del primo record logico.

L'apertura in estensione che si ottiene con la parola chiave '**EXTEND**', può essere utilizzata soltanto con file sequenziali e serve

a consentire l'aggiunta di record a partire dalla fine del file iniziale. Pertanto, il puntatore relativo iniziale si trova dopo la fine dell'ultimo record logico e l'utilizzo di questo file avviene nello stesso modo di un'apertura in scrittura, con la differenza che il contenuto precedente non viene cancellato.

Se il file che viene aperto è associato a una variabile indicata con l'opzione '**FILE STATUS**' nell'istruzione '**SELECT**' (nella sezione '**FILE-CONTROL**' di '**ENVIRONMENT DIVISION**'), il valore di tale variabile viene aggiornato.

Tabella 72.199. File con organizzazione sequenziale (e accesso sequenziale).

Istruzione	Apertura in lettura ('**INPUT**')	Apertura in scrittura ('**OUTPUT**')	Apertura in lettura e scrittura ('**I-O**')	Apertura in estensione ('**EXTEND**')
`READ`	X		X	
`WRITE`		X		X
`REWRITE`			X	

Tabella 72.200. File con organizzazione relativa o a indice, con accesso sequenziale.

Istruzione	Apertura in lettura (‘**INPUT**’)	Apertura in scrittura (‘**OUTPUT**’)	Apertura in lettura e scrittura (‘**I-O**’)
READ	X		X
WRITE		X	
REWRITE			X
START	X		X
DELETE			X

Tabella 72.201. File con organizzazione relativa o a indice, con accesso diretto (*random*).

Istruzione	Apertura in lettura (‘**INPUT**’)	Apertura in scrittura (‘**OUTPUT**’)	Apertura in lettura e scrittura (‘**I-O**’)
READ	X		X
WRITE		X	X
REWRITE			X
START			
DELETE			X

Tabella 72.202. File con organizzazione relativa o a indice, con accesso dinamico.

Istruzione	Apertura in lettura (‘**INPUT**’)	Apertura in scrittura (‘**OUTPUT**’)	Apertura in lettura e scrittura (‘**I-O**’)
READ	X		X
WRITE		X	X
REWRITE			X
START	X		X
DELETE			X

72.12.15 Istruzione «PERFORM»

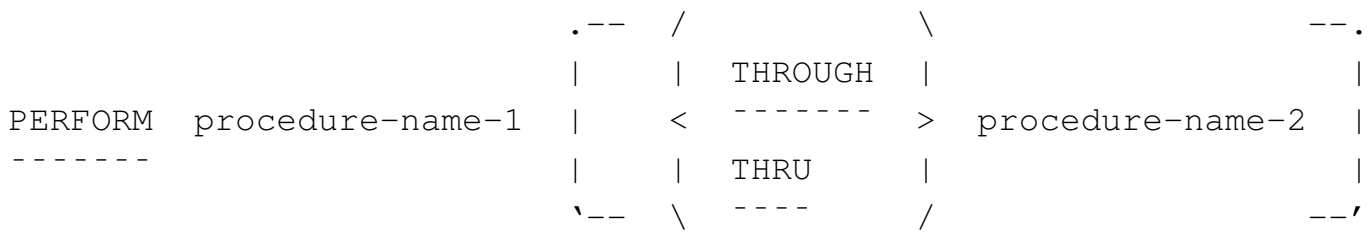
«

L’istruzione ‘**PERFORM**’ consente di eseguire un gruppo di istruzioni, contenute all’interno di sezioni o di paragrafi della divisione ‘**PROCEDURE DIVISION**’, riprendendo poi il funzionamento nell’istruzione successiva.

Sono disponibili schemi sintattici diversi, perché la chiamata di queste procedure può essere gestita in maniere differenti. In effetti, questa istruzione è il mezzo con cui realizzare delle iterazioni, normali e con enumerazione, pertanto si rende necessaria questa flessibilità da parte dell’istruzione ‘**PERFORM**’.

Nelle sezioni successive vengono descritte le varie forme di utilizzo dell’istruzione ‘**PERFORM**’, per livelli successivi di complessità. Si tenga conto che la spiegazione riguardo al funzionamento per un certo livello, riguarda anche quelli più complessi successivi.

72.12.15.1 Chiamata semplice

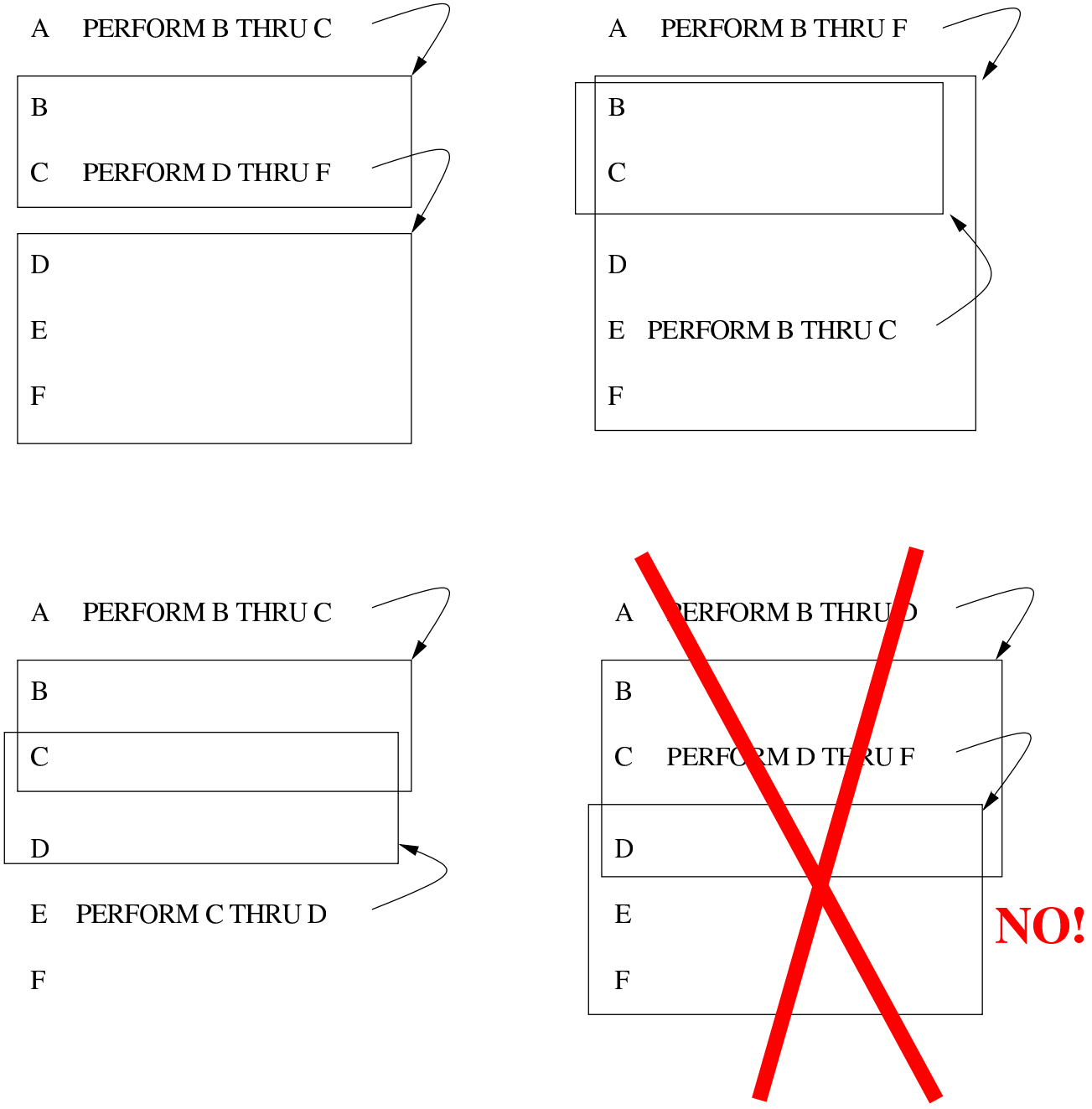


Secondo la forma di utilizzo più semplice dell'istruzione '**PERFORM**', la chiamata esegue una volta sola l'intervallo di procedure indicate. Per procedure qui si intendono dei paragrafi, oppure delle sezioni intere della divisione '**PROCEDURE DIVISION**'.

Se si indica soltanto un nome (di paragrafo o di sezione), si intende eseguire solo la procedura relativa; se si indica la parola '**THROUGH**' o '**THRU**' seguita da un altro nome, si intendono eseguire tutti i paragrafi o tutte le sezioni dal primo al secondo nome incluso.

Il fatto che la chiamata di una procedura avvenga in modo così libero, implica la necessità di stabilire delle restrizioni alle chiamate annidate: una procedura, o un insieme di procedure chiamate attraverso l'istruzione '**PERFORM**', possono contenere delle chiamate annidate. Queste chiamate interne, per poter essere eseguite correttamente, devono riguardare delle procedure più interne, oppure completamente esterne.

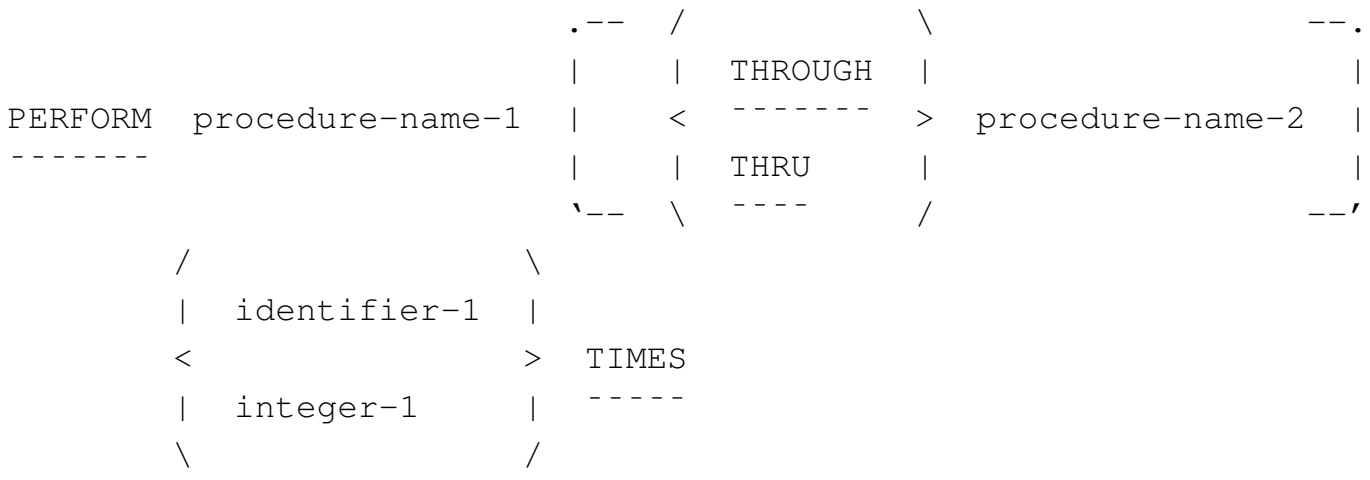
Figura 72.204. Schematizzazione delle forme di annidamento consentite e di quella non consentita (sbarrata).



La figura mostra schematicamente i vari modi in cui le istruzioni '**PERFORM**' possono annidarsi, o possono in qualche modo riguardare le stesse porzioni di codice. L'ultimo esempio, in basso a destra, non è ammissibile perché la chiamata dei paragrafi da '**D**' a '**F**' verrebbe interrotta alla conclusione del paragrafo '**D**', con il rientro dalla

prima istruzione '**PERFORM**'.

72.12.15.2 Chiamata ripetuta un certo numero di volte

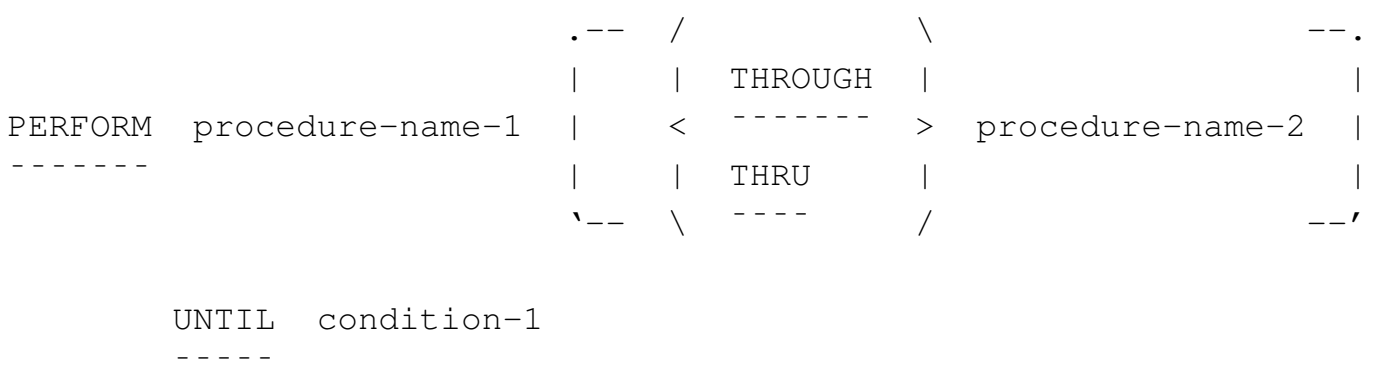


Aggiungendo allo schema già visto un numero intero, espresso sia in forma costante, sia attraverso una variabile, seguito dalla parola '**TIMES**', si intende ottenere a ripetizione della chiamata del gruppo di procedure indicato per quella quantità di volte.

Se il valore numerico indicato è pari a zero, oppure si tratta di un numero negativo, la chiamata delle procedure viene ignorata semplicemente.

72.12.15.3 Chiamata ripetuta con condizione di uscita

«

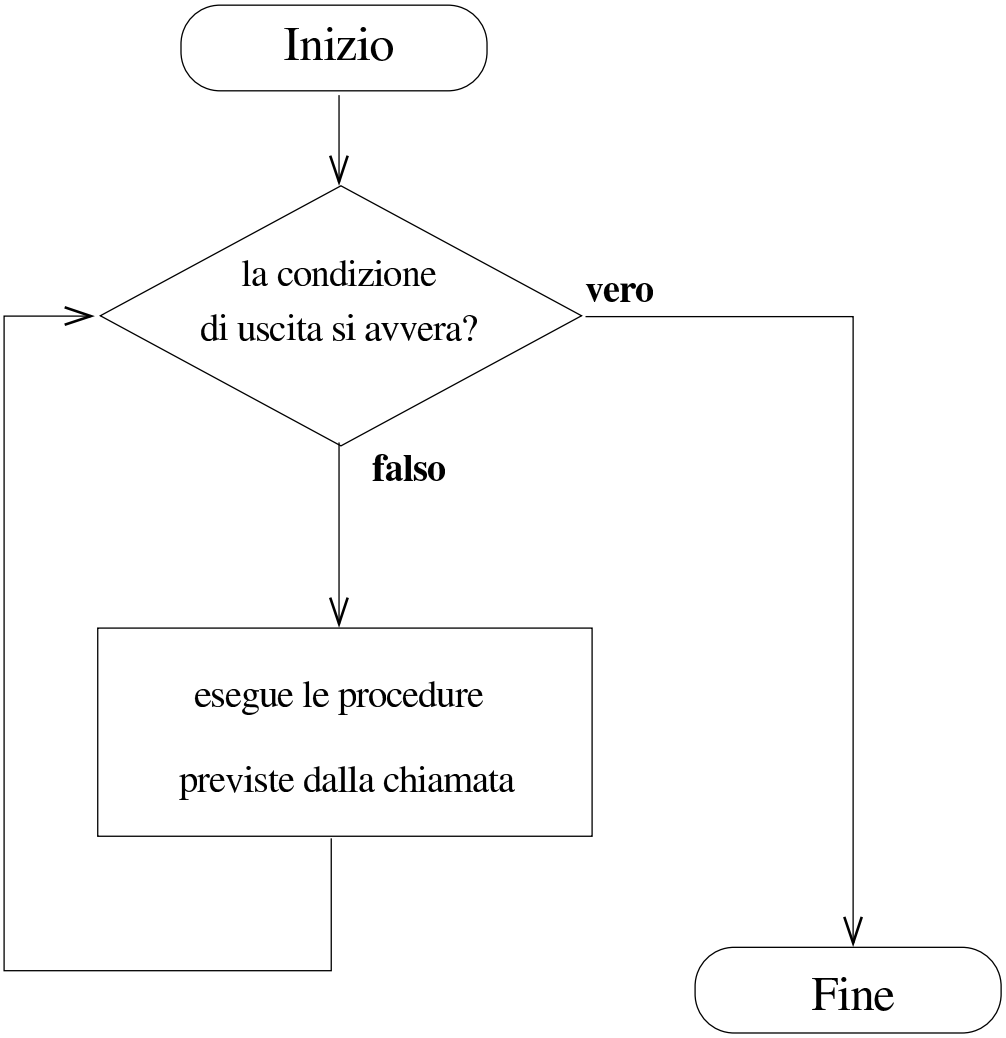


Quando nell'istruzione '**PERFORM**' compare la parola chiave '**UNTIL**', seguita da una condizione, si intende eseguire il gruppo di procedure indicate ripetutamente, fino a quando la condizione specificata restituisce il valore *Falso*.

La condizione di uscita viene verificata prima di eseguire ogni iterazione, pertanto, se risulta *Vero* all'inizio, le procedure non vengono eseguite.

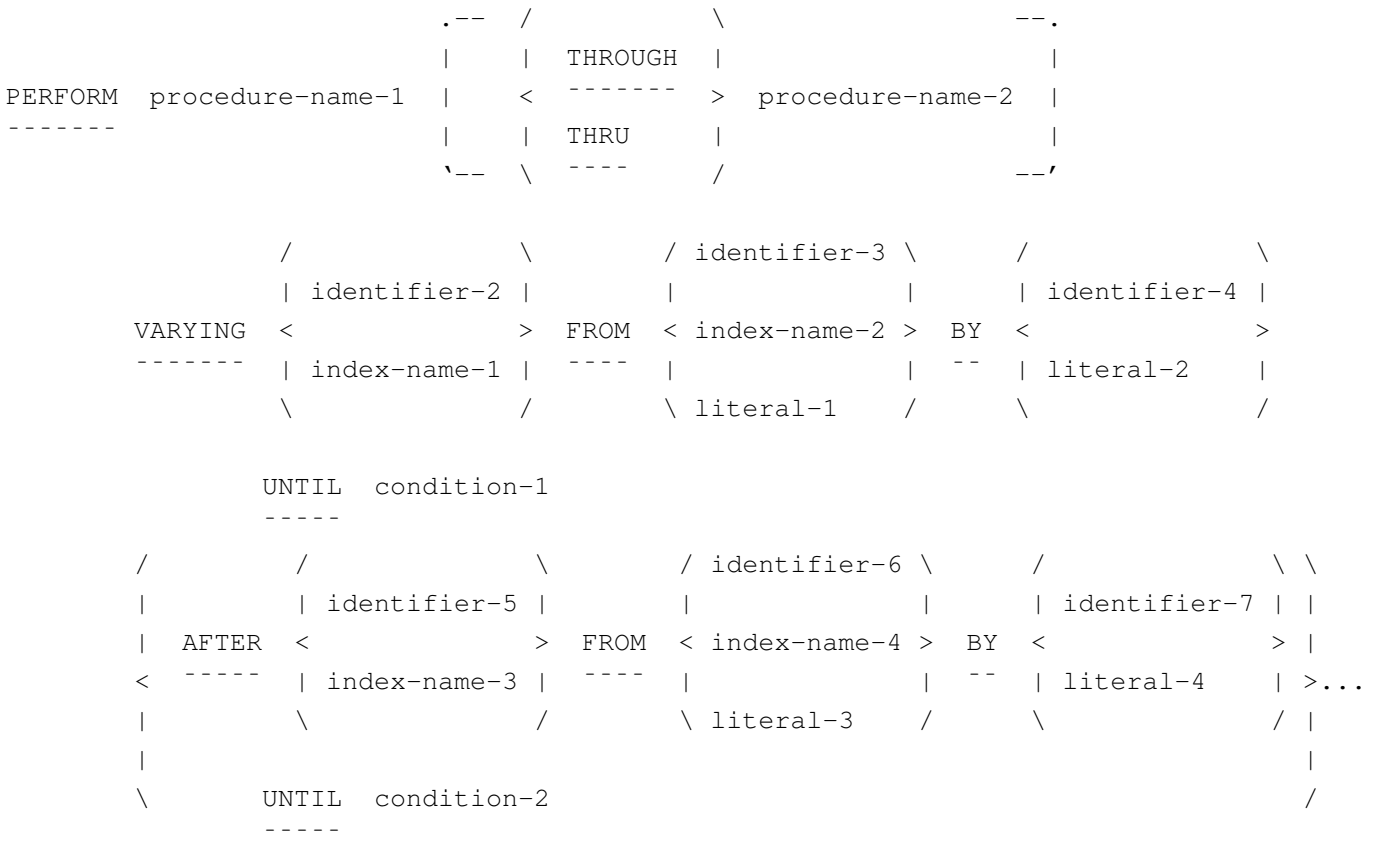
Rispetto ai linguaggi di programmazione comuni, il COBOL attribuisce alla parola '**UNTIL**' un significato opposto, anche se logico: «si esegue il ciclo fino a quanto si verifica la condizione». Il problema è che nel senso comune ciò significa che il ciclo va ripetuto in quanto la condizione continua ad avverarsi, mentre secondo il senso del COBOL il ciclo va ripetuto fino a quando si verifica la condizione di uscita, nel senso che il verificarsi della condizione di uscita fa terminare il ciclo.

Figura 72.207. Diagramma di flusso dell'istruzione '**PERFORM**' iterativa con una condizione di uscita.



72.12.15.4 Chiamata ripetuta con condizione di uscita e incremento di contatori

«



Con l'aggiunta della parola chiave '**VARYING**', si intende gestire un contatore numerico (rappresentato nello schema da ***identifier-2*** o da ***index-name-1***, che pertanto può essere una variabile numerica o un indice di una tabella), specificando il valore di partenza dopo la parola '**FROM**', l'incremento a ogni ciclo dopo la parola '**BY**' e la condizione di uscita dopo la parola '**UNTIL**'.

Possono essere gestiti più contatori, con un limite che dipende dal compilatore. A ogni modo, per aggiungere un contatore si usa la parola '**AFTER**', che ne introduce la descrizione, così come per la parola '**VARYING**'.

Il contatore che viene incrementato a ogni ciclo, è quello più interno, ovvero quello descritto dall'ultima parola '**AFTER**'. Quando per quel contatore si verifica la condizione di uscita, viene incrementato il contatore del livello precedente (la penultima parola '**AFTER**'

o direttamente '**VARYING**' in mancanza di quella) e azzerato quello interno.

Il ciclo termina quando sono scattate tutte le condizioni di uscita dei vari contatori.

Il linguaggio non pone vincoli alla gestione dei contatori indicati nell'istruzione '**PERFORM**', che possono essere alterati durante l'esecuzione delle procedure chiamate dall'istruzione stessa e in qualche modo possono contaminarsi tra di loro. Sta evidentemente al programmatore evitare di creare confusione nel programma, osservando anche che la sequenza esatta delle operazioni di incremento e azzeramento dei contatori cambia leggermente da uno standard all'altro del linguaggio.

Figura 72.209. Diagramma di flusso dell'istruzione '**PERFORM**' iterativa con l'incremento di un solo contatore.

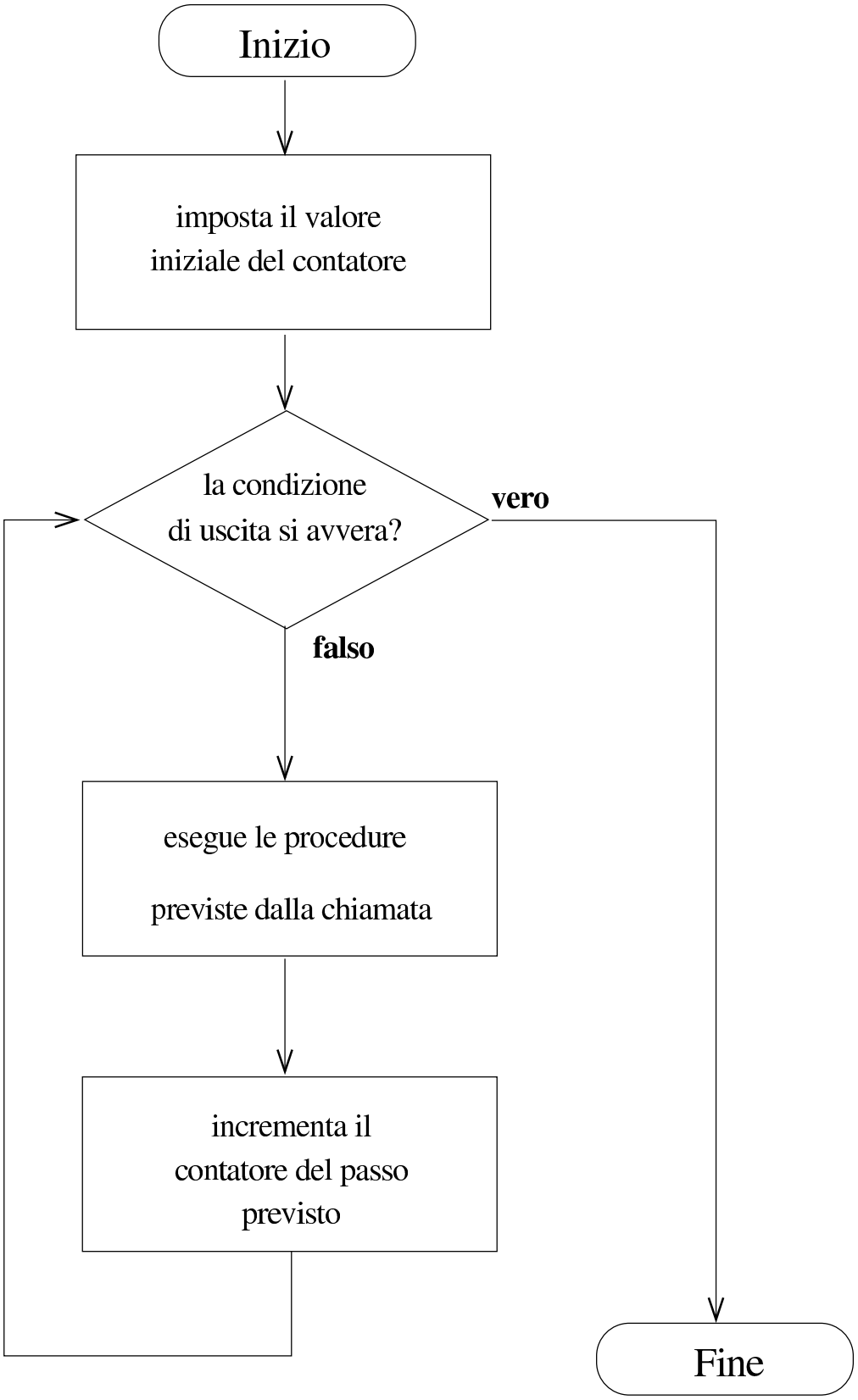
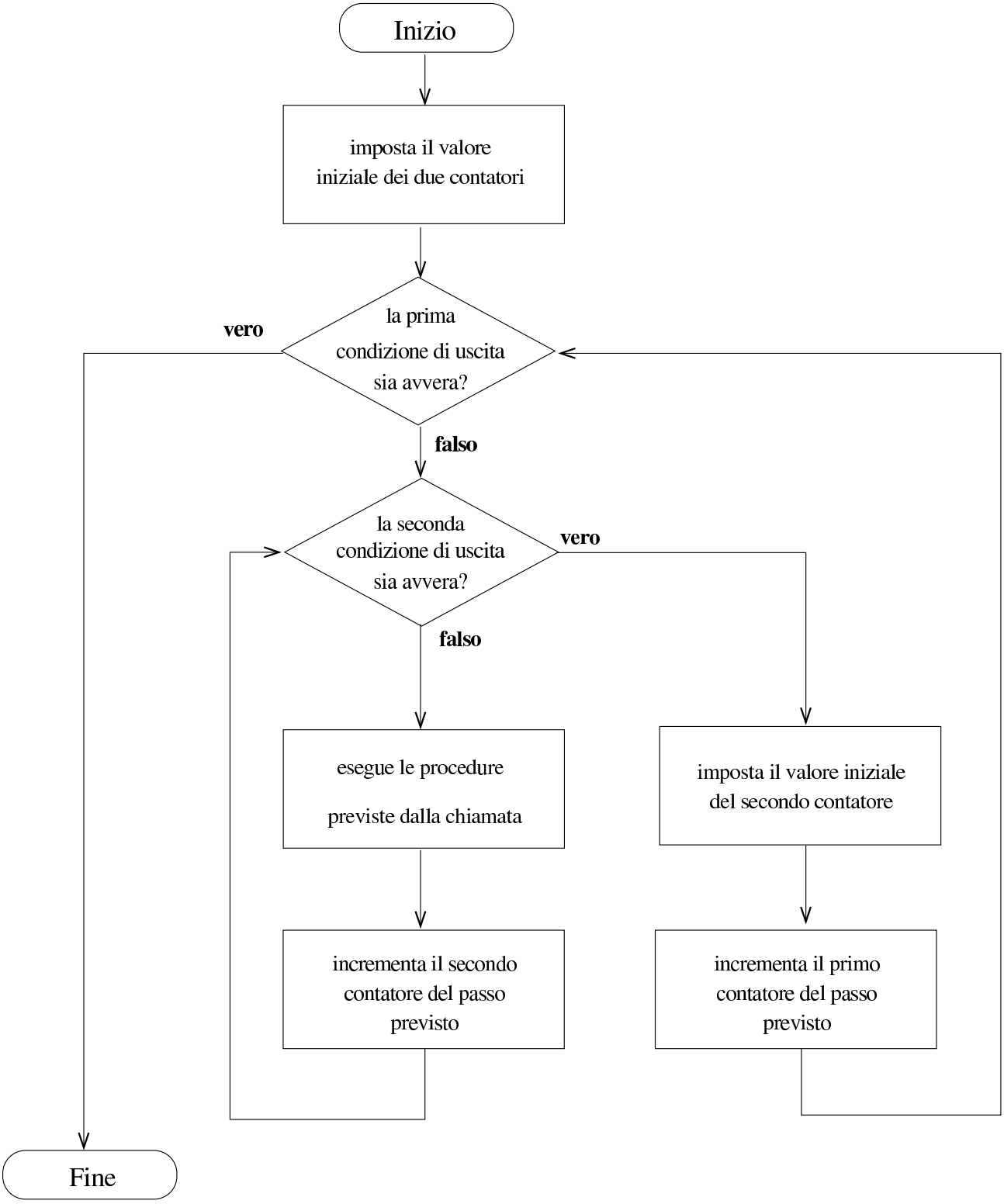


Figura 72.210. Diagramma di flusso dell'istruzione '**PERFORM**' iterativa con la gestione di due contatori.



L'esempio seguente mostra in modo molto semplice la gestione di

tre contatori, che scandiscono valori interi da zero a due, senza fare nulla altro di particolare.

Listato 72.211. Programma chiama un paragrafo incrementando tre contatori.

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   TEST-PERFORM.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 INSTALLATION. NANOLINUX IV,
000500               OPENCOBOL 0.31,
000600 DATE-WRITTEN. 2005-03-17.
000700*
000800 ENVIRONMENT DIVISION.
000900*
001000 DATA DIVISION.
001100*
001200 WORKING-STORAGE SECTION.
001300 77  CONTATORE-1          PIC 99.
001400 77  CONTATORE-2          PIC 99.
001500 77  CONTATORE-3          PIC 99.
001600*
001700 PROCEDURE DIVISION.
001800*------------------------ LIVELLO 0 -----------------
001900 MAIN.
002000     PERFORM VISUALIZZA-CONTATORI
002100             VARYING CONTATORE-1 FROM 0 BY 1
002200                     UNTIL CONTATORE-1 >= 2,
002300             AFTER   CONTATORE-2 FROM 0 BY 1
002400                     UNTIL CONTATORE-2 >= 2,
002500             AFTER   CONTATORE-3 FROM 0 BY 1
002600                     UNTIL CONTATORE-3 >= 2.
002700*
002800     STOP RUN.
002900*------------------------ LIVELLO 1 ----------------
003000 VISUALIZZA-CONTATORI.
```

```
003100     DISPLAY CONTATORE-1, " ", CONTATORE-2, " ",
003150             CONTATORE-3.
003200*
```

Una volta compilato questo programma, avviando ciò che si ottiene, si può vedere il risultato seguente:

```
00 00 00
00 00 01
00 01 00
00 01 01
01 00 00
01 00 01
01 01 00
01 01 01
```

72.12.16 Istruzione «READ»

L'istruzione '**READ**' serve a ottenere un record logico da un file, che risulta essere già stato aperto, in modo tale da consentire la lettura ('**INPUT**' o '**I-O**'). Sono disponibili formati diversi per l'utilizzo di questa istruzione, che dipendono dall'organizzazione del file a cui si accede.

Organizzazione sequenziale, relativa e a indice

```
READ file-name [NEXT] RECORD  [ INTO identifier ]
----           ----           ----

    [ AT END  { imperative-statement }... ]
      ---
```

Organizzazione relativa

```
READ file-name RECORD  [ INTO identifier ]
----                     ----

    [ INVALID KEY { imperative-statement }... ]
      -------
```

Organizzazione a indice

```
READ file-name RECORD  [ INTO identifier ]
----                     ----

    [ KEY IS  data-name ]
      ---

    [ INVALID KEY { imperative-statement }... ]
      -------
```

In tutti gli schemi sintattici che riguardano l'istruzione '**READ**', si può vedere che viene indicato immediatamente il nome del file (già aperto) che si vuole leggere. Successivamente, appare una parola chiave opzionale, '**INTO**', che precede il nome di una variabile; se viene specificata questa informazione, si intende fare in modo che il record logico ottenuto dal file, oltre che essere disponibile nella variabile strutturata dichiarata appositamente per questo, dopo l'indicatore di livello '**FD**' relativo, sia anche copiato in un'altra variabile. Inoltre, le istruzioni imperative (***imperative-statement***) che si possono inserire dopo le parole '**AT END**' e '**INVALID KEY**', servono a dichiarare cosa deve fare il programma nel caso la lettura fallisca per qualche motivo.

Se il file che viene letto è associato a una variabile indicata con l'opzione '**FILE STATUS**' nell'istruzione '**SELECT**' (nella sezione '**FILE-CONTROL**' di '**ENVIRONMENT DIVISION**'), il valore di tale variabile viene aggiornato.

Nel caso di un file a cui si accede sequenzialmente, si applica il primo schema sintattico. In questo caso l'istruzione '**READ**' fornisce il record attuale e sposta in avanti il puntatore al record, in modo che una lettura successiva fornisca il prossimo record. Quando l'accesso è dinamico e si vuole leggere un file in modo sequenziale, occorre aggiungere l'opzione '**NEXT**', per richiedere espressamente l'avanzamento al record successivo.

Quando si accede sequenzialmente, oppure in modo dinamico ma specificando che si richiede il record successivo, si può verificare un errore che consiste nel tentativo di leggere oltre la fine del file. Se ciò accade e se è stata specificata l'opzione '**AT END**', vengono eseguite le istruzioni che seguono tali parole.

La lettura sequenziale di un file relativo, comporta l'aggiornamento del valore della «chiave relativa», ovvero di quanto specificato con la dichiarazione '**RELATIVE KEY**' dell'istruzione '**SELECT**'.

La lettura sequenziale può essere applicata anche a un file organizzato a indice; in tal caso, la sequenza di lettura corrisponde a quella della chiave principale.

Quando si accede in modo diretto ai record all'interno di un file relativo, si utilizza il secondo schema sintattico, per ottenere il record specificato dal numero contenuto nella variabile che funge da chiave (come specificato nell'istruzione '**SELECT**', attraverso la dichiarazione '**RELATIVE KEY**'). Se un record con quel numero non esiste, si verifica la condizione controllata dall'opzione '**INVALID KEY**' e il programma esegue le istruzioni che questa controlla.

Il terzo formato sintattico si usa per i file organizzati a indice, con accesso diretto, in base alla chiave specificata. La chiave in questione

è quella primaria, salvo specificarla nell'istruzione '**READ**' con l'opzione '**KEY IS**'. La chiave cercata deve essere scritta in corrispondenza del campo che la contiene, all'interno del record dichiarato dopo l'indicatore di livello '**FD**' relativo al file, secondo le specifiche dell'istruzione '**SELECT**' ('**RECORD KEY**', o '**ALTERNATE RECORD KEY**'). Se la lettura avviene con successo, si ottiene il record che contiene quella chiave; altrimenti si verifica la condizione controllata dall'opzione '**INVALID KEY**' e le istruzioni relative vengono eseguite.

La lettura ad accesso diretto di un file a indice, consente di ottenere il primo record che soddisfa la corrispondenza con la chiave cercata; se sono presenti record con chiavi doppie, le altre corrispondenze devono essere raggiunte attraverso una lettura sequenziale.

Listato 72.216. Programma elementare che legge un file sequenziale, ad accesso sequenziale.

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   TEST-READ-SEQ.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 INSTALLATION. NANOLINUX IV,
000500               TINYCOBOL 0.61,
000600               OPENCOBOL 0.31.
000700 DATE-WRITTEN. 2005-03-12.
000800*
000900 ENVIRONMENT DIVISION.
001000*
001100 INPUT-OUTPUT SECTION.
001200*
001300 FILE-CONTROL.
001400*
001500     SELECT FILE-DA-LEGGERE ASSIGN TO "input.seq"
001600                            ORGANIZATION IS SEQUENTIAL.
```

```
001700*
001800 DATA DIVISION.
001900*
002000 FILE SECTION.
002100*
002200 FD  FILE-DA-LEGGERE
002300     LABEL RECORD IS STANDARD.
002400*
002500 01  RECORD-DA-LEGGERE PIC X(30).
002600*
002700 WORKING-STORAGE SECTION.
002800 01  EOF               PIC 9   VALUE ZERO.
002900*
003000 PROCEDURE DIVISION.
003100*------------------------ LIVELLO 0 -----------------
003200 MAIN.
003300     OPEN INPUT FILE-DA-LEGGERE.
003400     READ FILE-DA-LEGGERE
003500          AT END
003600               MOVE 1 TO EOF.
003700     PERFORM LETTURA UNTIL EOF = 1.
003800     CLOSE FILE-DA-LEGGERE.
003900*
004000     STOP RUN.
004100*------------------------ LIVELLO 1 -----------------
004200 LETTURA.
004300     DISPLAY RECORD-DA-LEGGERE.
004400     READ FILE-DA-LEGGERE
004500          AT END
004600               MOVE 1 TO EOF.
004700*
```

Listato 72.217. Programma elementare che legge un file sequenziale, ad accesso dinamico. Le differenze rispetto all'esempio precedente sono evidenziate.

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   TEST-READ-DYN.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 INSTALLATION. NANOLINUX IV,
000500               TINYCOBOL 0.61,
000600               OPENCOBOL 0.31.
000700 DATE-WRITTEN. 2005-03-12.
000800*
000900 ENVIRONMENT DIVISION.
001000*
001100 INPUT-OUTPUT SECTION.
001200*
001300 FILE-CONTROL.
001400*
001500     SELECT FILE-DA-LEGGERE ASSIGN TO "input.seq"
001600                            ORGANIZATION IS SEQUENTIAL
001700                            ACCESS MODE IS DYNAMIC.
001800*
001900 DATA DIVISION.
002000*
002100 FILE SECTION.
002200*
002300 FD  FILE-DA-LEGGERE
002400     LABEL RECORD IS STANDARD.
002500*
002600 01  RECORD-DA-LEGGERE PIC X(30).
002700*
002800 WORKING-STORAGE SECTION.
002900 01  EOF                PIC 9   VALUE ZERO.
003000*
```

```
003100 PROCEDURE DIVISION.
003200*------------------------ LIVELLO 0 -----------------
003300 MAIN.
003400     OPEN INPUT FILE-DA-LEGGERE.
003500     READ FILE-DA-LEGGERE
003600          AT END
003700               MOVE 1 TO EOF.
003800     PERFORM LETTURA UNTIL EOF = 1.
003900     CLOSE FILE-DA-LEGGERE.
004000*
004100     STOP RUN.
004200*------------------------ LIVELLO 1 -----------------
004300 LETTURA.
004400     DISPLAY RECORD-DA-LEGGERE.
004500     READ FILE-DA-LEGGERE NEXT RECORD
004600          AT END
004700               MOVE 1 TO EOF.
004800*
```

72.12.17 Istruzione «REWRITE»

L'istruzione '**REWRITE**' consente si sovrascrivere un record logico all'interno di un file, purché questo risieda all'interno di un'unità che consente un accesso diretto ai dati (le unità sequenziali come i nastri sono escluse). Per utilizzare l'istruzione '**REWRITE**' il file deve essere stato aperto in lettura e scrittura ('**I-O**'); inoltre, il record deve avere una dimensione fissa.

File organizzati in modo sequenziale

```
REWRITE record-name  [ FROM identifier ]
-------                ----
```

File organizzati in modo da consentire un accesso diretto

```
REWRITE record-name  [ FROM identifier ]
-------              ----

    [ INVALID KEY { imperative-statement }... ]
      -------
```

Gli schemi sintattici mostrati hanno in comune la prima parte: il nome della variabile che fa riferimento al record, serve a individuare implicitamente il file a cui si fa riferimento; la variabile indicata dopo la parola '**FROM**', permette di copiare tale variabile su quella del record, prima di procedere alla sovrascrittura, come se si usasse l'istruzione '**MOVE**' prima di '**REWRITE**':

```
MOVE identifier TO record-name;
----

REWRITE record-name
-------

    [ INVALID KEY { imperative-statement }... ]
      -------
```

Quando si utilizza l'istruzione '**REWRITE**' con un file aperto in modo sequenziale, prima è necessario che sia stata eseguita una lettura del record che si vuole sovrascrivere; la lettura implica la selezione del record. Nel caso particolare di un accesso sequenziale a un file con indice, oltre che leggere preventivamente il record da sovrascrivere, occorre accertarsi che la riscrittura mantenga la stessa chiave, altrimenti la riscrittura non avviene e si attiva invece l'opzione '**INVALID KEY**' (con l'esecuzione delle istruzioni che questa controlla). Oltre a questo, se il file prevede l'esistenza di una chiave secondaria e non sono ammesse chiavi doppie, se il record da sovrascrivere contiene una chiave secondaria già esistente in un altro, si ottiene, anche in questo caso, l'attivazione dell'opzione '**INVALID KEY**'.

Quando l'istruzione '**REWRITE**' si applica a file aperti attraverso un accesso diretto, dinamico o con chiave, la sovrascrittura non richiede più di procedere prima a una lettura del record, perché è sufficiente indicarlo tramite il numero ('**RELATIVE KEY**') oppure attraverso la chiave primaria. In tal caso, la condizione '**INVALID KEY**' si verifica quando il numero del record o la chiave primaria non corrispondono a nulla di già esistente nel file. Nel caso particolare dei file con indice, la condizione '**INVALID KEY**' si avvera anche quando, non essendo previste chiavi doppie, si tenta di modificare un record, immettendo però una chiave secondaria (non quella primaria) già esistente in un altro.

Listato 72.221. Programma elementare che legge un file sequenziale, ad accesso sequenziale, che quando incontra un record contenente lettere «A», lo sostituisce con lettere «Z».

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   TEST-READ-SEQ-REWRITE.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 INSTALLATION. NANOLINUX IV,
000500               TINYCOBOL 0.61,
000600               OPENCOBOL 0.31.
000700 DATE-WRITTEN. 2005-03-12.
000800*
000900 ENVIRONMENT DIVISION.
001000*
001100 INPUT-OUTPUT SECTION.
001200*
001300 FILE-CONTROL.
001400*
001500     SELECT FILE-DA-MODIFICARE ASSIGN TO "input.seq"
001600            ORGANIZATION IS SEQUENTIAL.
001700*
001800 DATA DIVISION.
```

```
001900*
002000 FILE SECTION.
002100*
002200 FD  FILE-DA-MODIFICARE
002300     LABEL RECORD IS STANDARD.
002400*
002500 01  RECORD-DA-MODIFICARE PIC X(30).
002600*
002700 WORKING-STORAGE SECTION.
002800 01  EOF               PIC 9   VALUE ZERO.
002900*
003000 PROCEDURE DIVISION.
003100*------------------------ LIVELLO 0 -----------------
003200 MAIN.
003300     OPEN I-O FILE-DA-MODIFICARE.
003400     READ FILE-DA-MODIFICARE
003500          AT END
003600               MOVE 1 TO EOF.
003700     PERFORM LETTURA-RISCRITTURA UNTIL EOF = 1.
003800     CLOSE FILE-DA-MODIFICARE.
003900*
004000     STOP RUN.
004100*------------------------ LIVELLO 1 -----------------
004200 LETTURA-RISCRITTURA.
004300     IF RECORD-DA-MODIFICARE
004350         = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
004400     THEN
004500         MOVE "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ"
004600              TO RECORD-DA-MODIFICARE,
004700         REWRITE RECORD-DA-MODIFICARE.
004800     READ FILE-DA-MODIFICARE
004900          AT END
005000               MOVE 1 TO EOF.
005100*
```

72.12.18 Istruzione «SEARCH»

L'istruzione '**SEARCH**' scandisce una tabella alla ricerca di un elemento che soddisfi una condizione, più o meno articolata, posizionando l'indice della tabella stessa in corrispondenza dell'elemento trovato. Sono disponibili due schemi sintattici: il primo serve per scandire le tabelle in modo sequenziale; il secondo serve per scandire delle tabelle ordinate, attraverso una ricerca binaria.

Ricerca sequenziale

```
                        .--          /                  \ --.
                        |            |  identifier-2    |   |
SEARCH  identifier-1    |  VARYING   <                  >   |
------                  |  -------   |  index-name-1    |   |
                        `--          \                  / --'
    [ AT END  { imperative-statement-1 }... ]
         ---

    /                     /                                  \  \
    |                     |  { imperative-statement-2 }...   |  |
    <  WHEN  condition-1  <                                  >  >...
    |  ----               |  NEXT SENTENCE                   |  |
    \                     \  ---- --------                   /  /
```

Ricerca binaria per tabelle ordinate

```
SEARCH ALL  identifier-1  [ AT END  { imperative-statement-1 }... ]
------ ---                     ---

                        /                                  \
                        |  { imperative-statement-2 }...   |
    WHEN  condition-1    <                                  >
    ----                |  NEXT SENTENCE                   |
                        \  ---- --------                   /
```

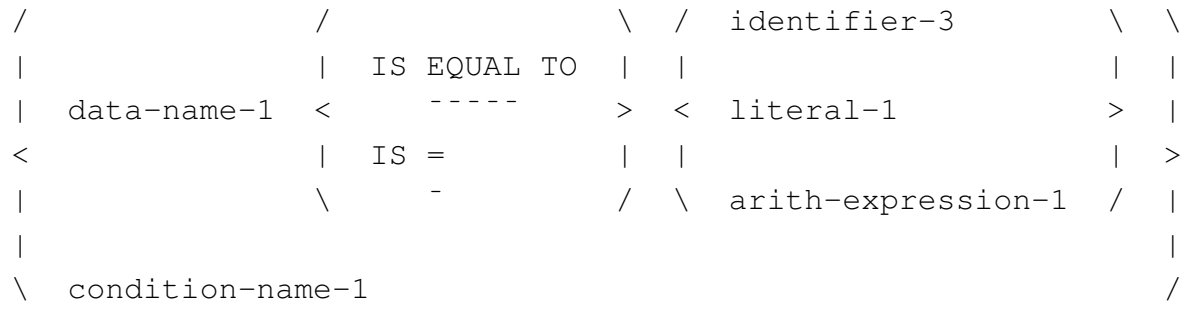
In entrambi i formati di utilizzo dell'istruzione '**SEARCH**', la variabile indicata come ***identifier-1*** deve essere stata dichiarata con l'opzione '**OCCURS**' e con l'opzione '**INDEXED BY**' (pertanto è obbligatorio

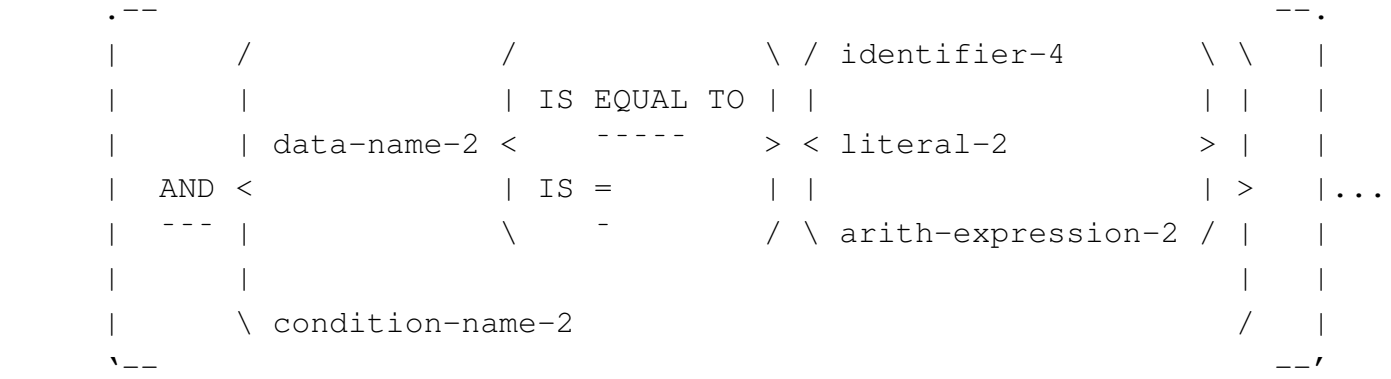
che gli sia stato attribuito un indice in modo esplicito). Nel caso del secondo formato, che si utilizza per una ricerca binaria, è obbligatorio che la variabile indicata come ***identifier-1*** sia stata dichiarata con l'opzione '**KEY IS**', che sta a specificare il fatto che la tabella è ordinata in base a una certa chiave.

L'opzione '**AT END**' di entrambi gli schemi sintattici precede una o più istruzioni da eseguire nel caso la ricerca fallisca.

La parola chiave '**WHEN**' precede una condizione, che deve essere soddisfatta per lo scopo della ricerca, dopo la quale vengono eseguite le istruzioni successive (***imperative-statement-2***). Quando la scansione avviene in modo sequenziale, secondo il primo formato, la condizione può essere espressa in modo abbastanza libero, inoltre si possono indicare condizioni differenti e gruppi diversi di istruzioni da eseguire; quando invece la ricerca avviene in modo ordinato (ricerca binaria), ci può essere una sola condizione, che verifichi la corrispondenza della chiave con il valore cercato (se ci sono chiavi secondarie, si combinano le condizioni con l'operatore '**AND**').

La condizione di una ricerca in una tabella ordinata (ricerca binaria) deve rispettare i limiti dello schema sintattico seguente, dove le metavariabili ***data-name*** sono le chiavi di ordinamento, che vanno indicate con gli indici necessari:





72.12.18.1 Ricerca sequenziale

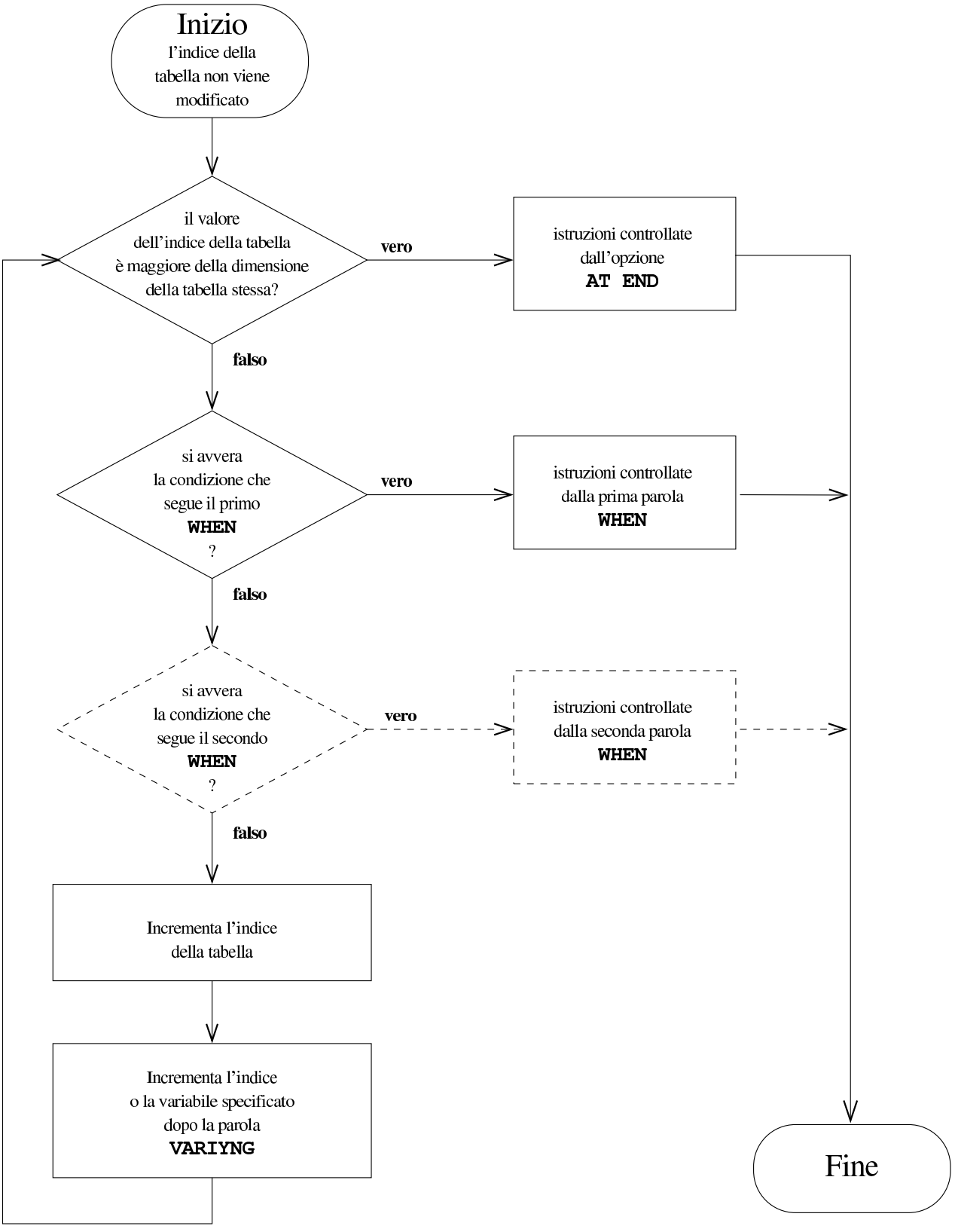
La ricerca sequenziale con l'istruzione '**SEARCH**', inizia dal valore che si trova già ad avere l'indice, proseguendo fino a soddisfare una delle condizioni, oppure fino alla fine degli elementi. Pertanto, se l'indice dovesse avere un valore maggiore del numero degli elementi della tabella, l'istruzione terminerebbe immediatamente.

L'istruzione '**SEARCH**', usata per una ricerca sequenziale, esegue un ciclo di verifiche delle condizioni poste, quindi incrementa l'indice della tabella e ricomincia i confronti, fino a quando si avvera una delle condizioni, oppure quando la tabella non ha più elementi. Oltre a incrementare l'indice della tabella, può incrementare un altro indice, di un'altra tabella, o semplicemente una variabile numerica, attraverso l'uso dell'opzione '**VARYING**'.

Tradizionalmente, il funzionamento dell'istruzione '**SEARCH**', quan-

do si usa per una scansione sequenziale di una tabella, lo si descrive attraverso un diagramma di flusso, nel quale si immagina di utilizzare due condizioni controllate dalla parola '**WHEN**', come si vede nella figura 72.225.

Figura 72.225. Esecuzione dell'istruzione '**SEARCH**' secondo il formato per la scansione sequenziale. Si mette in evidenza l'uso di due parole '**WHEN**' e si può comprendere come sarebbe con l'aggiunta di altre condizioni del genere.



Viene mostrato l'esempio di un programma completo che inizia con l'inserimento di dati all'interno di una tabella, quindi esegue una ricerca sequenziale al suo interno:

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   TEST-SEARCH.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 INSTALLATION. NANOLINUX IV,
000500               OPENCOBOL 0.31,
000600 DATE-WRITTEN. 2005-03-12.
000700*
000800 ENVIRONMENT DIVISION.
000900*
001000 DATA DIVISION.
001100*
001200 WORKING-STORAGE SECTION.
001300 01  RECORD-UTENTI.
001400     02  UTENTE              OCCURS 60 TIMES
001500                             INDEXED BY IND-UTENTE.
001600         03  COGNOME         PIC X(30).
001700         03  NOME            PIC X(30).
001800         03  NOTA            PIC X(200).
001900 77  EOJ                     PIC 9   VALUE ZERO.
002000 77  RISPOSTA                PIC XX.
002100 77  RICERCA                 PIC X(30).
002200*
002300 PROCEDURE DIVISION.
002400*------------------------ LIVELLO 0 -----------------
002500 MAIN.
002600     PERFORM INSERIMENTO-DATI
002700             VARYING IND-UTENTE FROM 1 BY 1
002800             UNTIL EOJ = 1.
002900     MOVE 0 TO EOJ.
003000     PERFORM SCANSIONE UNTIL EOJ = 1.
003100*
```

```
003200     STOP RUN.
003300*------------------------ LIVELLO 1 -----------------
003400 INSERIMENTO-DATI.
003500     DISPLAY IND-UTENTE, " INSERISCI IL COGNOME: ".
003600     ACCEPT COGNOME (IND-UTENTE).
003700     DISPLAY IND-UTENTE, " INSERISCI IL NOME: ".
003800     ACCEPT NOME (IND-UTENTE).
003900     DISPLAY IND-UTENTE,
003950             " INSERISCI UNA NOTA DESCRITTIVA: ".
004000     ACCEPT NOTA (IND-UTENTE).
004100*
004200     IF IND-UTENTE >= 60
004300       THEN
004400           MOVE 1 TO EOJ;
004500       ELSE
004600           DISPLAY "VUOI CONTINUARE? SI O NO",
004700           ACCEPT RISPOSTA;
004800           IF RISPOSTA = "SI"
004900             THEN
005000                 NEXT SENTENCE;
005100             ELSE
005200                 MOVE 1 TO EOJ.
005300*-------------------------------------------------
005400 SCANSIONE.
005500     DISPLAY "INSERISCI IL COGNOME DA CERCARE:".
005600     ACCEPT RICERCA.
005700     IF RICERCA = SPACES
005800       THEN
005900           MOVE 1 TO EOJ;
006000       ELSE
006100           SET IND-UTENTE TO 1,
006200           SEARCH UTENTE
006300                AT END
006400                     DISPLAY "IL COGNOME CERCATO ",
```

```
006500                                    "NON SI TROVA ",
006500                                    "NELLA TABELLA: ",
006600                                    QUOTE RICERCA QUOTE;
006700                          WHEN COGNOME (IND-UTENTE) = RICERCA
006800                              DISPLAY "IL COGNOME ", RICERCA,
006900                                      "SI TROVA NELLA ",
006950                                      "POSIZIONE ",
007000                                      IND-UTENTE.
007100*
```

Nell'esempio sono evidenziate le righe in cui si dichiara la tabella e quelle che eseguono la scansione. Si deve osservare che prima dell'istruzione '**SEARCH**', l'indice deve essere collocato manualmente nella posizione iniziale.

72.12.18.2 Ricerca in una tabella ordinata

«

La ricerca che si esegue con l'istruzione '**SEARCH ALL**' richiede che si rispettino alcune condizioni:

- i dati contenuti nella tabella devono risultare ordinati come previsto dalle chiavi già dichiarate;
- i dati contenuti nella tabella devono risultare tutti validi;
- le chiavi a cui si fa riferimento nella condizione di ricerca devono essere sufficienti a raggiungere l'informazione in modo univoco.

È importante considerare correttamente il problema dei dati validi: quando una tabella deve ricevere una quantità imprecisata di dati in elementi separati, questa deve essere stata dichiarata in modo abbastanza grande da poter contenere tutto, ma così facendo si ha la

certezza di avere una serie di celle vuote alla fine della tabella stessa. Per evitare che la scansione di ricerca tenga conto anche delle celle vuote, si dichiara la tabella con una quantità «variabile» di celle (con l'opzione '**OCCURS *m* TO *n* TIMES, DEPENDING ON *identifier***'). In realtà, più che trattarsi di una tabella che ha veramente una quantità variabile di celle, si fa in modo di stabilire qual è la dimensione massima, attraverso il controllo di una variabile apposita.

Dato il tipo di ricerca, non fa alcuna differenza il valore iniziale dell'indice della tabella.

L'esempio seguente mostra una variante del programma già descritto a proposito della ricerca sequenziale, modificato in modo da sfruttare una ricerca binaria. Si osservi che non è più necessario impostare il valore iniziale dell'indice, prima della scansione.

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   TEST-SEARCH-KEY.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 INSTALLATION. NANOLINUX IV,
000500               OPENCOBOL 0.31,
000600 DATE-WRITTEN. 2005-03-12.
000700*
000800 ENVIRONMENT DIVISION.
000900*
001000 DATA DIVISION.
001100*
001200 WORKING-STORAGE SECTION.
001300 01  RECORD-UTENTI.
001400     02  UTENTE           OCCURS 1 TO 60 TIMES
001500                          DEPENDING ON UTENTI-MAX
001600                          ASCENDING KEY IS COGNOME
001700                          INDEXED BY IND-UTENTE.
```

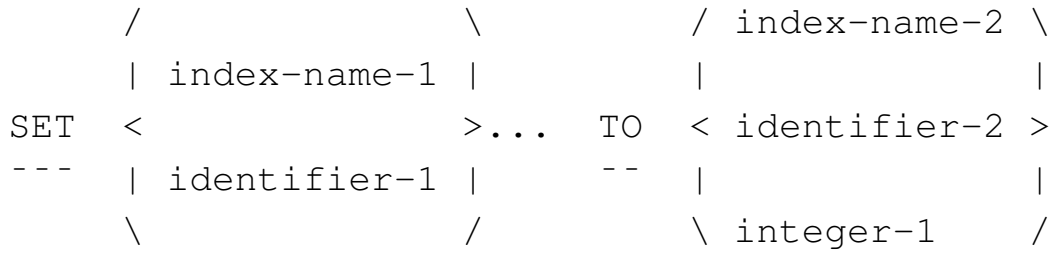
```
001800          03  COGNOME      PIC X(30).
001900          03  NOME         PIC X(30).
002000          03  NOTA         PIC X(200).
002100 77  UTENTI-MAX            USAGE IS INDEX.
002200 77  EOJ                   PIC 9   VALUE ZERO.
002300 77  RISPOSTA              PIC XX.
002400 77  RICERCA               PIC X(30).
002500*
002600 PROCEDURE DIVISION.
002700*------------------------ LIVELLO 0 -----------------
002800 MAIN.
002900     PERFORM INSERIMENTO-DATI
003000             VARYING IND-UTENTE FROM 1 BY 1
003100             UNTIL EOJ = 1.
003200     MOVE 0 TO EOJ.
003300     PERFORM SCANSIONE UNTIL EOJ = 1.
003400*
003500     STOP RUN.
003600*------------------------ LIVELLO 1 -----------------
003700 INSERIMENTO-DATI.
003800     MOVE IND-UTENTE TO UTENTI-MAX.
003900     DISPLAY IND-UTENTE, " INSERISCI IL COGNOME: ".
004000     ACCEPT COGNOME (IND-UTENTE).
004100     DISPLAY IND-UTENTE, " INSERISCI IL NOME: ".
004200     ACCEPT NOME (IND-UTENTE).
004300     DISPLAY IND-UTENTE,
004350             " INSERISCI UNA NOTA DESCRITTIVA: ".
004400     ACCEPT NOTA (IND-UTENTE).
004500*
004600     IF IND-UTENTE >= 60
004700       THEN
004800           MOVE 1 TO EOJ;
004900       ELSE
005000           DISPLAY "VUOI CONTINUARE? SI O NO",
```

```
005100           ACCEPT RISPOSTA;
005200           IF RISPOSTA = "SI"
005300             THEN
005400                 NEXT SENTENCE;
005500             ELSE
005600                 MOVE 1 TO EOJ.
005700*-------------------------------------------------
005800 SCANSIONE.
005900     DISPLAY "INSERISCI IL COGNOME DA CERCARE:".
006000     ACCEPT RICERCA.
006100     IF RICERCA = SPACES
006200       THEN
006300           MOVE 1 TO EOJ;
006400       ELSE
006600           SEARCH ALL UTENTE
006700                 AT END
006800                       DISPLAY "IL COGNOME CERCATO ",
006900                               "NON SI TROVA ",
006950                               "NELLA TABELLA: ",
007000                               QUOTE RICERCA QUOTE;
007100                 WHEN COGNOME (IND-UTENTE) = RICERCA
007200                     DISPLAY "IL COGNOME ", RICERCA,
007300                             "SI TROVA ",
007350                             "NELLA POSIZIONE ",
007400                             IND-UTENTE.
007500*
```

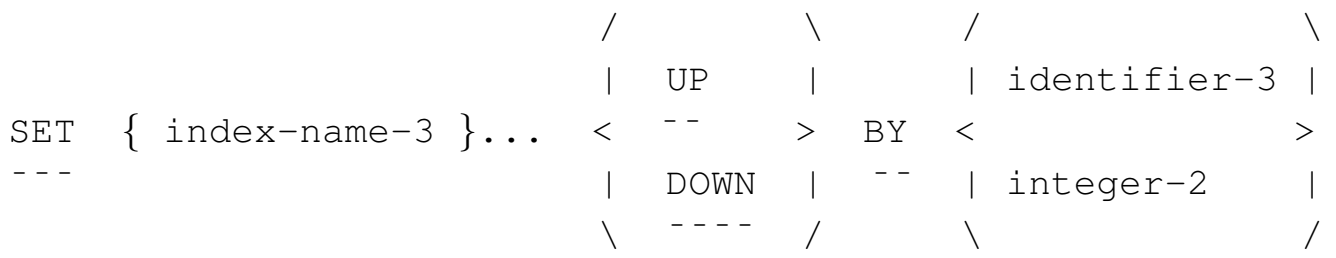
72.12.19 Istruzione «SET»

L'istruzione '**SET**' permette di attribuire un valore all'indice di una tabella; valore inteso come la posizione all'interno della stessa. Sono disponibili due schemi sintattici: attraverso il primo si attribuisce una

posizione determinata; con il secondo si incrementa o si decrementa l'indice di una certa quantità di posizioni.



Oppure:



In entrambi gli schemi sintattici, la variabile o le variabili indicate subito dopo la parola chiave '**SET**', sono quelle che rappresentano l'indice di una tabella e devono essere modificate. Nel primo caso, si intende assegnare loro il valore indicato o rappresentato dopo la parola chiave '**TO**', mentre nel secondo caso, l'indice viene incrementato ('**UP**') o diminuito ('**DOWN**') del valore posto dopo la parola chiave '**BY**'.

Quando nell'istruzione si usa una costante numerica, o una variabile numerica normale, deve trattarsi di un valore intero, che può essere senza segno, oppure può avere un segno positivo, con l'eccezione del caso dell'incremento o decremento dell'indice (nel secondo schema), dove può avere senso anche un segno negativo.

Nel primo schema sintattico, non sono ammesse tutte le combinazioni, rispetto a quando sembrerebbe dallo schema stesso. Per prima cosa, il valore che si attribuisce all'indice, deve essere valido nel-

l'ambito della tabella a cui si riferisce; inoltre, valgono gli abbinamenti dello schema successivo. Nello schema si distingue tra variabili intere normali, variabili di tipo indice associate a una tabella e variabili di tipo indice indipendenti.

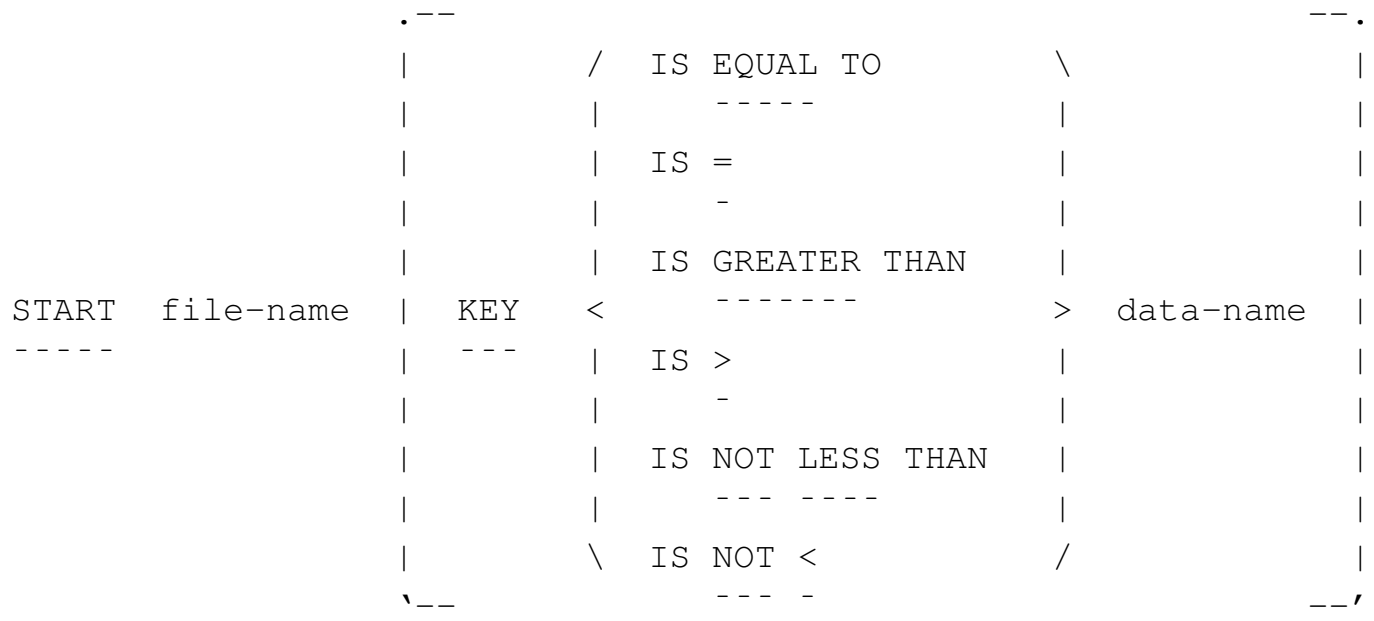
Tabella 72.230. Combinazioni degli operandi nell'istruzione '**SET**'.

	Variabile ricevente di tipo numerico intero (*integer data item*)	Variabile ricevente di tipo indice associata a una tabella (*index name*)	Variabile ricevente di tipo indice non associata ad alcuna tabella (*index data item*)
Valore assegnato costituito da una costante numerica intera	non ammesso	assegnamento valido	non ammesso
Valore assegnato costituito da una variabile numerica intera	non ammesso	assegnamento valido	non ammesso
Valore assegnato costituito da una variabile di tipo indice associata a una tabella	assegnamento valido	assegnamento valido	assegnamento valido
Valore assegnato costituito da una variabile di tipo indice non associata ad alcuna tabella	non ammesso	assegnamento valido	assegnamento valido

A seconda delle caratteristiche del compilatore, l'assegnamento di un valore a un indice può richiedere l'esecuzione di una conversione numerica appropriata.

72.12.20 Istruzione «START»

L'istruzione '**START**' consente di posizionare il puntatore del record logico di un file relativo o a indice, per il quale sia stato previsto un accesso sequenziale o dinamico.




```
[ INVALID KEY { imperative-statement }... ]
  -------
```

Il file indicato dopo la parola chiave '**START**' è quello all'interno del quale si vuole posizionare il puntatore del record logico. Come accennato, il file deve essere organizzato in modo relativo o a indice; inoltre, deve essere stato aperto in lettura ('**INPUT**') o in lettura e scrittura ('**I-O**').

La variabile che appare alla fine dello schema sintattico (***data-name***), può avere due significati differenti: se si tratta di un file organizzato in modo relativo, questa deve individuare la variabile definita con la dichiarazione '**RELATIVE KEY**' dell'istruzione '**SELECT**' del file stesso; se si tratta di un file organizzato a indice,

deve trattarsi della chiave di ordinamento (dichiarata come '**RECORD KEY**' o '**ALTERNATE RECORD KEY**' nell'istruzione '**SELECT**'), tenendo conto che può trattarsi di una porzione inferiore della chiave stessa, purché questa porzione si trovi a partire dall'inizio (a sinistra) della chiave.

L'opzione '**INVALID KEY**' introduce una o più istruzioni che vengono eseguite nel caso l'istruzione '**START**' fallisca a causa dell'indicazione di una chiave che con combacia secondo il tipo di confronto richiesto.

Nello schema sintattico, la parola chiave '**KEY**' precede un gruppo di parole che servono a stabilire la ***condizione di ricerca***. La corrispondenza con la chiave (costituita dal numero del record o dalla chiave di ordinamento vera e propria) può essere richiesta in modo esatto, oppure attraverso un altro tipo di relazione. Il record che per primo soddisfa la condizione di ricerca, è quello che viene selezionato. Una volta eseguita la selezione, il record potrebbe essere letto con l'istruzione '**READ**'.

Tabella 72.232. Condizione di ricerca.

Operatore	Descrizione
`KEY IS EQUAL TO data-name` `---  -----` `KEY IS = data-name` `---  -`	la chiave, o la sua porzione, corrisponde esattamente
`KEY IS GREATER THAN data-name` `---  -------` `KEY IS > data-name` `---  -`	la chiave del record è superiore al valore specificato

Operatore	Descrizione
`KEY IS NOT LESS THEN data-name` `---  ---  ----` `KEY IS NOT < THEN data-name` `---  ---  -`	la chiave del record non è inferiore (è maggiore o uguale) al valore specificato

La condizione di ricerca (assieme alla parola chiave '**KEY**') e il nome della variabile che ha il ruolo di chiave, possono essere omessi. In tal caso, la ricerca avviene in base alla corrispondenza esatta con il valore che ha la variabile che costituisce la chiave relativa del file, oppure con quello che ha il campo della chiave primaria dello stesso.

Quando la chiave indicata nell'istruzione '**START**' corrisponde a una porzione iniziale della chiave primaria o secondaria del file, il confronto si basa solo su quella porzione di chiave, ignorando il resto; nello stesso modo, se la chiave indicata nell'istruzione è più grande della chiave primaria o di quella secondaria, il confronto si basa solo sulla dimensione della chiave che ha il file effettivamente (che risulta essere più breve).

Comunque sia l'esito della ricerca, l'esecuzione dell'istruzione '**START**', provoca l'aggiornamento della variabile che rappresenta lo stato del file ('**FILE STATUS**').

Listato 72.233. Programma elementare che legge un file relativo, ad accesso sequenziale, partendo dal terzo record.

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   TEST-READ-SEQ-START.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-03-13.
000500*
000600 ENVIRONMENT DIVISION.
```

```
000700*
000800 INPUT-OUTPUT SECTION.
000900*
001000 FILE-CONTROL.
001100*
001200     SELECT FILE-DA-LEGGERE ASSIGN TO "input.rel"
001300                            ORGANIZATION IS RELATIVE
001400                            RELATIVE KEY IS N-RECORD
001500                            ACCESS MODE IS SEQUENTIAL.
001600*
001700 DATA DIVISION.
001800*
001900 FILE SECTION.
002000*
002100 FD  FILE-DA-LEGGERE
002200     LABEL RECORD IS STANDARD.
002300*
002400 01  RECORD-DA-LEGGERE PIC X(30).
002500*
002600 WORKING-STORAGE SECTION.
002700 77  EOF               PIC 9   VALUE ZERO.
002800 77  N-RECORD          PIC 999 VALUE ZERO.
002900*
003000 PROCEDURE DIVISION.
003100*------------------------ LIVELLO 0 -----------------
003200 MAIN.
003300     OPEN INPUT FILE-DA-LEGGERE.
003400     MOVE 3 TO N-RECORD.
003500     START FILE-DA-LEGGERE KEY IS EQUAL TO N-RECORD
003600           INVALID KEY
003700                     MOVE 1 TO EOF.
003800     READ FILE-DA-LEGGERE
003900          AT END
004000               MOVE 1 TO EOF.
```

```
004100     PERFORM LETTURA UNTIL EOF = 1.
004200     CLOSE FILE-DA-LEGGERE.
004300*
004400     STOP RUN.
004500*------------------------ LIVELLO 1 -----------------
004600 LETTURA.
004700     DISPLAY RECORD-DA-LEGGERE.
004800     READ FILE-DA-LEGGERE
004900          AT END
005000               MOVE 1 TO EOF.
005100*
```

72.12.21 Istruzione «STOP RUN»

L'istruzione '**STOP RUN**' conclude il funzionamento del programma; pertanto, può trovarsi soltanto alla fine di un gruppo di istruzioni.

```
STOP RUN.
---- ---
```

Storicamente esiste una versione alternativa, ma superata, dell'istruzione '**STOP**', alla quale si associa una costante, allo scopo di mostrare tale valore attraverso il terminale principale. In quella situazione, l'esecuzione del programma veniva sospesa e poteva essere fatta riprendere dall'utente.

Considerato che esiste la possibilità di usare istruzioni come '**DISPLAY**' e '**ACCEPT**', è meglio utilizzare esclusivamente l'istruzione '**STOP RUN**' per l'arresto del programma, senza altre varianti.

72.12.22 Istruzione «STRING»

«

L'istruzione '**STRING**' consente di riempire delle variabili alfanumeriche specificando un punto di inizio, espresso in caratteri.

```
          /  /                \                  /  identifier-2  \  \
          |  |  identifier-1   |                  |                |  |
STRING    <  <                >... DELIMITED BY  <  literal-2     >  >...
------    |  |  literal-1      |    ---------     |                |  |
          \  \                /                  \  SIZE          /  /
                                                     ----

    INTO  identifier-3
    ----

    [ WITH POINTER  identifier-4 ]
           -------

    [ ON OVERFLOW  { imperative-statement-1 }... ]
         --------
```

Quello che si mette dopo la parola chiave '**STRING**' è un elenco di valori che si traducono in informazioni alfanumeriche, che vengono considerati come se fossero concatenati tra di loro. Dopo la parola '**DELIMITED**' si deve specificare un modo per delimitare la stringa complessiva indicata a sinistra. Se si usa la parola chiave '**SIZE**', si intende considerare tutta la stringa alfanumerica complessiva, altrimenti, si seleziona solo la parte che si trova a sinistra, prima di ciò che viene indicato come riferimento.

La stringa complessiva, eventualmente ridotta a destra in qualche modo, viene copiata all'interno della variabile indicata dopo la parola '**INTO**'. La stringa viene copiata a partire dalla prima posizione, oppure dalla posizione specificata dal numero indicato dopo la parola '**POINTER**'. Dopo la parola '**POINTER**' va indicata una variabile numerica, che, oltre a indicare la posizione iniziale dell'inserimento della stringa, viene incrementata di conseguenza, per i caratteri che

vengono inseriti effettivamente.

Se si utilizza la parola '**OVERFLOW**', le istruzioni che appaiono subito dopo tale parola vengono eseguite se l'inserimento nella variabile di destinazione va oltre la fine della variabile stessa.

L'esempio successivo mostra un piccolo programma completo che compila in più fasi una variabile ricevente. La variabile ricevente contiene inizialmente una serie di simboli '**#**', per consentire di vedere facilmente cosa succede al suo interno, durante le varie fasi.

Listato 72.236. Programma elementare che dimostra il funzionamento di '**STRING**'.

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   TEST-STRING.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 INSTALLATION. NANOLINUX IV,
000500               OPENCOBOL 0.31,
000600 DATE-WRITTEN. 2005-03-16.
000700*
000800 ENVIRONMENT DIVISION.
000900*
001000 DATA DIVISION.
001100*
001200 WORKING-STORAGE SECTION.
001300 77  TESTO-RICEVENTE      PIC X(40) VALUE ALL "#".
001400 77  PUNTATORE            PIC 99.
001500*
001600 PROCEDURE DIVISION.
001700*------------------------ LIVELLO 0 ----------------
001800 MAIN.
001900     MOVE 1 TO PUNTATORE.
002000     DISPLAY PUNTATORE, " ", TESTO-RICEVENTE.
002100     STRING "CIAO", SPACE, DELIMITED BY SIZE
002200            INTO TESTO-RICEVENTE
```

```
002250            WITH POINTER PUNTATORE.
002300     DISPLAY PUNTATORE, " ", TESTO-RICEVENTE.
002400     STRING "COME STAI?" DELIMITED BY SIZE
002500            INTO TESTO-RICEVENTE
002550            WITH POINTER PUNTATORE.
002600     DISPLAY PUNTATORE, " ", TESTO-RICEVENTE.
002700     MOVE 11 TO PUNTATORE.
002800     STRING "VA LA VITA?" DELIMITED BY SIZE
002900            INTO TESTO-RICEVENTE
002950            WITH POINTER PUNTATORE.
003000     DISPLAY PUNTATORE, " ", TESTO-RICEVENTE.
003100*
003200     STOP RUN.
003300*
```

Dopo aver compilato il programma, eseguendo ciò che si ottiene, di dovrebbe vedere il risultato seguente attraverso il terminale:

```
01 ########################################
06 CIAO ##################################
16 CIAO COME STAI?########################
22 CIAO COME VA LA VITA?##################
```

Come si può vedere leggendo il sorgente del programma, dopo l'inserimento della stringa '**CIAO** ', la variabile usata come puntatore all'interno della variabile di destinazione, si trova a essere già posizionata sulla sesta colonna, in modo che un inserimento ulteriore si trovi già nella posizione necessaria. Dopo, viene riposizionato il puntatore per sovrascrivere la parola «STAI».

72.12.23 Istruzione «SUBTRACT»

L'istruzione '**SUBTRACT**' consente di eseguire delle sottrazioni. Sono previsti diversi formati per l'utilizzo di questa istruzione.

```
           /               \  .--             --.
           | identifier-1 |  |  identifier-2  |
SUBTRACT   <               >  |                |...
--------   | literal-1     |  |  literal-2     |
           \               /  `--             --'

    FROM { identifier-m  [ROUNDED] }...
    ----                  -------

    [ ON SIZE ERROR  imperative-statement ]
         ---- -----
```

Nello schema sintattico appena mostrato, si vede che dopo la parola chiave '**SUBTRACT**' si elencano delle costanti o variabili con valore numerico, che vengono sommate assieme inizialmente, per poi sottrarre tale valore dal contenuto delle variabili specificate dopo la parola chiave '**FROM**'. L'opzione '**ROUNDED**' richiede di eseguire un arrotondamento se la variabile ricevente non può rappresentare in modo esatto il valore; l'opzione '**SIZE ERROR**' serve a eseguire un'istruzione nel caso una delle variabili riceventi non possa accogliere la porzione più significativa del valore ottenuto dalla somma. Si osservi l'esempio seguente:

```
000000     SUBTRACT 1, 2, 3, FROM A.
```

Supponendo che la variabile '**A**', prima della somma contenga il valore 10, dopo la somma contiene il valore 4 (10-1-2-3).

```
          /               \   .-- /               \ --.
          | identifier-1  |   |    | identifier-2  |   |
SUBTRACT <                >   |    <               >   |...
--------  | literal-1     |   |    | literal-2     |   |
          \               /   `-- \               / --'

          /               \
          | identifier-3  |
     FROM <                >   [ROUNDED] ...
     ----  | identifier-3  |    -------
          \               /

     GIVING { identifier-n   [ROUNDED] }...
     ------                   -------

     [ ON SIZE ERROR  imperative-statement ]
          ---- -----
```

Quando si utilizza la parola chiave '**GIVING**', si può indicare un solo valore dopo la parola chiave '**FROM**' e il risultato della sottrazione viene assegnato alle variabili che sono elencate dopo la parola '**GIVING**', senza tenere in considerazione il loro valore iniziale. Valgono le stesse considerazioni già fatte a proposito delle opzioni '**ROUNDED**' e '**SIZE ERROR**'. Si osservi l'esempio seguente:

```
000000     SUBTRACT 1, 2, 3, FROM 10 GIVING A.
```

Qualunque sia il valore iniziale della variabile '**A**', dopo la somma questa contiene il valore 4 (10-1-2-3).

```
           /                \
           | CORR           |
SUBTRACT   < ----           >  identifier-1  FROM  identifier-2  [ROUNDED]
--------   | CORRESPONDING  |                 ----                -------
           \ -------------- /

    [ ON SIZE ERROR  imperative-statement ]
         ---- -----
```

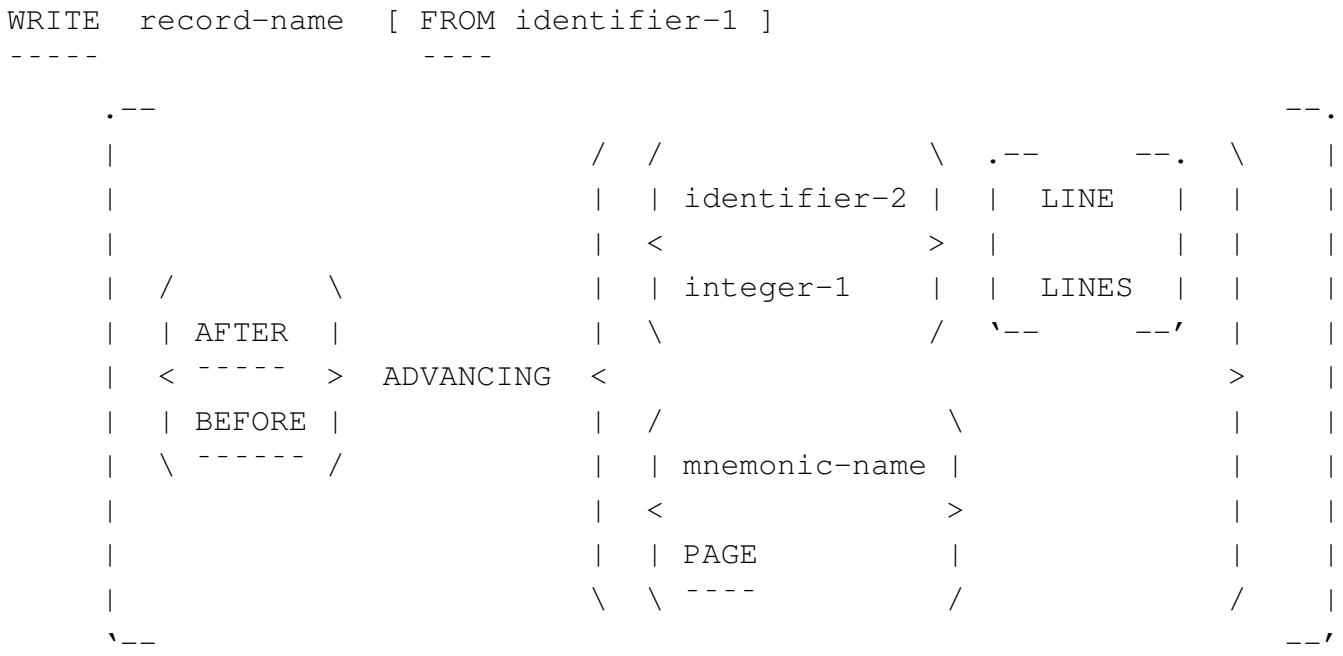
In questo ultimo caso, la sottrazione fa riferimento a variabili strutturate, dove i campi della prima variabile devono essere sottratti ai campi della seconda variabile che hanno lo stesso nome della prima. Valgono le stesse considerazioni già fatte a proposito delle opzioni '**ROUNDED**' e '**SIZE ERROR**'.

72.12.24 Istruzione «WRITE»

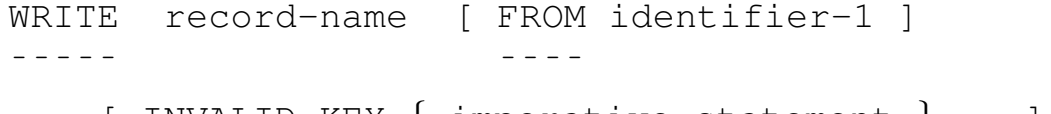
L'istruzione '**WRITE**' scrive un record logico in un file, aperto in modo appropriato. Nel caso di un file organizzato in modo sequenziale, il file può essere aperto in scrittura ('**OUTPUT**') o in estensione ('**EXTEND**'); nel caso di un file ad accesso diretto, organizzato in modo relativo o a indice, questo può essere stato aperto in scrittura ('**OUTPUT**') o in lettura e scrittura ('**I-O**'), inoltre, se si usa l'accesso sequenziale, è consentito anche in caso di apertura in estensione ('**EXTEND**').

L'istruzione '**WRITE**' viene usata con due schemi sintattici alternativi: uno per i file organizzati in modo sequenziale e l'altro per tutti gli altri casi. Il formato adatto ai file sequenziali contiene, in particolare, opzioni specifiche per l'avanzamento della carta di una stampante.

File organizzati in modo sequenziale



File organizzati in modo relativo e a indice



Gli schemi sintattici mostrati hanno in comune la prima parte: il nome della variabile che fa riferimento al record, serve a individuare implicitamente il file; la variabile indicata dopo la parola opzionale '**FROM**', permette di copiare tale variabile su quella del record, prima di procedere alla scrittura, come se si usasse l'istruzione '**MOVE**' prima di '**WRITE**':

```
MOVE identifier-1 TO record-name;
----

WRITE record-name
-----

    [ omissis ]
```

Quando la scrittura avviene con successo, il contenuto del record non è più disponibile in memoria, a meno di averne una copia per altri motivi (per esempio a causa dell'utilizzo dell'opzione '**FROM**').

La scrittura di un file organizzato in modo sequenziale implica l'utilizzo del primo schema sintattico. Nello schema sintattico non è previsto il controllo di alcuna condizione di errore, che comunque potrebbe verificarsi, quando per qualche ragione non è possibile scrivere nel file. Le opzioni '**AFTER ADVANCING**' e '**BEFORE ADVANCING**', servono rispettivamente per richiedere un avanzamento preventivo o successivo alla scrittura. Per un file di dati, non ha significato l'uso di tali opzioni, che invece servono precisamente per la stampa, o per la creazione di file di testo (destinati eventualmente alla stampa). L'avanzamento può essere specificato in un numero intero di righe (***identifier-2*** o ***integer-1***), oppure richiedendo un salto pagina, con la parola chiave '**PAGE**'. Il nome mnemonico che può essere indicato in alternativa alla parola chiave '**PAGE**' può servire per attribuire un nome alternativo proprio alla parola '**PAGE**', oppure a fare riferimento a un'altra parola chiave (alternativa a '**PAGE**'), che si riferisce a caratteristiche speciali, legate alla stampa, che il proprio compilatore può gestire.

Si osservi che un file organizzato in modo sequenziale, per il quale abbiano senso le opzioni di avanzamento, è bene che sia stato dichiarato con l'opzione '**LINE SEQUENTIAL**'.

Il secondo schema sintattico può essere usato per i file che non hanno un'organizzazione sequenziale. In questo caso vengono a mancare i controlli di avanzamento della riga o della pagina, ma si aggiunge

la verifica di un errore di scrittura, attraverso l'opzione '**INVALID KEY**', dopo la quale appaiono le istruzioni da eseguire in caso di problemi.

Nel caso di file organizzati in modo relativo, ad accesso sequenziale, il comportamento è lo stesso che si avrebbe con un file sequenziale puro e semplice, con la differenza che la variabile designata a contenere il numero del record viene impostata automaticamente e che si può verificare la condizione controllata dall'opzione '**INVALID KEY**' se si tenta di espandere il file oltre i limiti imposti esternamente al programma. Se invece questo tipo di file viene usato con un accesso diretto o dinamico, il numero del record (inserito nella variabile definita con la dichiarazione '**RELATIVE KEY**' dell'istruzione '**SELECT**' del file stesso) deve essere indicato espressamente: se il numero indicato corrisponde a un record già esistente, oppure se si tenta di scrivere oltre i limiti stabiliti esternamente al programma, si ottiene la condizione di errore controllata dall'opzione '**INVALID KEY**'.

Nel caso di file organizzati a indice, l'inserimento dei record avviene tenendo conto delle chiavi previste per questo. In linea di principio, le chiavi non devono essere doppie; pertanto, il tentativo di inserire un record che contiene una chiave già esistente nel file (primaria o secondaria che sia), provoca un errore che può essere controllato attraverso l'opzione '**INVALID KEY**'. Naturalmente, se nella dichiarazione delle chiavi è stato stabilito che possono anche essere doppie, tale errore non si verifica e la scrittura avviene con successo.

Un file organizzato a indice può essere scritto utilizzando un accesso sequenziale, ma in tal caso, la scrittura deve avvenire rispettando l'ordine crescente della chiave primaria, altrimenti si verifica un errore che si può controllare con l'opzione '**INVALID KEY**'.

L'utilizzo dell'istruzione '**WRITE**' implica l'aggiornamento della variabile che rappresenta lo stato del file ('**FILE STATUS**').

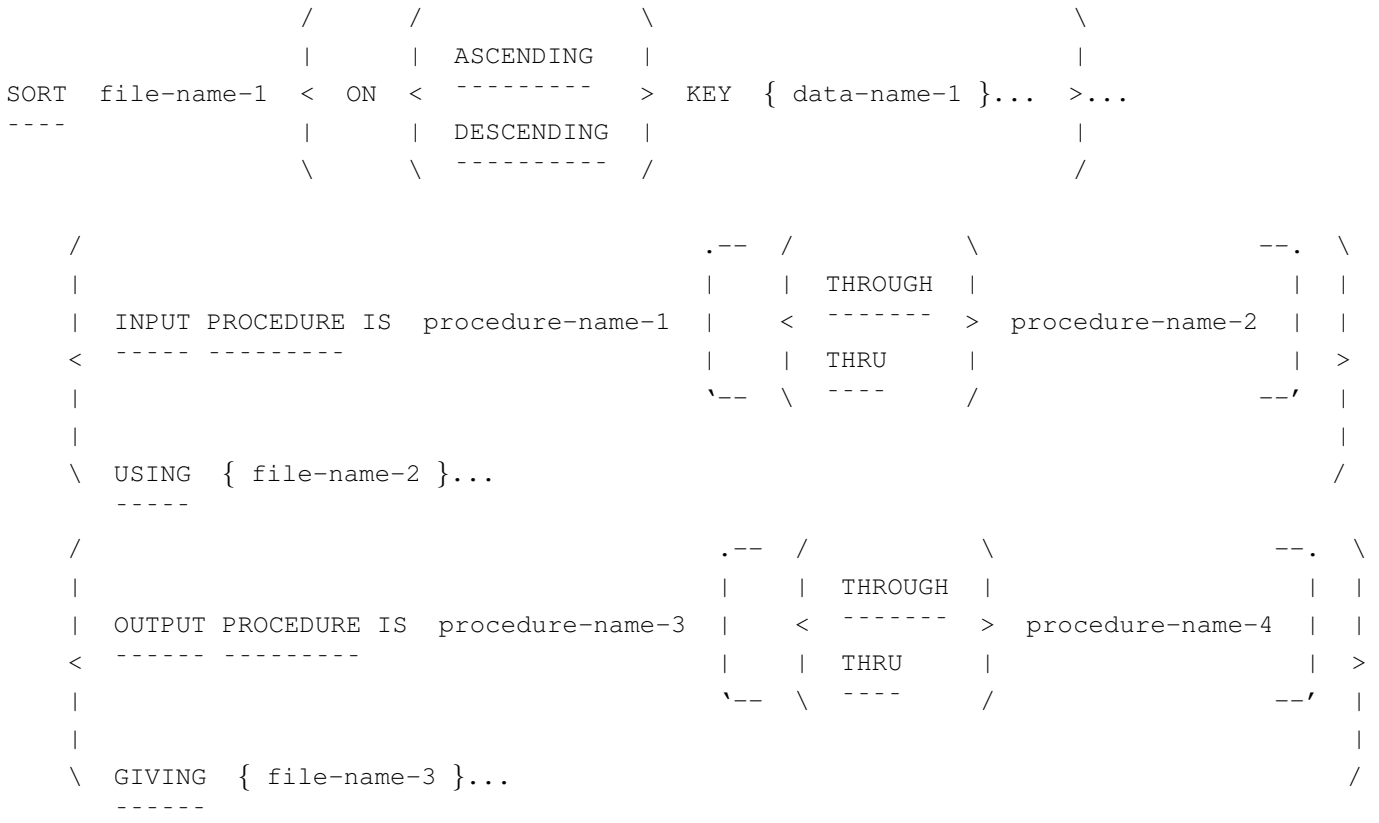
72.13 Riordino e fusione

Il riordino e la fusione del contenuto dei file sono gestite normalmente attraverso funzionalità speciali del linguaggio COBOL. Si utilizzano in particolare file dichiarati con l'indicatore di livello '**SD**' nella sezione '**FILE SECTION**', per svolgere la funzione di riordino o di fusione, mentre i file da ordinare o da fondere, assieme al risultato dell'ordinamento o della fusione, possono essere file normali organizzati secondo le esigenze del programma.

I file che prendono parte alle operazioni di riordino e di fusione, **non devono essere aperti o chiusi** durante tali operazioni.

72.13.1 Riordino

Il riordino di un file, con l'istruzione '**SORT**' del COBOL, richiede in linea di massima il coinvolgimento di tre file: il file che formalmente serve come appoggio per svolgere la funzione di ordinamento, dichiarato nella sezione '**FILE SECTION**' con l'indicatore di livello '**SD**'; un file per i dati in ingresso da ordinare; un file per accogliere il risultato del procedimento di ordinamento.



Il file che nello schema sintattico appare nominato come ***file-name-1***, è quello che deve essere dichiarato nella sezione '**FILE SECTION**' con l'indicatore di livello '**SD**'. Dopo la parola '**ASCENDING**' si indica un elenco di chiavi di ordinamento crescenti; dopo la parola '**DESCENDING**' si indica un elenco di chiavi di ordinamento decrescenti. Le chiavi di ordinamento sono campi del record del file ***file-name-1*** e la possibilità di indicare più chiavi serve a definire una gerarchia di ordinamento quando se ne crea la necessità. In pratica, la presenza di più chiavi fa sì che in presenza di chiavi doppie a un certo livello gerarchico, permetta di distinguere l'ordine dei record utilizzando anche le chiavi di livello inferiore.

Nell'ordinamento di un file, la presenza di record con tutte le chiavi previste doppie, ha l'unico inconveniente di non poter stabilire quale sequenza effettiva ottengono tali record dopo l'ordinamento.

Il file da ordinare può essere costituito dal nome che appare dopo la parola '**USING**', oppure può essere generato da un gruppo di procedure del programma, specificate dopo le parole '**INPUT PROCEDURE**'. Il file indicato dopo la parola '**USING**' è un file dichiarato normalmente, con l'organizzazione e l'accesso desiderati.

Il file che risulta dall'ordinamento può essere costituito dal nome che appare dopo la parola '**GIVING**', oppure può essere letto da un gruppo di procedure del programma, specificate dopo le parole '**OUTPUT PROCEDURE**'. Il file indicato dopo la parola '**GIVING**' è un file dichiarato normalmente, con l'organizzazione e l'accesso desiderati.

La gestione dei dati in ingresso o in uscita, attraverso delle procedure, viene descritto in altre sezioni; per il momento viene mostrato un esempio di ordinamento tipico, che coinvolge il file per il riordino, più due file per i dati (in ingresso e in uscita).

Listato 72.247. Programma elementare che dimostra il funzionamento di '**SORT**'.

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   TEST-SORT.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-02-25.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 INPUT-OUTPUT SECTION.
```

```
000900*
001000 FILE-CONTROL.
001100*
001200     SELECT FILE-DA-ORDINARE     ASSIGN TO "input.seq".
001300     SELECT FILE-ORDINATO        ASSIGN TO "output.seq".
001400     SELECT FILE-PER-IL-RIORDINO ASSIGN TO "sort.tmp".
001500*
001600 DATA DIVISION.
001700*
001800 FILE SECTION.
001900*
002000 FD  FILE-DA-ORDINARE.
002100 01  RECORD-DA-ORDINARE           PIC X(10).
002200*
002300 FD  FILE-ORDINATO.
002400 01  RECORD-ORDINATO              PIC X(10).
002500*
002600 SD  FILE-PER-IL-RIORDINO.
002700*
002800 01  RECORD-PER-IL-RIORDINO.
002900     02  CHIAVE-ORDINAMENTO       PIC X(5).
003000     02  FILLER                   PIC X(5).
003100*
003200 PROCEDURE DIVISION.
003300*----------------------- LIVELLO 0 ----------------
003400 MAIN.
003500     SORT FILE-PER-IL-RIORDINO,
003600          ON ASCENDING KEY CHIAVE-ORDINAMENTO,
003700          USING FILE-DA-ORDINARE,
003800          GIVING FILE-ORDINATO.
003900*
004000     STOP RUN.
004100*
```

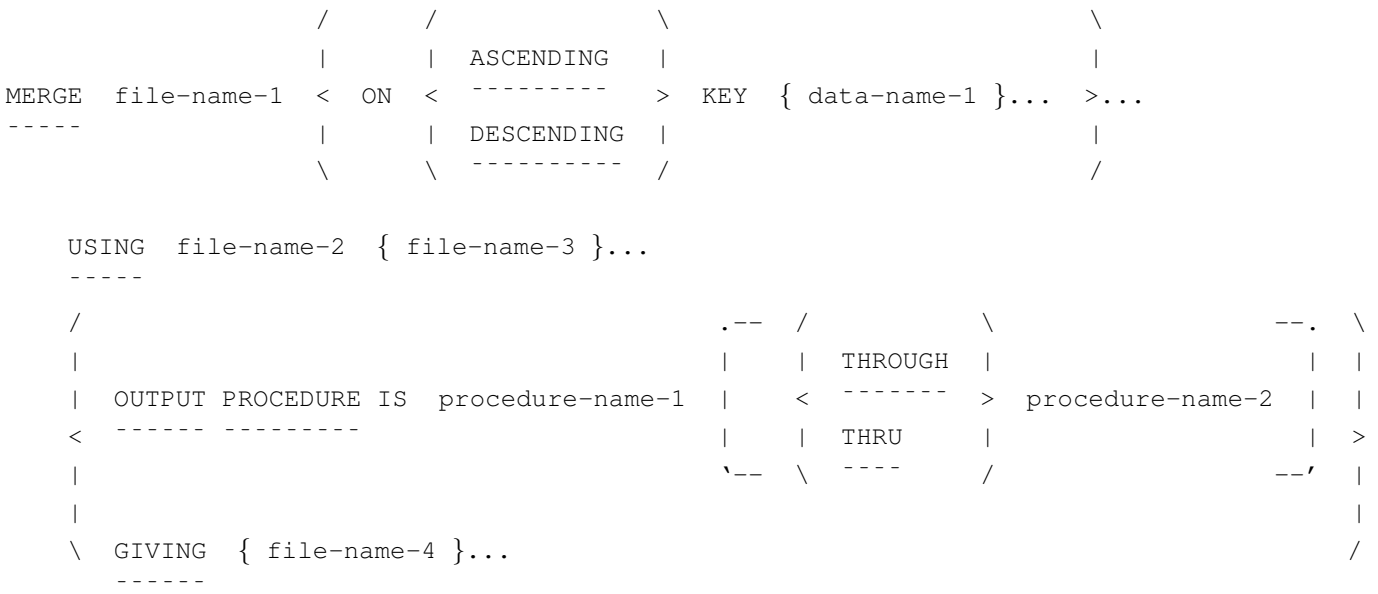
Nell'esempio, il file usato per ottenere il riordino è '`sort.tmp`'; il

file da ordinare è ‘`input.seq`’ (organizzato in modo sequenziale); il file ordinato che si ottiene è ‘`output.seq`’ (anche questo organizzato in modo sequenziale). Come chiave di ordinamento si prendono in considerazione i primi cinque byte del record.

Lo schema sintattico consentirebbe l’indicazione di più file da ordinare e di più file ordinati da generare. Nel primo caso, i dati dei vari file vengono raccolti assieme e considerati parte di un file unico da ordinare; i file ordinati da generare, invece, rappresentano copie dello stesso risultato ordinato.

72.13.2 Fusione

La fusione di due o più file, con l’istruzione ‘**`MERGE`**’ del COBOL, richiede la presenza di due o più file ordinati nello stesso modo, che si vogliono mettere insieme in un solo file ordinato. Per compiere questa funzione, si aggiunge un file ulteriore, dichiarato nella sezione ‘**`FILE SECTION`**’ con l’indicatore di livello ‘**`SD`**’.



La prima parte dello schema sintattico va interpretata nello stesso modo di quello per il riordino; dove ***file-name-1*** è il file che de-

ve essere dichiarato nella sezione '**FILE SECTION**' con l'indicatore di livello '**SD**' e le variabili indicate dopo le parole '**ASCENDING**' o '**DESCENDING**' sono le chiavi di ordinamento previste nei file in ingresso.

Successivamente si può osservare nello schema sintattico che sono previsti soltanto file in ingresso, dopo la parola '**USING**', tenendo in considerazione il fatto che devono essere almeno due. Questi file **devono risultare già ordinati** secondo le chiavi previste, altrimenti il risultato della fusione non è prevedibile.

Il risultato della fusione può essere costituito dal nome che appare dopo la parola '**GIVING**', oppure può essere letto da un gruppo di procedure del programma, specificate dopo le parole '**OUTPUT PROCEDURE**'. Il file indicato dopo la parola '**GIVING**' è un file dichiarato normalmente, con l'organizzazione e l'accesso desiderati.

La gestione dei dati in uscita, attraverso delle procedure, viene descritto in altre sezioni; per il momento viene mostrato un esempio di fusione tipico, si hanno i file '`input-1.seq`' e '`input-2.seq`' ordinati, si vuole ottenere il file '`output.seq`' con la somma dei record, mantenendo l'ordinamento:

Listato 72.249. Programma elementare che dimostra il funzionamento di '**MERGE**'.

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   TEST-MERGE.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-03-18.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 INPUT-OUTPUT SECTION.
```

```
000900*
001000 FILE-CONTROL.
001100*
001200     SELECT FILE-INPUT-1 ASSIGN TO "input-1.seq".
001300     SELECT FILE-INPUT-2 ASSIGN TO "input-2.seq".
001400     SELECT FILE-OUTPUT  ASSIGN TO "output.seq".
001500     SELECT FILE-PER-LA-FUSIONE
001550                         ASSIGN TO "merge.tmp".
001600*
001700 DATA DIVISION.
001800*
001900 FILE SECTION.
002000*
002100 FD  FILE-INPUT-1
002200 01  RECORD-1                      PIC X(10).
002300*
002400 FD  FILE-INPUT-2
002500 01  RECORD-2                      PIC X(10).
002600*
002700 FD  FILE-OUTPUT
002800 01  RECORD-OUTPUT                 PIC X(10).
002900*
003000 SD  FILE-PER-LA-FUSIONE.
003100*
003200 01  RECORD-PER-LA-FUSIONE.
003300     02  CHIAVE-ORDINAMENTO        PIC X(5).
003400     02  FILLER                    PIC X(5).
003500*
003600 PROCEDURE DIVISION.
003700*------------------------ LIVELLO 0 -----------------
003800 MAIN.
003900     MERGE FILE-PER-LA-FUSIONE
004000           ON ASCENDING KEY CHIAVE-ORDINAMENTO,
004100           USING FILE-INPUT-1,
```

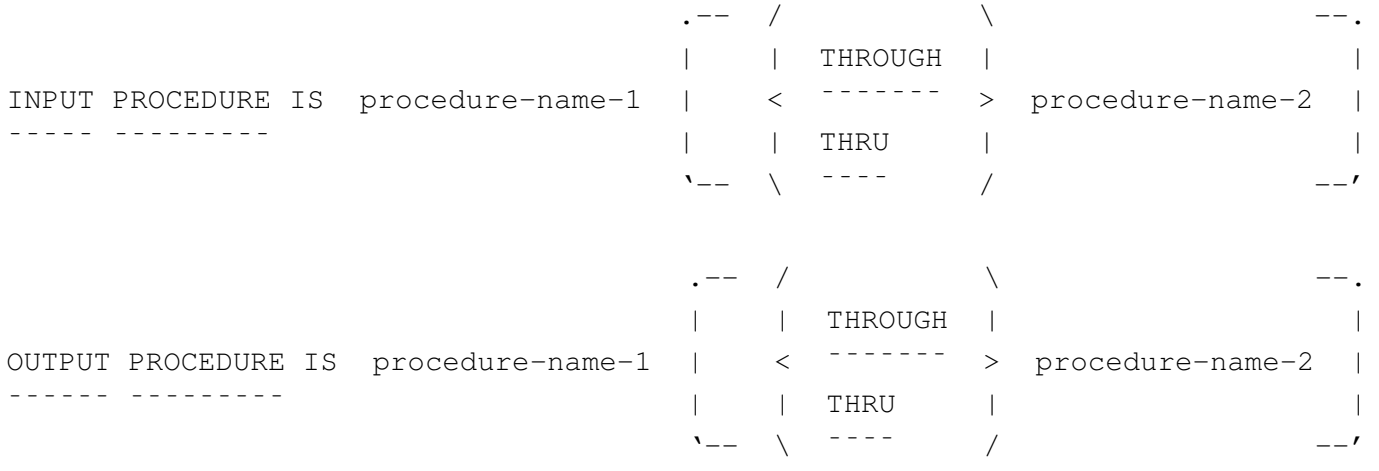
```
004200                    FILE-INPUT-2,
004300              GIVING FILE-OUTPUT.
004400*
004500      STOP RUN.
004600*
```

Lo schema sintattico consentirebbe l'indicazione di più file ordinati da generare: se viene indicato più di un file per raccogliere il risultato della fusione, questi ottengono lo stesso contenuto; i file in sé possono essere differenti, se possiedono una diversa organizzazione.

72.13.3 Gestire i dati in ingresso o in uscita attraverso delle procedure

«

Nelle istruzioni '**SORT**' e '**MERGE**', a seconda dei casi, esiste la possibilità di specificare un gruppo di procedure con le forme seguenti:



Queste procedure sono da intendere come un intervallo di sezioni o di paragrafi della divisione '**PROCEDURE DIVISION**', da ***procedure-name-1*** a ***procedure-name-2***. Questa porzione di sezione o di paragrafi deve però rispettare delle condizioni: deve servire esclusivamente per lo scopo del riordino o della fusione; non può

contenere chiamate a procedure esterne; non può essere usata nel programma per fini differenti.

In generale, se si intendono usare delle procedure per generare dati da ordinare, leggere i dati ordinati o fusi, conviene gestire la divisione '**PROCEDURE DIVISION**' in sezioni. L'esempio seguente mostra proprio una sezione che potrebbe essere usata per leggere il risultato di un file ordinato o fuso:

Listato 72.252. Esempio di sezione da usare come procedura di uscita di '**SORT**' o '**MERGE**'.

```
004100 MOSTRA-FILE-ORDINATO SECTION.
004200 INIZIO.
004300     PERFORM MOSTRA-RECORD UNTIL EOF = 1.
004400     GO TO FINE.
004500 MOSTRA-RECORD.
004600     RETURN FILE-PER-IL-RIORDINO RECORD
004700           AT END MOVE 1 TO EOF,
004800                  DISPLAY "FINE DEL FILE ORDINATO".
004900     IF EOF = 0
005000     THEN
005100         DISPLAY RECORD-PER-IL-RIORDINO.
005200 FINE.
005300     EXIT.
```

Nell'esempio si vede anche l'uso del famigerato '**GO TO**', allo scopo di uscire dalla sezione dopo l'esecuzione del ciclo di chiamate al paragrafo '**MOSTRA-RECORD**', dal momento che l'istruzione '**EXIT**', secondo lo standard, deve trovarsi da sola in un paragrafo.

72.13.4 Lettura del risultato dell'ordinamento o della fusione attraverso una procedura

«

Quando si usano le istruzioni '**SORT**' o '**MERGE**', invece di generare un file ordinato o fuso, è possibile leggere il risultato dell'ordinamento o della fusione, specificando la chiamata di un intervallo di procedure (paragrafi o sezioni):

```
                                        .--  /          \                  --.
                                        |    |  THROUGH  |                    |
OUTPUT PROCEDURE IS  procedure-name-1   |    <  -------  >  procedure-name-2  |
------ ---------                        |    |  THRU     |                    |
                                        `--  \  ----     /                  --'
```

Nell'ambito dell'intervallo di procedure chiamato, occorre usare l'istruzione '**RETURN**' per leggere questi dati dal file di riordino o di fusione:

```
RETURN file-name-1 [NEXT] RECORD  [ INTO identifier ]
------             ----           ----

    AT END  { imperative-statement }...
       ---
```

L'istruzione '**RETURN**' funziona a tutti gli effetti come l'istruzione '**READ**' di un file sequenziale, dove il file indicato è precisamente quello che appare nell'istruzione '**SORT**' o '**MERGE**' chiamante, con la stessa metavariabile.

Listato 72.255. Esempio di lettura del risultato di un ordinamento attraverso una procedura.

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   TEST-SORT-4.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-03-18.
000500*
000600 ENVIRONMENT DIVISION.
```

```
000700*
000800 INPUT-OUTPUT SECTION.
000900*
001000 FILE-CONTROL.
001100*
001200     SELECT FILE-DA-ORDINARE     ASSIGN TO "input.seq".
001300     SELECT FILE-PER-IL-RIORDINO ASSIGN TO "sort.tmp".
001400*
001500 DATA DIVISION.
001600*
001700 FILE SECTION.
001800*
001900 FD  FILE-DA-ORDINARE.
002000 01  RECORD-DA-ORDINARE           PIC X(10).
002100*
002200 SD  FILE-PER-IL-RIORDINO.
002300*
002400 01  RECORD-PER-IL-RIORDINO.
002500     02  CHIAVE-ORDINAMENTO       PIC X(5).
002600     02  FILLER                   PIC X(5).
002700
002800 WORKING-STORAGE SECTION.
002900 77  EOF                          PIC 9 VALUE 0.
003000*
003100 PROCEDURE DIVISION.
003200*------------------------ LIVELLO 0 -----------------
003300 MAIN SECTION.
003400 INIZIO.
003500     SORT FILE-PER-IL-RIORDINO,
003600          ON ASCENDING KEY CHIAVE-ORDINAMENTO,
003700          USING FILE-DA-ORDINARE,
003800          OUTPUT PROCEDURE IS MOSTRA-FILE-ORDINATO.
003900*
004000     STOP RUN.
004100*
```

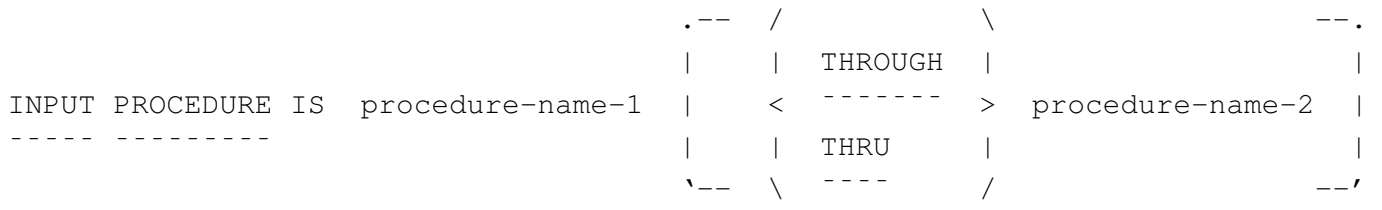
```
004200*---------------------- SORT-MERGE PROCEDURE ----------
004300 MOSTRA-FILE-ORDINATO SECTION.
004400 INIZIO.
004500     PERFORM MOSTRA-RECORD UNTIL EOF = 1.
004600     GO TO FINE.
004700 MOSTRA-RECORD.
004800     RETURN FILE-PER-IL-RIORDINO RECORD
004900           AT END MOVE 1 TO EOF,
005000                  DISPLAY "FINE DEL FILE ORDINATO".
005100     IF EOF = 0
005200     THEN
005300         DISPLAY RECORD-PER-IL-RIORDINO.
005400 FINE.
005500     EXIT.
005600*
```

L'esempio riguarda la visualizzazione di un file ordinato, senza generare il file stesso, ma si applica tale e quale al caso della fusione.

72.13.5 Acquisizione dei dati per il riordino da una procedura

«

Limitatamente al caso del riordino, con l'istruzione '**SORT**', è possibile acquisire i record da riordinare attraverso una procedura:



Nell'ambito dell'intervallo di procedure chiamato, occorre usare l'istruzione '**RELEASE**' per passare formalmente un record. L'istruzione '**RELEASE**' si utilizza e si comporta come l'istruzione '**WRITE**' per i file sequenziali:

```
WRITE  record-name  [ FROM identifier-1 ]
-----                ----
```

Il record è il nome della variabile strutturata corrispondente del file che esegue in pratica l'ordinamento, ovvero quello che nello schema sintattico dell'istruzione '**SORT**' appare come ***file-name-1***.

Listato 72.258. Esempio di acquisizione di record da ordinare attraverso l'inserimento diretto.

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   TEST-SORT-3.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-03-18.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 INPUT-OUTPUT SECTION.
000900*
001000 FILE-CONTROL.
001100*
001200     SELECT FILE-PER-IL-RIORDINO ASSIGN TO "sort.tmp".
001300*
001400 DATA DIVISION.
001500*
001600 FILE SECTION.
001700*
001800 SD  FILE-PER-IL-RIORDINO.
001900*
002000 01  RECORD-PER-IL-RIORDINO.
002100     02  CHIAVE-ORDINAMENTO       PIC X(5).
002200     02  FILLER                   PIC X(5).
002300
002400 WORKING-STORAGE SECTION.
002500 77  EOJ                          PIC 9 VALUE 0.
```

```
002600 77  EOF                              PIC 9 VALUE 0.
002700 77  DATI-INSERITI                    PIC X(10).
002800*
002900 PROCEDURE DIVISION.
003000*------------------------ LIVELLO 0 ----------------
003100 MAIN SECTION.
003200 INIZIO.
003300     SORT FILE-PER-IL-RIORDINO,
003400          ON ASCENDING KEY CHIAVE-ORDINAMENTO,
003500          INPUT PROCEDURE IS INSERIMENTO-DATI,
003600          OUTPUT PROCEDURE IS MOSTRA-FILE-ORDINATO.
003700*
003800     STOP RUN.
003900*
004000*--------------------- SORT-MERGE PROCEDURE ----------
004100 MOSTRA-FILE-ORDINATO SECTION.
004200 INIZIO.
004300     PERFORM MOSTRA-RECORD UNTIL EOF = 1.
004400     GO TO FINE.
004500 MOSTRA-RECORD.
004600     RETURN FILE-PER-IL-RIORDINO RECORD
004700           AT END MOVE 1 TO EOF,
004800                  DISPLAY "FINE DEL FILE ORDINATO".
004900     IF EOF = 0
005000     THEN
005100         DISPLAY RECORD-PER-IL-RIORDINO.
005200 FINE.
005300     EXIT.
005400*-------------------------------------------------
005500 INSERIMENTO-DATI SECTION.
005600 INIZIO.
005700     PERFORM INSERISCI-RECORD UNTIL EOJ = 1.
005800     GO TO FINE.
005900 INSERISCI-RECORD.
```

```
006000      DISPLAY "INSERISCI UN RECORD DA 10 CARATTERI:".
006100      ACCEPT DATI-INSERITI.
006200      IF DATI-INSERITI = SPACES
006300      THEN
006400          MOVE 1 TO EOJ;
006500      ELSE
006600          MOVE DATI-INSERITI TO RECORD-PER-IL-RIORDINO,
006700          RELEASE RECORD-PER-IL-RIORDINO.
006800 FINE.
006900      EXIT.
007000*
```

L'esempio è completo, in quanto anche il risultato del riordino viene gestito tramite una procedura. Nella fase di inserimento dati, si può osservare che un inserimento nullo (pari all'inserimento di tutti spazi), implica la conclusione di quella fase.

72.14 Riferimenti

«

- Christopher Heng, *Free COBOL compilers and interpreters*, http://www.thefreecountry.com/compilers/cobol.shtml
- *Programming manuals and tutorials*, *COBOL*, http://www.theamericanprogrammer.com/programming/manuals.cobol.html
- *MPE/iX and HP e3000 Technical Documentation*, *HP COBOL II/XL*
  - *Programmer's guide*, http://wayback.archive.org/web/2006*/http://docs.hp.com/en/424/31500-90014.pdf
  - *Quick reference guide*, http://wayback.archive.org/web/2006*/http://docs.hp.com/en/425/31500-90015.pdf

  - *Reference manual*, http://wayback.archive.org/web/2006*/http://docs.hp.com/en/426/31500-90013.pdf

- *Compaq COBOL Reference Manual*, http://www.helsinki.fi/atk/unix/dec_manuals/cobv27ua27/cobrm_contents.htm

73.1 Preparazione 1733

73.1.1 Problema del modulo di programmazione 1733

73.1.2 Riepilogo di alcuni concetti importanti del linguaggio 1737

73.1.3 TinyCOBOL 1740

73.1.4 OpenCOBOL 1742

73.2 Esempi elementari 1743

73.2.1 ELM0100: prodotto tra due numeri 1743

73.2.2 ELM0200: prodotto tra due numeri 1744

73.2.3 ELM0300: prodotto tra due numeri 1746

73.2.4 ELM0400: prodotto tra due numeri 1748

73.2.5 ELM0500: prodotto tra due numeri 1751

73.2.6 ELM0600: inserimento dati in un vettore 1754

73.2.7 ELM0700: inserimento dati in un vettore 1757

73.2.8 ELM0800: inserimento dati in un vettore 1760

73.2.9 ELM0900: ricerca sequenziale all'interno di un vettore 1763

73.2.10 ELM1000: ricerca sequenziale all'interno di un vettore 1767

73.2.11 ELM1100: ricerca sequenziale all'interno di un vettore 1770

73.2.12 ELM1300: creazione di un file sequenziale 1774

73.2.13 ELM1400: estensione di un file sequenziale 1777

73.2.14 ELM1500: lettura di un file sequenziale 1780
73.3 Esempi elementari con i file 1785
73.3.1 AGO-83-1: estensione di un file sequenziale 1785
73.3.2 AGO-83-2: lettura sequenziale e ricerca di una chiave 1786
73.3.3 AGO-83-3: estensione di un file relativo 1788
73.3.4 AGO-83-4: lettura di un file relativo ad accesso diretto 1790
73.3.5 AGO-83-5: creazione di un file a indice 1792
73.3.6 AGO-83-6: lettura di un file a indice ad accesso diretto 1794
73.3.7 AGO-83-8: lettura di un file a indice ad accesso dinamico 1796
73.3.8 AGO-83-10: lettura di un file a indice ad accesso dinamico 1799
73.3.9 AGO-83-12: lettura di un file a indice ad accesso dinamico 1802
73.3.10 AGO-83-13: creazione di un file sequenziale con dati da rielaborare 1805
73.3.11 AGO-83-14: lettura e riscrittura di un file sequenziale 1807
73.3.12 AGO-83-15: estensione di un file sequenziale contenente aggiornamenti successivi 1809
73.3.13 AGO-83-16: aggiornamento di un file a indice . . 1811
73.3.14 AGO-83-18: fusione tra due file sequenziali ordinati 1814

73.3.15 AGO-83-20: riordino attraverso la fusione 1817

73.4 Approfondimento: una tecnica per simulare la ricorsione in COBOL 1825

73.4.1 Il concetto di locale e di globale 1825

73.4.2 La ricorsione 1827

73.4.3 Proprietà del linguaggio ricorsivo 1828

73.4.4 Descrizione della tecnica per simulare la ricorsione in COBOL 1828

73.4.5 Torre di Hanoi 1831

73.4.6 Quicksort (ordinamento non decrescente) 1836

73.4.7 Permutazioni 1847

73.4.8 Bibliografia 1854

73.5 Riferimenti 1858

Questo capitolo tratta di casi pratici di programmazione in linguaggio COBOL, con l'intento di recuperare un vecchio lavoro realizzato con il sostegno di Antonio Bernardi, durante i primi anni 1980, utilizzando un elaboratore Burroughs B91.

Figura 73.1. *Mainframe* Burroughs B1900 del 1985: un sogno mai realizzato. La foto originale proviene da *http://www.kiwanja.net/photos.htm* ed è di Ken Banks. La foto viene riprodotta qui con il permesso del suo autore.

73.1 Preparazione

Il linguaggio COBOL nasce quando l'inserimento dei dati in un elaboratore avveniva principalmente attraverso schede perforate, pertanto, da questo derivano delle limitazioni nel modo in cui vanno scritte le sue direttive.

73.1.1 Problema del modulo di programmazione

Il linguaggio COBOL nasce imponendo dei vincoli al modo di utilizzare gli spazi orizzontali nel file del sorgente. Questi vincoli consentivano di amministrare con un certo criterio la procedura di perforazione e riutilizzo delle schede perforate.

Terminata l'era delle schede perforate, i compilatori hanno cominciato a essere più disponibili e ad accettare codice COBOL scritto senza rispettare i vincoli del modulo di programmazione tradizionale (normalmente viene eliminato l'obbligo della numerazione delle righe e l'area in cui è possibile scrivere le istruzioni si estende per un numero indefinito di colonne, cancellando la funzione della zona identificativa del programma); tuttavia, il suggerimento che qui viene dato è di continuare a usare il modello originale, considerata la particolarità del linguaggio di programmazione, che perderebbe la sua logica estetica. Il listato successivo mostra l'esempio di un programma COBOL molto breve, dove si può vedere l'utilizzo delle varie aree secondo il criterio del modulo di programmazione del linguaggio.

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   ELM0100.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 1985-02-12.
000500*
```

```
000600 ENVIRONMENT DIVISION.
000700*
000800 DATA DIVISION.
000900*
001000 WORKING-STORAGE SECTION.                                  WSS-0000
001100 01  A PIC 9(7).                                          WSS-0000
001200 01  B PIC 9(7).                                          WSS-0000
001300 01  C PIC 9(14).                                         WSS-0000
001400*
001500 PROCEDURE DIVISION.
001600*-----------------------------------------------------------------
001700 MAIN.
001800     DISPLAY "MOLTIPLICAZIONE DI DUE NUMERI".
001900     DISPLAY "INSERISCI IL PRIMO ELEMENTO".
002000     ACCEPT A.
002100     DISPLAY "INSERISCI IL SECONDO ELEMENTO".
002200     ACCEPT B.
002300     COMPUTE C = A * B.
002400     DISPLAY C.
002500*
002600     STOP RUN.
002700*
```

Nell'esempio si può osservare: l'uso dell'asterisco nella settima colonna per indicare un commento; la presenza di direttive che iniziano a dalla colonna ottava e di altre che iniziano dalla colonna dodicesima; l'indicazione di un'etichetta distintiva nelle otto colonne finali ('**`WSS-0000`**'), in corrispondenza di alcune righe (probabilmente per ricordare che quella porzione proviene da un altro programma).

Si osservi che quanto appare nelle ultime otto colonne non ha valore per il linguaggio di programmazione, ma rappresenta un modo per individuare gruppi di righe che possono avere qualche tipo di importanza, oppure qualunque altro tipo di annotazione.

Generalmente, i compilatori consentono di specificare con qua-

le formato viene fornito il file sorgente; la scelta è normalmente tra un formato «fisso» (tradizionale), oppure libero (senza vincoli particolari).

Dal momento che attualmente la numerazione delle righe è divenuta puramente un fatto estetico, ci si può aiutare con uno script per rinumerare il sorgente. Il listato successivo mostra uno script molto semplice, che presuppone di ricevere dallo standard input un file sorgente con i numeri di riga, anche se errati, emettendo lo stesso sorgente attraverso lo standard output, ma con una numerazione progressiva uniforme (una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/cobol-line-renumber.sh*).

```
#!/bin/sh
#
# cobol-line-renumber.sh INCREMENT < SOURCE_COB > NEW_SOURCE_COB
#
INCREMENT="$1"
LINE=""
NUMBER="0"
NUMBER_FORMATTED=""
#
while read LINE
do
    NUMBER=$(($NUMBER+$INCREMENT))
    NUMBER_FORMATTED=`printf %000006d $NUMBER`
    LINE=`echo "$LINE" | sed s/^[0-9][0-9][0-9][0-9][0-9][0-9]//`
    LINE="$NUMBER_FORMATTED$LINE"
    echo "$LINE"
done
```

In pratica, supponendo che lo script si chiami '**`cobol-line-renumber.sh`**', si potrebbe usare come nell'esempio seguente:

```
$ cobol-line-renumber.sh < sorgente.cob > rinumerato.cob [Invio]
```

73.1.1.1 Compatibilità con i compilatori

«

I compilatori nati dopo la fine delle schede perforate possono essere più o meno disposti ad accettare la presenza della numerazione delle righe o delle colonne finali di commento. Generalmente questi compilatori consentono di indicare un'opzione che specifica il formato del sorgente; tuttavia si può utilizzare uno script simile a quello seguente, per eliminare le colonne della numerazione delle righe e le colonne descrittive di identificazione del programma:

```
#!/usr/bin/perl
#
# cobol-compile SOURCE_COB SOURCE_COB_NEW
#
use utf8;
binmode (STDOUT, ":utf8");
binmode (STDERR, ":utf8");
binmode (STDIN,  ":utf8");
#
$source=$ARGV[0];
$source_new=$ARGV[1];
$line="";
#
open (SOURCE,     "<:utf8", "$source");
open (SOURCE_NEW, ">:utf8", "$source_new");
#
while ($line = <SOURCE>)
  {
    chomp ($line);
    $line =~ m/^[0-9][0-9][0-9][0-9][0-9][0-9](.*)$/;
    $line = $1;
    if ($line =~ m/^(.{66}).*$/)
```

```
      {
        $line = $1;
      }
    print SOURCE_NEW ("$line\n");
  }
close (SOURCE_NEW);
close (SOURCE);
#
```

Eventualmente, se il problema consistesse soltanto nella rimozione del numero di riga, si potrebbe usare uno script molto più semplice:

```
#!/bin/sh
#
# cobol-compile SOURCE_COB SOURCE_COB_NEW
#
SOURCE="$1"
SOURCE_NEW="$2"
cat $SOURCE | sed s/^[0-9][0-9][0-9][0-9][0-9][0-9]//g > $SOURCE_NEW
```

73.1.2 Riepilogo di alcuni concetti importanti del linguaggio

In generale, le istruzioni del linguaggio COBOL sono da intendere come frasi scritte in inglese, che terminano con un punto fermo. In certe situazioni, si riuniscono più istruzioni in un'unica «frase», che termina con un punto, ma in tal caso, spesso si usa la virgola e il punto e virgola per concludere le istruzioni singole.

Le istruzioni del linguaggio si compongono in linea di massima di parole chiave, costanti letterali e operatori matematici. Le parole chiave sono scritte usando lettere maiuscole (dell'alfabeto inglese) e il trattino normale ('-'). In generale, i simboli che si possono usare nel linguaggio sono abbastanza limitati, con l'eccezione del conte-

nuto delle costanti alfanumeriche letterali, che teoricamente potrebbero contenere qualunque simbolo (escluso quello che si usa come delimitatore) secondo le potenzialità del compilatore particolare.

Tabella 73.6. I simboli disponibili nel linguaggio.

Simboli	Descrizione	Simboli	Descrizione
'0'..'9'	cifre numeriche	'A'..'Z'	lettere maiuscole dell'alfabeto inglese (latino)
' '	spazio		
'+'	segno più	'-'	segno meno o trattino
'*'	asterisco	'/'	barra obliqua
'$'	dollaro o segno di valuta	','	virgola
';'	punto e virgola	'.'	punto fermo
'('	parentesi aperta	')'	parentesi chiusa
'<'	minore	'>'	maggiore

Le parole chiave più importanti del linguaggio sono dei «verbi» imperativi, che descrivono un comando che si vuole sia eseguito. Un gruppo interessante di parole chiave è rappresentato dalle «costanti figurative», che servono a indicare verbalmente delle costanti di uso comune. Per esempio, la parola chiave '**ZERO**' rappresenta uno o più zeri, in base al contesto.

Le stringhe sono delimitate da virgolette (apici doppi) e di solito non sono previste forme di protezione per incorporare le virgolette stesse all'interno delle stringhe: per questo occorre suddividere le stringhe, concatenandole con la costante figurativa '**QUOTE**'.

La gestione numerica del COBOL è speciale rispetto ai linguaggi

di programmazione comuni, perché le variabili vengono dichiarate con la loro dimensione di cifre esatta, stabilendo anche la quantità di decimali e il modo in cui l'informazione deve essere gestita. In pratica, si stabilisce il modo in cui il valore deve essere rappresentato, lasciando al compilatore il compito di eseguire ogni volta tutte le conversioni necessarie. Sotto questo aspetto, un programma COBOL ha una gestione per i valori numerici molto pesante, quindi più lenta rispetto ad altri linguaggi, dove i valori numerici sono gestiti in base alle caratteristiche fisiche della CPU e le conversioni di tipo devono essere dichiarate esplicitamente.

Le variabili usate nel linguaggio sono sempre globali e come tali vanno dichiarate in una posizione apposita. Tali variabili, salvo situazioni eccezionali, fanno sempre parte di un record, inteso come una raccolta di campi di informazioni. Questa gestione particolare costringe a stabilire esattamente le dimensioni che ogni informazione deve avere se registrata nella memoria di massa (dischi, nastri o altro) o se stampata. In un certo senso, questa caratteristica può impedire o rendere difficile l'uso di una forme di codifica dei caratteri che preveda una dimensione variabile degli stessi, considerato che i record possono essere rimappati, trattando anche valori numerici come insiemi di cifre letterali.

Questo particolare, che non è affatto di poco conto, suggerisce di usare il linguaggio per gestire dati rappresentabili con il codice ASCII tradizionale, ovvero con i primi 127 punti di codifica (da U+0000 a U+007F). Naturalmente sono disponibili compilatori che permettono di superare questo problema, ma in tal caso occorre verificare come vengono gestiti effettivamente i dati.

Le istruzioni COBOL possono essere scritte usando più righe, avendo l'accortezza di continuare a partire dall'area «B»; in generale non c'è bisogno di indicare esplicitamente che l'istruzione sta continuando nella riga successiva, perché si usa il punto fermo per riconoscere la loro conclusione. Tuttavia, in situazioni eccezionali, si può spezzare una parola chiave o anche una stringa letterale; in tal caso, nella settima colonna delle riga che continua, va inserito il segno '**-**', inoltre, se si tratta di una stringa, la sua ripresa va iniziata nuovamente con le virgolette. A ogni modo, considerato che difficilmente si devono scrivere parole chiave molto lunghe e che le stringhe letterali si possono concatenare, è auspicabile che la continuazione nella riga successiva con l'indicatore nella settima colonna sia evitata del tutto.

I commenti nel sorgente si indicano inserendo un asterisco nella settima colonna; se invece si mette una barra obliqua ('**/**') si vuole richiedere un salto pagina, in fase di stampa, ammesso che il compilatore preveda questo.

73.1.3 TinyCOBOL

«

TinyCOBOL[1] è un compilatore COBOL che tende alla conformità con gli standard del 1985. Come per ogni compilatore COBOL ci sono delle differenze rispetto al linguaggio «standard», in par-

ticolare è disponibile la possibilità di recepire gli argomenti della riga di comando e di accedere ai flussi standard dei sistemi Unix (standard input, standard output e standard error).

La compilazione di un programma si ottiene attraverso il programma '**htcobol**', che, salvo l'uso dell'opzione '**-F**', si aspetta di trovare un sorgente senza numerazione delle righe e senza il blocco descrittivo finale delle colonne da 73 a 80. In pratica, ciò consentirebbe si disporre di un'area B (per le istruzioni) molto più ampia.

```
htcobol [opzioni] file_sorgente_cobol
```

Il programma '**htcobol**' si aspetta che il file sorgente abbia un nome con un'estensione '`.cob`' e, in tal caso, l'estensione può anche essere omessa. Se non si specificano opzioni, si ottiene un file eseguibile con lo stesso nome del sorgente, ma senza l'estensione '`.cob`'.

Tabella 73.7. Alcune opzioni.

Opzione	Descrizione
`-o` ***file***	Richiede che il file generato dalla compilazione abbia il nome stabilito dall'argomento dell'opzione.
`-X`	Richiede che il file sorgente sia scritto senza numerazione delle righe e senza commenti nelle colonne da 73 a 80; tuttavia questa è la modalità di funzionamento predefinita.
`-F`	Richiede che il file sorgente sia scritto secondo il formato tradizionale (con la numerazione delle righe e con il limite dell'area «B»).

Vengono mostrati alcuni esempi.

- `$` **`htcobol -F esempio.cob`**[*Invio*]

Compila il programma ‘`esempio.cob`’, generando il file eseguibile ‘**`esempio`**’. Se non vengono riscontrati errori, la compilazione non genera alcun messaggio.

- `$` **`htcobol -F -o programma esempio.cob`**[*Invio*]

  Compila il programma ‘`esempio.cob`’, generando il file eseguibile ‘**`programma`**’. Se non vengono riscontrati errori, la compilazione non genera alcun messaggio.

73.1.4 OpenCOBOL

«

OpenCOBOL[2] è un compilatore COBOL che genera codice in linguaggio C e si avvale di GCC per arrivare a produrre il file eseguibile finale. In generale si utilizza per la compilazione il programma ‘**`cobc`**’ che si prende cura di tutti i passaggi necessari:

```
cobc [opzioni] file_sorgente_cobol
```

Tabella 73.8. Alcune opzioni.

Opzione	Descrizione
`-free`	Richiede che il file sorgente sia scritto in formato «libero» (senza i vincoli della numerazione delle righe e senza commenti nelle colonne da 73 a 80.
`-fixed`	Richiede che il file sorgente sia scritto secondo il formato tradizionale (con la numerazione delle righe e con il limite tradizionale dell’area «B»).

L’esempio seguente compila il file ‘`esempio.cob`’ e genera il file eseguibile ‘**`esempio`**’:

`$` **`cobc esempio.cob`**[*Invio*]

73.2 Esempi elementari

Qui si raccolgono alcuni esempi elementari di programmi COBOL, risalenti a un lavoro didattico del 1985. Salvo dove indicato in maniera differente, gli esempi mostrati funzionano regolarmente se compilati con OpenCOBOL 0.31.

73.2.1 ELM0100: prodotto tra due numeri

Variabili

‘**A**’ è il moltiplicando;

‘**B**’ è il moltiplicatore;

‘**C**’ è il risultato.

Descrizione

Il calcolo viene eseguito attraverso l’istruzione ‘**COMPUTE**’.

Paragrafo ‘**MAIN**’

Il programma si svolge unicamente all’interno di questo paragrafo. Il programma riceve dall’esterno i valori per le variabili ‘**A**’ e ‘**B**’, esegue il prodotto tramite l’istruzione ‘**COMPUTE**’ mettendo il risultato nella variabile ‘**C**’.

Viene visualizzato il contenuto della variabile ‘**C**’ con l’istruzione ‘**DISPLAY**’.

Il programma si arresta perché incontra l’istruzione ‘**STOP RUN**’.

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/ELM0100.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   ELM0100.
```

```
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 1985-02-12.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 DATA DIVISION.
000900*
001000 WORKING-STORAGE SECTION.
001100 01  A PIC 9(7).
001200 01  B PIC 9(7).
001300 01  C PIC 9(14).
001400*
001500 PROCEDURE DIVISION.
001600*-----------------------------------------------------
001700 MAIN.
001800     DISPLAY "MOLTIPLICAZIONE DI DUE NUMERI".
001900     DISPLAY "INSERISCI IL PRIMO ELEMENTO".
002000     ACCEPT A.
002100     DISPLAY "INSERISCI IL SECONDO ELEMENTO".
002200     ACCEPT B.
002300     COMPUTE C = A * B.
002400     DISPLAY C.
002500*
002600     STOP RUN.
002700*
```

73.2.2 ELM0200: prodotto tra due numeri

«

Variabili

‘**A**’ è il moltiplicando;

‘**B**’ è il moltiplicatore;

‘**C**’ è il risultato; questa variabile viene inizializzata a zero in fase di dichiarazione.

Descrizione

Il calcolo viene eseguito sommando alla variabile ‘**C**’ la variabile ‘**A**’ per ‘**B**’ volte.

Paragrafo ‘**MAIN**’

Il programma riceve dall’esterno i valori per le variabili ‘**A**’ e ‘**B**’. Attraverso l’istruzione ‘**PERFORM**’ viene eseguito il paragrafo ‘**SOMMA**’ per ‘**B**’ volte; al termine di questo ciclo il risultato della moltiplicazione si trova nella variabile ‘**C**’, che viene visualizzato con l’istruzione ‘**DISPLAY**’.

Il programma si arresta perché incontra l’istruzione ‘**STOP RUN**’.

Paragrafo ‘**SOMMA**’

Il paragrafo somma al contenuto della variabile ‘**C**’ il contenuto della variabile ‘**A**’. Dal momento che questo paragrafo viene eseguito ‘**B**’ volte, la variabile ‘**C**’ finisce con il contenere il risultato del prodotto di «A×B».

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/ELM0200.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   ELM0200.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 1985-02-14.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 DATA DIVISION.
000900*
```

```
001000 WORKING-STORAGE SECTION.
001100 01  A PIC 9(7).
001200 01  B PIC 9(7).
001300 01  C PIC 9(14)   VALUE ZERO.
001400*
001500 PROCEDURE DIVISION.
001600*------------------------ LIVELLO 0 -----------------
001700 MAIN.
001800     DISPLAY "MOLTIPLICAZIONE DI DUE NUMERI".
001900     DISPLAY "INSERISCI IL PRIMO ELEMENTO".
002000     ACCEPT A.
002100     DISPLAY "INSERISCI IL SECONDO ELEMENTO".
002200     ACCEPT B.
002300     PERFORM SOMMA B TIMES.
002400     DISPLAY C.
002500*
002600     STOP RUN.
002700*------------------------ LIVELLO 1 -----------------
002800 SOMMA.
002900     COMPUTE C = C + A.
003000*
```

73.2.3 ELM0300: prodotto tra due numeri

«

Variabili

‘**A**’ è il moltiplicando;

‘**B**’ è il moltiplicatore;

‘**C**’ è il risultato.

Descrizione

Il calcolo viene eseguito sommando alla variabile ‘**C**’ la variabile ‘**A**’ per ‘**B**’ volte. Per ogni esecuzione di tale somma, la variabile

‘**B**’ viene diminuita di una unità, cosicché il ciclo delle somme viene arrestato quando ‘**B**’ è ormai a zero.

Paragrafo ‘**MAIN**’

Vengono ricevuti dall’esterno i valori per le variabili ‘**A**’ e ‘**B**’. Viene eseguito tramite l’istruzione ‘**PERFORM**’ il paragrafo ‘**SOMMA**’ fino a quando la variabile ‘**B**’ raggiunge lo zero. A quel punto la variabile ‘**C**’ contiene il risultato del prodotto, che viene visualizzato con l’istruzione ‘**DISPLAY**’.

Il programma si arresta perché incontra l’istruzione ‘**STOP RUN**’.

Paragrafo ‘**SOMMA**’

Inizialmente viene decrementato di una unità il contenuto della variabile ‘**B**’, quindi viene sommato al contenuto di ‘**C**’ il valore di ‘**A**’.

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/ELM0300.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   ELM0300.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 1985-04-13.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 DATA DIVISION.
000900*
001000 WORKING-STORAGE SECTION.
001100 01  A PIC 9(7).
001200 01  B PIC 9(7).
001300 01  C PIC 9(14)   VALUE ZERO.
001400*
001500 PROCEDURE DIVISION.
```

```
001600*------------------------ LIVELLO 0 ----------------
001700 MAIN.
001800     DISPLAY "MOLTIPLICAZIONE DI DUE NUMERI".
001900     DISPLAY "INSERISCI IL PRIMO ELEMENTO".
002000     ACCEPT A.
002100     DISPLAY "INSERISCI IL SECONDO ELEMENTO".
002200     ACCEPT B.
002300     PERFORM SOMMA UNTIL B = 0.
002400     DISPLAY C.
002500*
002600     STOP RUN.
002700*------------------------ LIVELLO 1 ----------------
002800 SOMMA.
002900     COMPUTE B = B - 1.
003000     COMPUTE C = C + A.
003100*
```

73.2.4 ELM0400: prodotto tra due numeri

«

Variabili

‘**A**’ è il moltiplicando;

‘**B**’ è il moltiplicatore;

‘**C**’ è il risultato;

‘**EOJ**’ quando assume il valore 1 il programma si arresta;

‘**RISPOSTA**’ è la variabile che riceve la risposta, un ‘**SI**’ o un ‘**NO**’, per la continuazione o meno con un altro calcolo.

Descrizione

Il calcolo viene eseguito sommando alla variabile ‘**C**’ la variabile ‘**A**’ per ‘**B**’ volte. Per ogni esecuzione di tale somma, la variabile

‘**B**’ viene diminuita di una unità, cosicché il ciclo delle somme viene arrestato quando ‘**B**’ è ormai a zero.

Il programma si arresta solo se gli viene dato un comando apposito, altrimenti continua a richiedere altri dati per l’esecuzione di un altro prodotto.

Paragrafo ‘**MAIN**’

Vengono ricevuti dall’esterno i valori per le variabili ‘**A**’ e ‘**B**’ tramite il paragrafo ‘**INSERIMENTO-DATI**’.

Viene eseguito il paragrafo ‘**LAVORO**’ ripetutamente, terminando il ciclo quando la variabile ‘**EOJ**’ contiene il valore uno.

Il programma si arresta perché incontra l’istruzione ‘**STOP RUN**’.

Paragrafo ‘**LAVORO**’

Viene eseguito tramite l’istruzione ‘**PERFORM**’ il paragrafo ‘**SOMMA**’ ripetutamente, terminando il ciclo quando la variabile ‘**B**’ contiene il valore zero. A quel punto, la variabile ‘**C**’ contiene il risultato del prodotto, che viene visualizzato con l’istruzione ‘**DISPLAY**’.

Il programma riceve dall’esterno una parola: un ‘**SI**’ o un ‘**NO**’; se viene fornita la stringa ‘**SI**’ (scritta con lettere maiuscole) il programma azzera il contenuto della variabile ‘**C**’ ed esegue il paragrafo ‘**INSERIMENTO-DATI**’, altrimenti, viene messo il valore uno nella variabile ‘**EOJ**’.

Paragrafo ‘**INSERIMENTO-DATI**’

Il paragrafo riceve dall’esterno i valori per le variabili ‘**A**’ e ‘**B**’.

Paragrafo '**SOMMA**'

Inizialmente viene decrementato di una unità il contenuto della variabile '**B**', quindi viene sommato al contenuto di '**C**' il valore di '**A**'.

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/ELM0400.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   ELM0400.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 1985-02-14.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 DATA DIVISION.
000900*
001000 WORKING-STORAGE SECTION.
001100 01  A         PIC 9(7).
001200 01  B         PIC 9(7).
001300 01  C         PIC 9(14)   VALUE ZERO.
001400 01  EOJ       PIC 9       VALUE ZERO.
001500 01  RISPOSTA PIC XX.
001600*
001700 PROCEDURE DIVISION.
001800*------------------------ LIVELLO 0 -----------------
001900 MAIN.
002000     PERFORM INSERIMENTO-DATI.
002100     PERFORM LAVORO UNTIL EOJ = 1.
002200*
002300     STOP RUN.
002400*------------------------ LIVELLO 1 -----------------
002500 LAVORO.
002600     PERFORM SOMMA UNTIL B = 0.
002700     DISPLAY C.
```

```
002800*
002900     DISPLAY "VUOI CONTINUARE? SI O NO".
003000     ACCEPT RISPOSTA.
003100*
003200     IF RISPOSTA = "SI"
003300       THEN
003400           MOVE ZERO TO C,
003500           PERFORM INSERIMENTO-DATI;
003600       ELSE
003700           MOVE 1 TO EOJ.
003800*------------------------ LIVELLO 2 -----------------
003900 INSERIMENTO-DATI.
004000     DISPLAY "INSERISCI IL PRIMO ELEMENTO".
004100     ACCEPT A.
004200     DISPLAY "INSERISCI IL SECONDO ELEMENTO".
004300     ACCEPT B.
004400*-------------------------------------------------
004500 SOMMA.
004600     COMPUTE B = B - 1.
004700     COMPUTE C = C + A.
004800*
```

73.2.5 ELM0500: prodotto tra due numeri

«

Variabili

‘**A**’ è il moltiplicando;

‘**B**’ è il moltiplicatore;

‘**C**’ è il risultato;

‘**EOJ**’ quando assume il valore 1 il programma si arresta;

‘**RISPOSTA**’ è la variabile che riceve la risposta, un ‘**SI**’ o un ‘**NO**’, per la continuazione o meno con un altro calcolo.

Descrizione

Il calcolo viene eseguito sommando alla variabile '**C**' la variabile '**A**' per '**B**' volte. Il controllo di questa somma viene effettuato da un ciclo '**PERFORM VARYING**' che decrementa di una unità la variabile '**B**', partendo dal suo valore iniziale, fino a quando si riduce a zero, nel qual caso il ciclo si arresta.

Paragrafo '**MAIN**'

Vengono ricevuti dall'esterno i valori per le variabili '**A**' e '**B**' tramite il paragrafo '**INSERIMENTO-DATI**'.

Viene eseguito il paragrafo '**LAVORO**' ripetutamente, terminando il ciclo quando la variabile '**EOJ**' contiene il valore uno.

Il programma si arresta perché incontra l'istruzione '**STOP RUN**'.

Paragrafo '**LAVORO**'

Viene eseguito tramite l'istruzione '**PERFORM**' il paragrafo '**SOMMA**' ripetutamente, decrementando il valore della variabile '**B**', fino a zero, quando il ciclo termina. A quel punto, la variabile '**C**' contiene il risultato del prodotto, che viene visualizzato con l'istruzione '**DISPLAY**'.

Il programma riceve dall'esterno una parola: un '**SI**' o un '**NO**'; se viene fornita la stringa '**SI**' (scritta con lettere maiuscole) il programma azzera il contenuto della variabile '**C**' ed esegue il paragrafo '**INSERIMENTO-DATI**', altrimenti, viene messo il valore uno nella variabile '**EOJ**'.

Paragrafo '**INSERIMENTO-DATI**'

Il paragrafo riceve dall'esterno i valori per le variabili '**A**' e '**B**'.

Paragrafo ‘**SOMMA**’

Viene sommato al contenuto di ‘**C**’ il valore di ‘**A**’.

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/ELM0500.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   ELM0500.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 1985-02-14.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 DATA DIVISION.
000900*
001000 WORKING-STORAGE SECTION.
001100 01  A         PIC 9(7).
001200 01  B         PIC 9(7).
001300 01  C         PIC 9(14)   VALUE ZERO.
001400 01  EOJ       PIC 9       VALUE ZERO.
001500 01  RISPOSTA PIC XX.
001600*
001700 PROCEDURE DIVISION.
001800*------------------------ LIVELLO 0 -----------------
001900 MAIN.
002000     PERFORM INSERIMENTO-DATI.
002100     PERFORM LAVORO UNTIL EOJ = 1.
002200*
002300     STOP RUN.
002400*------------------------ LIVELLO 1 -----------------
002500 LAVORO.
002600     PERFORM SOMMA VARYING B FROM B BY -1 UNTIL B = 0.
002700     DISPLAY C.
002800*
002900     DISPLAY "VUOI CONTINUARE? SI O NO".
```

```
003000     ACCEPT RISPOSTA.
003100*
003200     IF RISPOSTA = "SI"
003300       THEN
003400           MOVE ZERO TO C,
003500           PERFORM INSERIMENTO-DATI;
003600       ELSE
003700           MOVE 1 TO EOJ.
003800*------------------------ LIVELLO 2 ----------------
003900 INSERIMENTO-DATI.
004000     DISPLAY "INSERISCI IL PRIMO ELEMENTO".
004100     ACCEPT A.
004200     DISPLAY "INSERISCI IL SECONDO ELEMENTO".
004300     ACCEPT B.
004400*-----------------------------------------------------
004500 SOMMA.
004600     COMPUTE C = C + A.
004700*
```

73.2.6 ELM0600: inserimento dati in un vettore

«

Variabili

‘**RECORD-ELEMENTI**’ è una variabile che si scompone in un array;

‘**ELEMENTO**’ è l’array che costituisce ‘**RECORD-ELEMENTI**’;

‘**INDICE**’ è l’indice usato per scandire gli elementi;

‘**EOJ**’ quando assume il valore 1 il programma si arresta;

‘**RISPOSTA**’ è la variabile che riceve la risposta, un ‘**SI**’ o un ‘**NO**’, per la continuazione o meno con un altro calcolo.

Descrizione

Il programma esegue semplicemente un inserimento di dati all'interno degli elementi dell'array, con un accesso libero (bisogna ricordare che l'indice del primo elemento è uno), specificando prima l'indice e poi il valore (il carattere) da attribuire all'elemento.

Paragrafo '**MAIN**'

Viene eseguito una volta il paragrafo '**INSERIMENTO-INDICE**', che serve a ricevere il valore dell'indice di inserimento dall'utente.

Viene eseguito il paragrafo '**LAVORO**' ripetutamente, terminando il ciclo quando la variabile '**EOJ**' contiene il valore uno.

Viene visualizzato il valore di tutta la variabile '**RECORD-ELEMENTI**', attraverso l'istruzione '**DISPLAY**'.

Il programma si arresta perché incontra l'istruzione '**STOP RUN**'.

Paragrafo '**LAVORO**'

Il programma riceve dall'esterno il valore per '**ELEMENTO(INDICE)**'.

Il programma riceve dall'esterno l'assenso o il dissenso riguardo alla continuazione dell'esecuzione; se l'intenzione è di proseguire, viene eseguito il paragrafo '**INSERIMENTO-INDICE**', altrimenti viene messo il valore uno nella variabile '**EOJ**'.

Paragrafo '**INSERIMENTO-INDICE**'

Il programma riceve dall'esterno il valore per la variabile '**INDICE**'.

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/ELM0600.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   ELM0600.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 1985-02-14.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 DATA DIVISION.
000900*
001000 WORKING-STORAGE SECTION.
001100 01  RECORD-ELEMENTI.
001200     02  ELEMENTO  PIC X   OCCURS 9 TIMES.
001300 01  INDICE        PIC 9.
001400 01  EOJ           PIC 9   VALUE ZERO.
001500 01  RISPOSTA      PIC XX.
001600*
001700 PROCEDURE DIVISION.
001800*------------------------ LIVELLO 0 -----------------
001900 MAIN.
002000     PERFORM INSERIMENTO-INDICE.
002100     PERFORM LAVORO UNTIL EOJ = 1.
002200     DISPLAY RECORD-ELEMENTI.
002300*
002400     STOP RUN.
002500*------------------------ LIVELLO 1 -----------------
002600 LAVORO.
002700     DISPLAY "INSERISCI I DATI DI UN ELEMENTO ",
002750             "(UN SOLO CARATTERE)".
002800     ACCEPT ELEMENTO(INDICE).
002900*
003000     DISPLAY "VUOI CONTINUARE? SI O NO".
003100     ACCEPT RISPOSTA.
003200*
```

```
003300     IF RISPOSTA = "SI"
003400       THEN
003500           PERFORM INSERIMENTO-INDICE;
003600       ELSE
003700           MOVE 1 TO EOJ.
003800*------------------------ LIVELLO 2 ----------------
003900 INSERIMENTO-INDICE.
004000     DISPLAY "INSERISCI L'INDICE".
004100     ACCEPT INDICE.
004200*
```

73.2.7 ELM0700: inserimento dati in un vettore

Variabili

‘**RECORD-ELEMENTI**’ è una variabile che si scompone in un array;

‘**ELEMENTO**’ è l’array che costituisce ‘**RECORD-ELEMENTI**’;

‘**INDICE**’ è l’indice usato per scandire gli elementi;

‘**EOJ**’ quando assume il valore 1 il programma si arresta;

‘**RISPOSTA**’ è la variabile che riceve la risposta, un ‘**SI**’ o un ‘**NO**’, per la continuazione o meno con un altro calcolo.

Descrizione

Il programma esegue semplicemente un inserimento di dati all’interno degli elementi dell’array, con un accesso libero (bisogna ricordare che l’indice del primo elemento è uno), specificando prima l’indice e poi il valore (il carattere) da attribuire all’elemento.

Se l’indice che si inserisce è zero, viene richiesto nuovamente di fornire un dato valido.

Paragrafo '**MAIN**'

Viene eseguito paragrafo '**INSERIMENTO-INDICE**', che serve a ricevere il valore dell'indice di inserimento dall'utente, ripetendo l'operazione se il valore fornito è minore o uguale a zero.

Viene eseguito il paragrafo '**LAVORO**' ripetutamente, terminando il ciclo quando la variabile '**EOJ**' contiene il valore uno.

Viene visualizzato il valore di tutta la variabile '**RECORD-ELEMENTI**', attraverso l'istruzione '**DISPLAY**'.

Il programma si arresta perché incontra l'istruzione '**STOP RUN**'.

Paragrafo '**LAVORO**'

Il programma riceve dall'esterno il valore per '**ELEMENTO(INDICE)**'.

Il programma riceve dall'esterno l'assenso o il dissenso riguardo alla continuazione dell'esecuzione; se l'intenzione è di proseguire, dopo l'azzeramento della variabile '**INDICE**' viene eseguito il paragrafo '**INSERIMENTO-INDICE**', ripetutamente, ponendo come condizione di conclusione il fatto che la variabile '**INDICE**' abbia un valore maggiore di zero. Se invece l'utente rinuncia a proseguire, viene messo il valore uno nella variabile '**EOJ**'.

Paragrafo '**INSERIMENTO-INDICE**'

Il programma riceve dall'esterno il valore per la variabile '**INDICE**'.

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/ELM0700.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   ELM0700.
000300 AUTHOR.       DANIELE GIACOMINI.
```

```
000400 DATE-WRITTEN. 1985-02-14.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 DATA DIVISION.
000900*
001000 WORKING-STORAGE SECTION.
001100 01  RECORD-ELEMENTI.
001200     02  ELEMENTO  PIC X   OCCURS 9 TIMES.
001300 01  INDICE        PIC 9.
001400 01  EOJ           PIC 9   VALUE ZERO.
001500 01  RISPOSTA      PIC XX.
001600*
001700 PROCEDURE DIVISION.
001800*------------------------ LIVELLO 0 -----------------
001900 MAIN.
002000     PERFORM INSERIMENTO-INDICE UNTIL INDICE > ZERO.
002100     PERFORM LAVORO UNTIL EOJ = 1.
002200     DISPLAY RECORD-ELEMENTI.
002300*
002400     STOP RUN.
002500*------------------------ LIVELLO 1 -----------------
002600 LAVORO.
002700     DISPLAY "INSERISCI I DATI DI UN ELEMENTO ",
002750             "(UN SOLO CARATTERE)".
002800     ACCEPT ELEMENTO(INDICE).
002900*
003000     DISPLAY "VUOI CONTINUARE? SI O NO".
003100     ACCEPT RISPOSTA.
003200*
003300     IF RISPOSTA = "SI"
003400       THEN
003500           MOVE ZERO TO INDICE,
003600           PERFORM INSERIMENTO-INDICE
```

```
003650                    UNTIL INDICE > ZERO;
003700         ELSE
003800             MOVE 1 TO EOJ.
003900*------------------------- LIVELLO 2 -----------------
004000 INSERIMENTO-INDICE.
004100     DISPLAY "INSERISCI L'INDICE".
004200     ACCEPT INDICE.
004300*
```

73.2.8 ELM0800: inserimento dati in un vettore

«

Variabili

'**RECORD-ELEMENTI**' è una variabile che si scompone in un array;

'**ELEMENTO**' è l'array che costituisce '**RECORD-ELEMENTI**';

'**INDICE**' è l'indice usato per scandire gli elementi;

'**EOJ**' quando assume il valore 1 il programma si arresta;

'**RISPOSTA**' è la variabile che riceve la risposta, un '**SI**' o un '**NO**', per la continuazione o meno con un altro calcolo.

Descrizione

Il programma esegue semplicemente un inserimento di dati all'interno degli elementi dell'array, con un accesso libero (bisogna ricordare che l'indice del primo elemento è uno), specificando prima l'indice e poi il valore (il carattere) da attribuire all'elemento.

Se l'indice che si inserisce è zero, viene richiesto nuovamente di fornire un dato valido.

Paragrafo ‘**MAIN**’

Viene eseguito paragrafo ‘**INSERIMENTO-INDICE**’, che serve a ricevere il valore dell’indice di inserimento dall’utente.

Viene eseguito il paragrafo ‘**LAVORO**’ ripetutamente, terminando il ciclo quando la variabile ‘**EOJ**’ contiene il valore uno.

Viene visualizzato il valore di tutta la variabile ‘**RECORD-ELEMENTI**’, attraverso l’istruzione ‘**DISPLAY**’.

Il programma si arresta perché incontra l’istruzione ‘**STOP RUN**’.

Paragrafo ‘**LAVORO**’

Il programma riceve dall’esterno il valore per ‘**ELEMENTO(INDICE)**’.

Il programma riceve dall’esterno l’assenso o il dissenso riguardo alla continuazione dell’esecuzione; se l’intenzione è di proseguire viene eseguito il paragrafo ‘**INSERIMENTO-INDICE**’, in caso contrario, viene messo il valore uno nella variabile ‘**EOJ**’.

Paragrafo ‘**INSERIMENTO-INDICE**’

Il programma riceve dall’esterno il valore per la variabile ‘**INDICE**’, quindi controlla che questo sia diverso da zero; in caso contrario, si ha una chiamata dello stesso paragrafo, in modo ricorsivo.

A causa della caratteristica ricorsiva del paragrafo ‘**INSERIMENTO-INDICE**’, nel programma originale era riportato in un commento: «attenzione! può essere nocivo».

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/ELM0800.cob* .

```
000100 IDENTIFICATION DIVISION.
```

```
000200 PROGRAM-ID.   ELM0800.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 1985-02-14.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 DATA DIVISION.
000900*
001000 WORKING-STORAGE SECTION.
001100 01  RECORD-ELEMENTI.
001200     02  ELEMENTO  PIC X   OCCURS 9 TIMES.
001300 01  INDICE        PIC 9.
001400 01  EOJ           PIC 9   VALUE ZERO.
001500 01  RISPOSTA      PIC XX.
001600*
001700 PROCEDURE DIVISION.
001800*------------------------ LIVELLO 0 -----------------
001900 MAIN.
002000     PERFORM INSERIMENTO-INDICE.
002100     PERFORM LAVORO UNTIL EOJ = 1.
002200     DISPLAY RECORD-ELEMENTI.
002300*
002400     STOP RUN.
002500*------------------------ LIVELLO 1 -----------------
002600 LAVORO.
002700     DISPLAY "INSERISCI I DATI DI UN ELEMENTO",
002800             " (UN SOLO CARATTERE)".
002900     ACCEPT ELEMENTO(INDICE).
003000*
003100     DISPLAY "VUOI CONTINUARE? SI O NO".
003200     ACCEPT RISPOSTA.
003300*
003400     IF RISPOSTA = "SI"
003500       THEN
```

```
003600           PERFORM INSERIMENTO-INDICE;
003700       ELSE
003800           MOVE 1 TO EOJ.
003900*------------------------ LIVELLO 2 ----------------
004000 INSERIMENTO-INDICE.
004100     DISPLAY "INSERISCI L'INDICE".
004200     ACCEPT INDICE.
004300     IF INDICE = 0
004400       THEN
004500           PERFORM INSERIMENTO-INDICE.
004600*
```

73.2.9 ELM0900: ricerca sequenziale all'interno di un vettore

Variabili

'**RECORD-ELEMENTI**' è una variabile usata per accogliere una stringa;

'**ELEMENTO**' è un array che scompone '**RECORD-ELEMENTI**' in caratteri singoli;

'**POSIZIONE**' è l'indice usato per scandire gli elementi della stringa;

'**EOJ**' quando assume il valore 1 il programma si arresta;

'**RISPOSTA**' è la variabile che riceve la risposta, un '**SI**' o un '**NO**', per la continuazione o meno con un altro calcolo;

'**LETTERA**' è la variabile che contiene la lettera da cercare nella stringa.

Descrizione

Il programma riceve dall'esterno il contenuto di una stringa e di una lettera che dovrebbe essere contenuta nella stringa stessa;

successivamente il programma scandisce la stringa come vettore di caratteri e individua la prima posizione in cui appare la lettera cercata.

Paragrafo '**MAIN**'

Viene eseguito paragrafo '**INSERIMENTO-DATI**'.

Viene eseguito il paragrafo '**LAVORO**' ripetutamente, terminando il ciclo quando la variabile '**EOJ**' contiene il valore uno.

Il programma si arresta perché incontra l'istruzione '**STOP RUN**'.

Paragrafo '**LAVORO**'

Il programma esegue il paragrafo '**RICERCA**'.

A questo punto la variabile '**POSIZIONE**' contiene la posizione della lettera contenuta nella variabile '**LETTERA**' e viene visualizzata.

Il programma riceve dall'esterno l'assenso o il dissenso riguardo alla continuazione dell'esecuzione; se l'intenzione è di proseguire, viene eseguito il paragrafo '**INSERIMENTO-DATI**', in caso contrario, viene messo il valore uno nella variabile '**EOJ**'.

Paragrafo '**INSERIMENTO-DATI**'

Il programma riceve dall'esterno una stringa da inserire nella variabile '**RECORD-ELEMENTI**' e la lettera da ricercare nella stringa.

Paragrafo '**RICERCA**'

Viene eseguito un paragrafo che non esegue alcunché (l'istruzione '**EXIT**') scandendo l'indice '**POSIZIONE**' a partire da uno, con passo unitario, terminando quando il contenuto di '**ELEMENTO(POSIZIONE)**' coincide con il valore di '**LETTERA**',

ovvero quando la posizione della lettera nella stringa è stata trovata.

In pratica, il paragrafo '**EXIT-PARAGRAPH**' è una scusa per utilizzare la scansione dell'istruzione '**PERFORM VARYING**'.

Paragrafo '**EXIT-PARAGRAPH**'

Il paragrafo non fa alcunché.

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/ELM0900.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   ELM0900.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 1985-02-15.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 DATA DIVISION.
000900*
001000 WORKING-STORAGE SECTION.
001100 01  RECORD-ELEMENTI.
001200     02  ELEMENTO  PIC X   OCCURS 60 TIMES.
001300 01  POSIZIONE     PIC 99.
001500 01  EOJ           PIC 9   VALUE ZERO.
001600 01  RISPOSTA      PIC XX.
001700 01  LETTERA       PIC X.
001800*
001900 PROCEDURE DIVISION.
002000*------------------------ LIVELLO 0 -----------------
002100 MAIN.
002200     PERFORM INSERIMENTO-DATI.
002300     PERFORM LAVORO UNTIL EOJ = 1.
002400*
002500     STOP RUN.
```

```
002600*------------------------ LIVELLO 1 -----------------
002700 LAVORO.
002800     PERFORM RICERCA.
002900     DISPLAY "LA LETTERA ", LETTERA,
003000             " E' NELLA POSIZIONE ", POSIZIONE.
003100*
003200     DISPLAY "VUOI CONTINUARE? SI O NO".
003300     ACCEPT RISPOSTA.
003400*
003500     IF RISPOSTA = "SI"
003600       THEN
003700           PERFORM INSERIMENTO-DATI;
003800       ELSE
003900           MOVE 1 TO EOJ.
004000*------------------------ LIVELLO 2 -----------------
004100 INSERIMENTO-DATI.
004200     DISPLAY "INSERISCI LA FRASE".
004300     ACCEPT RECORD-ELEMENTI.
004400*
004500     DISPLAY "INSERISCI LA LETTERA DA TROVARE".
004600     ACCEPT LETTERA.
004700*-------------------------------------------------
004800 RICERCA.
004900     PERFORM EXIT-PARAGRAPH
005000             VARYING POSIZIONE FROM 1 BY 1
005100             UNTIL ELEMENTO(POSIZIONE) = LETTERA.
005200*------------------------ LIVELLO 3 -----------------
005300 EXIT-PARAGRAPH.
005400     EXIT.
005500*
```

73.2.10 ELM1000: ricerca sequenziale all'interno di un vettore

«

Variabili

'**RECORD-ELEMENTI**' è una variabile usata per accogliere una stringa;

'**ELEMENTO**' è un array che scompone '**RECORD-ELEMENTI**' in caratteri singoli;

'**POSIZIONE**' è l'indice usato per scandire gli elementi della stringa;

'**EOJ**' quando assume il valore 1 il programma si arresta;

'**RISPOSTA**' è la variabile che riceve la risposta, un '**SI**' o un '**NO**', per la continuazione o meno con un altro calcolo;

'**LETTERA**' è la variabile che contiene la lettera da cercare nella stringa.

Descrizione

Il programma riceve dall'esterno il contenuto di una stringa e di una lettera che dovrebbe essere contenuta nella stringa stessa; successivamente il programma scandisce la stringa come vettore di caratteri e individua la prima posizione in cui appare la lettera cercata.

Rispetto a '**ELM0900**' la scansione della stringa si arresta anche se non viene trovata alcuna corrispondenza.

Paragrafo '**MAIN**'

Viene eseguito paragrafo '**INSERIMENTO-DATI**'.

Viene eseguito il paragrafo '**LAVORO**' ripetutamente, terminando il ciclo quando la variabile '**EOJ**' contiene il valore uno.

Il programma si arresta perché incontra l'istruzione '**STOP RUN**'.

Paragrafo '**LAVORO**'

Il programma esegue il paragrafo '**RICERCA**'.

A questo punto la variabile '**POSIZIONE**' contiene la posizione della lettera contenuta nella variabile '**LETTERA**' e viene visualizzata.

Il programma riceve dall'esterno l'assenso o il dissenso riguardo alla continuazione dell'esecuzione; se l'intenzione è di proseguire, viene eseguito il paragrafo '**INSERIMENTO-DATI**', in caso contrario, viene messo il valore uno nella variabile '**EOJ**'.

Paragrafo '**INSERIMENTO-DATI**'

Il programma riceve dall'esterno una stringa da inserire nella variabile '**RECORD-ELEMENTI**' e la lettera da ricercare nella stringa.

Paragrafo '**RICERCA**'

Viene eseguito un paragrafo che non esegue alcunché (l'istruzione '**EXIT**') scandendo l'indice '**POSIZIONE**' a partire da uno, con passo unitario, terminando quando si supera la dimensione della stringa oppure quando il contenuto di '**ELEMENTO(POSIZIONE)**' coincide con il valore di '**LETTERA**', ovvero quando la posizione della lettera nella stringa è stata trovata.

In pratica, il paragrafo '**EXIT-PARAGRAPH**' è una scusa per utilizzare la scansione dell'istruzione '**PERFORM VARYING**'.

Paragrafo '**EXIT-PARAGRAPH**'

Il paragrafo non fa alcunché.

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/ELM1000.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   ELM1000.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 1985-02-15.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 DATA DIVISION.
000900*
001000 WORKING-STORAGE SECTION.
001100 01  RECORD-ELEMENTI.
001200     02  ELEMENTO  PIC X   OCCURS 60 TIMES.
001300 01  POSIZIONE     PIC 99.
001400 01  EOJ           PIC 9   VALUE ZERO.
001500 01  RISPOSTA      PIC XX.
001600 01  LETTERA       PIC X.
001700*
001800 PROCEDURE DIVISION.
001900*------------------------ LIVELLO 0 -----------------
002000 MAIN.
002100     PERFORM INSERIMENTO-DATI.
002200     PERFORM LAVORO UNTIL EOJ = 1.
002300*
002400     STOP RUN.
002500*------------------------ LIVELLO 1 -----------------
002600 LAVORO.
002700     PERFORM RICERCA.
002800     DISPLAY "LA LETTERA ", LETTERA,
002900             " E' NELLA POSIZIONE ", POSIZIONE.
003000*
003100     DISPLAY "VUOI CONTINUARE? SI O NO".
003200     ACCEPT RISPOSTA.
003300*
```

```
003400     IF RISPOSTA = "SI"
003500       THEN
003600           PERFORM INSERIMENTO-DATI;
003700       ELSE
003800           MOVE 1 TO EOJ.
003900*------------------------ LIVELLO 2 -----------------
004000 INSERIMENTO-DATI.
004100     DISPLAY "INSERISCI LA FRASE".
004200     ACCEPT RECORD-ELEMENTI.
004300*
004400     DISPLAY "INSERISCI LA LETTERA DA TROVARE".
004500     ACCEPT LETTERA.
004600*-------------------------------------------------
004700 RICERCA.
004800     PERFORM EXIT-PARAGRAPH
004900             VARYING POSIZIONE FROM 1 BY 1
005000             UNTIL POSIZIONE > 60
005100             OR    ELEMENTO(POSIZIONE) = LETTERA.
005200*------------------------ LIVELLO 3 -----------------
005300 EXIT-PARAGRAPH.
005400     EXIT.
005500*
```

73.2.11 ELM1100: ricerca sequenziale all'interno di un vettore

«

Variabili

'**RECORD-ELEMENTI**' è una variabile usata per accogliere una stringa;

'**ELEMENTO**' è un array che scompone '**RECORD-ELEMENTI**' in caratteri singoli;

‘**POSIZIONE**’ è l’indice usato per scandire gli elementi della stringa;

‘**EOJ**’ quando assume il valore 1 il programma si arresta;

‘**RISPOSTA**’ è la variabile che riceve la risposta, un ‘**SI**’ o un ‘**NO**’, per la continuazione o meno con un altro calcolo;

‘**LETTERA**’ è la variabile che contiene la lettera da cercare nella stringa.

Descrizione

Il programma riceve dall’esterno il contenuto di una stringa e di una lettera che dovrebbe essere contenuta nella stringa stessa; successivamente il programma scandisce la stringa come vettore di caratteri e individua la prima posizione in cui appare la lettera cercata.

Rispetto a ‘**ELM1000**’ si ottiene un avvertimento quando si indica una lettera che non è contenuta nella frase.

Paragrafo ‘**MAIN**’

Viene eseguito paragrafo ‘**INSERIMENTO-DATI**’.

Viene eseguito il paragrafo ‘**LAVORO**’ ripetutamente, terminando il ciclo quando la variabile ‘**EOJ**’ contiene il valore uno.

Il programma si arresta perché incontra l’istruzione ‘**STOP RUN**’.

Paragrafo ‘**LAVORO**’

Il programma esegue il paragrafo ‘**RICERCA**’.

A questo punto la variabile ‘**POSIZIONE**’ contiene la posizione della lettera contenuta nella variabile ‘**LETTERA**’: se il valore della posizione supera la dimensione massima dell’array, si ottiene

un avvertimento dell'impossibilità di trovare la corrispondenza, altrimenti viene visualizzata la posizione trovata.

Il programma riceve dall'esterno l'assenso o il dissenso riguardo alla continuazione dell'esecuzione; se l'intenzione è di proseguire, viene eseguito il paragrafo '**INSERIMENTO-DATI**', in caso contrario, viene messo il valore uno nella variabile '**EOJ**'.

Paragrafo '**INSERIMENTO-DATI**'

Il programma riceve dall'esterno una stringa da inserire nella variabile '**RECORD-ELEMENTI**' e la lettera da ricercare nella stringa.

Paragrafo '**RICERCA**'

Viene eseguito un paragrafo che non esegue alcunché (l'istruzione '**EXIT**') scandendo l'indice '**POSIZIONE**' a partire da uno, con passo unitario, terminando quando si supera la dimensione della stringa oppure quando il contenuto di '**ELEMENTO(POSIZIONE)**' coincide con il valore di '**LETTERA**', ovvero quando la posizione della lettera nella stringa è stata trovata.

In pratica, il paragrafo '**EXIT-PARAGRAPH**' è una scusa per utilizzare la scansione dell'istruzione '**PERFORM VARYING**'.

Paragrafo '**EXIT-PARAGRAPH**'

Il paragrafo non fa alcunché.

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/ELM1100.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   ELM1100.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 1985-02-15.
```

```
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 DATA DIVISION.
000900*
001000 WORKING-STORAGE SECTION.
001100 01  RECORD-ELEMENTI.
001200     02  ELEMENTO  PIC X   OCCURS 60 TIMES.
001300 01  POSIZIONE     PIC 99.
001400 01  EOJ           PIC 9   VALUE ZERO.
001500 01  RISPOSTA      PIC XX.
001600 01  LETTERA       PIC X.
001700*
001800 PROCEDURE DIVISION.
001900*------------------------ LIVELLO 0 -----------------
002000 MAIN.
002100     PERFORM INSERIMENTO-DATI.
002200     PERFORM LAVORO UNTIL EOJ = 1.
002300*
002400     STOP RUN.
002500*------------------------ LIVELLO 1 -----------------
002600 LAVORO.
002700     PERFORM RICERCA.
002800*
002900     IF POSIZIONE < 61
003000       THEN
003100           DISPLAY "LA LETTERA ", LETTERA,
003200                   " E' NELLA POSIZIONE ", POSIZIONE;
003300       ELSE
003400           DISPLAY "LA LETTERA ", LETTERA,
003500                   " NON E' CONTENUTA NELLA FRASE".
003600*
003700     DISPLAY "VUOI CONTINUARE? SI O NO".
003800     ACCEPT RISPOSTA.
```

```
003900*
004000     IF RISPOSTA = "SI"
004100       THEN
004200           PERFORM INSERIMENTO-DATI;
004300       ELSE
004400           MOVE 1 TO EOJ.
004500*------------------------ LIVELLO 2 -----------------
004600 INSERIMENTO-DATI.
004700     DISPLAY "INSERISCI LA FRASE".
004800     ACCEPT RECORD-ELEMENTI.
004900*
005000     DISPLAY "INSERISCI LA LETTERA DA TROVARE".
005100     ACCEPT LETTERA.
005200*-------------------------------------------------
005300 RICERCA.
005400     PERFORM EXIT-PARAGRAPH
005500             VARYING POSIZIONE FROM 1 BY 1
005600             UNTIL POSIZIONE > 60
005700             OR    ELEMENTO(POSIZIONE) = LETTERA.
005800*------------------------ LIVELLO 3 -----------------
005900 EXIT-PARAGRAPH.
006000     EXIT.
006100*
```

73.2.12 ELM1300: creazione di un file sequenziale

«

File

‘**FILE-DA-SCRIVERE**’ rappresenta il file che viene creato dal programma (il nome del file è ‘`output.seq`’). Il file è di tipo sequenziale, dove la riga ha una dimensione fissa; non si prevede l’inserimento di un codice di interruzione di riga alla fine delle righe.

Variabili

‘**RECORD-DA-SCRIVERE**’ è la riga del file da creare;

‘**EOJ**’ quando assume il valore 1 il programma si arresta.

Descrizione

Il programma riceve dall’esterno il contenuto di ogni riga e di volta in volta lo registra nel file. Il programma termina il lavoro quando la stringa inserita contiene solo asterischi (almeno 30, pari alla larghezza massima prevista di ogni riga).

Paragrafo ‘**MAIN**’

Viene aperto in scrittura il file da creare.

Viene eseguito il paragrafo ‘**INSERIMENTO-DATI**’.

Viene eseguito il paragrafo ‘**LAVORO**’ ripetutamente, concludendo il ciclo quando la variabile ‘**EOJ**’ contiene il valore uno.

Viene chiuso il file da creare.

Il programma si arresta perché incontra l’istruzione ‘**STOP RUN**’.

Paragrafo ‘**LAVORO**’

Si controlla se la stringa inserita contiene soltanto asterischi; se è così viene messo il valore uno nella variabile ‘**EOJ**’, altrimenti viene scritta la riga inserita nel file da scrivere e subito dopo viene eseguito nuovamente il paragrafo ‘**INSERIMENTO-DATI**’.

Paragrafo ‘**INSERIMENTO-DATI**’

Il paragrafo riceve dall’esterno il contenuto di una riga da registrare nel file, tenendo conto che vengono prese in considerazione al massimo i primi 30 caratteri, pari alla dimensione della variabile che deve accogliere i dati.

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/ELM1300.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   ELM1300.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 1985-02-20.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 INPUT-OUTPUT SECTION.
000900*
001000 FILE-CONTROL.
001100*
001200     SELECT FILE-DA-SCRIVERE ASSIGN TO "output.seq"
001300            ORGANIZATION IS SEQUENTIAL.
001400*
001500 DATA DIVISION.
001600*
001700 FILE SECTION.
001800*
001900 FD  FILE-DA-SCRIVERE
002000     LABEL RECORD IS STANDARD.
002100*
002200 01  RECORD-DA-SCRIVERE PIC X(30).
002300*
002400 WORKING-STORAGE SECTION.
002500 01  EOJ                PIC 9   VALUE ZERO.
002600*
002700 PROCEDURE DIVISION.
002800*------------------------ LIVELLO 0 -----------------
002900 MAIN.
003000     OPEN OUTPUT FILE-DA-SCRIVERE.
003100     PERFORM INSERIMENTO-DATI.
003200     PERFORM LAVORO UNTIL EOJ = 1.
003300     CLOSE FILE-DA-SCRIVERE.
```

```
003400*
003500     STOP RUN.
003600*------------------------ LIVELLO 1 -----------------
003700 LAVORO.
003800     IF RECORD-DA-SCRIVERE = ALL "*"
003900       THEN
004000           MOVE 1 TO EOJ;
004100       ELSE
004200           WRITE RECORD-DA-SCRIVERE,
004300           PERFORM INSERIMENTO-DATI.
004400*------------------------ LIVELLO 2 -----------------
004500 INSERIMENTO-DATI.
004600     DISPLAY "INSERISCI IL RECORD".
004700     DISPLAY "PER FINIRE INSERISCI TUTTI ASTERISCHI".
004800     ACCEPT RECORD-DA-SCRIVERE.
004900*
```

Per fare in modo che le righe del file siano concluse come avviene di solito nei file di testo, con un codice di interruzione di riga, occorre specificare nell'istruzione '**SELECT**' un accesso di tipo '**LINE SEQUENTIAL**'.

73.2.13 ELM1400: estensione di un file sequenziale

File

'**FILE-DA-SCRIVERE**' rappresenta il file che viene esteso dal programma (il nome del file è '`output.seq`'). Il file è di tipo sequenziale, dove la riga ha una dimensione fissa; non si prevede l'inserimento di un codice di interruzione di riga alla fine delle righe.

Variabili

‘**RECORD-DA-SCRIVERE**’ è la riga del file da creare;

‘**EOJ**’ quando assume il valore 1 il programma si arresta.

Descrizione

Il programma riceve dall’esterno il contenuto di ogni riga e di volta in volta lo registra nel file. Il programma termina il lavoro quando la stringa inserita contiene solo asterischi (almeno 30, pari alla larghezza massima prevista di ogni riga).

Paragrafo ‘**MAIN**’

Viene aperto in scrittura in aggiunta il file da creare.

Viene eseguito il paragrafo ‘**INSERIMENTO-DATI**’.

Viene eseguito il paragrafo ‘**LAVORO**’ ripetutamente, concludendo il ciclo quando la variabile ‘**EOJ**’ contiene il valore uno.

Viene chiuso il file da creare.

Il programma si arresta perché incontra l’istruzione ‘**STOP RUN**’.

Paragrafo ‘**LAVORO**’

Si controlla se la stringa inserita contiene soltanto asterischi; se è così viene messo il valore uno nella variabile ‘**EOJ**’, altrimenti viene scritta la riga inserita nel file da scrivere e subito dopo viene eseguito nuovamente il paragrafo ‘**INSERIMENTO-DATI**’.

Paragrafo ‘**INSERIMENTO-DATI**’

Il paragrafo riceve dall’esterno il contenuto di una riga da registrare nel file, tenendo conto che vengono prese in considerazione al massimo i primi 30 caratteri, pari alla dimensione della variabile che deve accogliere i dati.

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/ELM1400.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   ELM1400.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 1985-02-20.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 INPUT-OUTPUT SECTION.
000900*
001000 FILE-CONTROL.
001100*
001200     SELECT FILE-DA-SCRIVERE ASSIGN TO "output.seq"
001300            ORGANIZATION IS SEQUENTIAL.
001400*
001500 DATA DIVISION.
001600*
001700 FILE SECTION.
001800*
001900 FD  FILE-DA-SCRIVERE
002000     LABEL RECORD IS STANDARD.
002100*
002200 01  RECORD-DA-SCRIVERE PIC X(30).
002300*
002400 WORKING-STORAGE SECTION.
002500 01  EOJ                PIC 9   VALUE ZERO.
002600*
002700 PROCEDURE DIVISION.
002800*------------------------ LIVELLO 0 -----------------
002900 MAIN.
003000     OPEN EXTEND FILE-DA-SCRIVERE.
003100     PERFORM INSERIMENTO-DATI.
003200     PERFORM LAVORO UNTIL EOJ = 1.
003300     CLOSE FILE-DA-SCRIVERE.
```

```
003400*
003500     STOP RUN.
003600*------------------------ LIVELLO 1 -----------------
003700 LAVORO.
003800     IF RECORD-DA-SCRIVERE = ALL "*"
003900       THEN
004000           MOVE 1 TO EOJ;
004100       ELSE
004200           WRITE RECORD-DA-SCRIVERE,
004300           PERFORM INSERIMENTO-DATI.
004400*------------------------ LIVELLO 2 -----------------
004500 INSERIMENTO-DATI.
004600     DISPLAY "INSERISCI LA RIGA".
004700     DISPLAY "PER FINIRE INSERISCI TUTTI ASTERISCHI".
004800     ACCEPT RECORD-DA-SCRIVERE.
004900*
```

Per fare in modo che le righe del file siano concluse come avviene di solito nei file di testo, con un codice di interruzione di riga, occorre specificare nell'istruzione '**SELECT**' un accesso di tipo '**LINE SEQUENTIAL**'.

73.2.14 ELM1500: lettura di un file sequenziale

«

File

'**FILE-DA-LEGGERE**' rappresenta il file che viene letto dal programma (il nome del file è '`input.seq`'). Il file è di tipo sequenziale, dove ogni riga ha una dimensione fissa e non si fa affidamento sulla presenza di un codice di interruzione di riga.

Variabili

'**RECORD-DA-LEGGERE**' è la riga del file da leggere;

‘**EOF**’ quando assume il valore 1 indica che la lettura ha superato la fine del file.

Descrizione

Il programma visualizza il contenuto di un file.

La lettura avviene a blocchi di 30 caratteri, indipendentemente dal fatto che siano presenti dei codici di interruzione di riga. Diversamente, per fare in modo che la lettura sia al massimo di 30 caratteri, ma rispettando anche i codici di interruzione di riga, occorre specificare nell’istruzione ‘**SELECT**’ un accesso di tipo ‘**LINE SEQUENTIAL**’.

Paragrafo ‘**MAIN**’

Viene aperto in lettura il file da leggere.

Viene eseguita la lettura di un primo blocco, pari alla dimensione della variabile ‘**RECORD-DA-LEGGERE**’; se si verifica la condizione ‘**AT END**’, ovvero se il file è vuoto, viene messo il valore uno nella variabile ‘**EOF**’.

Viene eseguito il paragrafo ‘**LETTURA**’, ripetutamente, utilizzando come condizione di arresto il fatto che la variabile ‘**EOF**’ contenga il valore uno.

Viene chiuso il file da leggere.

Il programma si arresta perché incontra l’istruzione ‘**STOP RUN**’.

Paragrafo ‘**LETTURA**’

Viene visualizzata la porzione di file appena letta.

Viene eseguita la lettura del file da leggere; se si verifica la condizione ‘**AT END**’, ovvero se la lettura non ha acquisito alcunché, viene messo il valore uno nella variabile ‘**EOF**’.

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/ELM1500.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   ELM1500.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 1985-02-20.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 INPUT-OUTPUT SECTION.
000900*
001000 FILE-CONTROL.
001100*
001200     SELECT FILE-DA-LEGGERE ASSIGN TO "input.seq"
001300            ORGANIZATION IS SEQUENTIAL.
001400*
001500 DATA DIVISION.
001600*
001700 FILE SECTION.
001800*
001900 FD  FILE-DA-LEGGERE
002000     LABEL RECORD IS STANDARD.
002100*
002200 01  RECORD-DA-LEGGERE PIC X(30).
002300*
002400 WORKING-STORAGE SECTION.
002500 01  EOF               PIC 9   VALUE ZERO.
002600*
002700 PROCEDURE DIVISION.
002800*------------------------ LIVELLO 0 -----------------
002900 MAIN.
003000     OPEN INPUT FILE-DA-LEGGERE.
003100     READ FILE-DA-LEGGERE
003200          AT END
003300               MOVE 1 TO EOF.
```

```
003400     PERFORM LETTURA UNTIL EOF = 1.
003500     CLOSE FILE-DA-LEGGERE.
003600*
003700     STOP RUN.
003800*------------------------ LIVELLO 1 -----------------
003900 LETTURA.
004000     DISPLAY RECORD-DA-LEGGERE.
004100     READ FILE-DA-LEGGERE
004200          AT END
004300               MOVE 1 TO EOF.
004400*
```

Figura 73.23. Foto ricordo della festa conclusiva di un corso sul linguaggio COBOL realizzato con l'elaboratore Burroughs B91, presumibilmente tra il 1982 e il 1983. Nell'immagine, l'ingegnere che ha tenuto il corso compila un diploma preparato per scherzo dagli studenti che lo hanno frequentato.

73.3 Esempi elementari con i file

Qui si raccolgono alcuni esempi elementari di programmi COBOL per l'accesso ai file, risalenti a un lavoro didattico del 1983. Salvo dove indicato in maniera differente, gli esempi mostrati funzionano regolarmente se compilati con OpenCOBOL 0.31.

73.3.1 AGO-83-1: estensione di un file sequenziale

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/AGO-83-1.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   AGO-83-1.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-03-20.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 INPUT-OUTPUT SECTION.
000900*
001000 FILE-CONTROL.
001100*
001200     SELECT FILE-DA-SCRIVERE ASSIGN TO "file.seq"
001300            ORGANIZATION IS SEQUENTIAL.
001400*
001500 DATA DIVISION.
001600*
001700 FILE SECTION.
001800*
001900 FD  FILE-DA-SCRIVERE
002000     LABEL RECORD IS STANDARD.
002100*
002200 01  RECORD-DA-SCRIVERE.
002300     02  CODICE-FILE      PIC 9(10) COMP.
```

```
002400     02  TESTO              PIC X(75).
002500*
002600 WORKING-STORAGE SECTION.
002700*
002800 01  CAMPI-SCALARI.
002900     02  EOJ                PIC 9     COMP VALUE IS 0.
003000*
003100 PROCEDURE DIVISION.
003200*------------------------ LIVELLO 0 ----------------
003300 MAIN.
003400     OPEN EXTEND FILE-DA-SCRIVERE.
003500     PERFORM INSERIMENTO-DATI UNTIL EOJ = 1.
003600     CLOSE FILE-DA-SCRIVERE.
003700     STOP RUN.
003800*------------------------ LIVELLO 1 ----------------
003900 INSERIMENTO-DATI.
004000     DISPLAY "INSERISCI PRIMA IL CODICE NUMERICO, ",
004050             "POI IL TESTO"
004100     ACCEPT CODICE-FILE.
004200     IF CODICE-FILE = 0
004300       THEN
004400           MOVE 1 TO EOJ,
004500       ELSE
004600           ACCEPT TESTO,
004700           WRITE RECORD-DA-SCRIVERE.
004800*
```

73.3.2 AGO-83-2: lettura sequenziale e ricerca di una chiave

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/AGO-83-2.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   AGO-83-2.
```

```
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 1983-08.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 INPUT-OUTPUT SECTION.
000900*
001000 FILE-CONTROL.
001100*
001200     SELECT FILE-DA-LEGGERE ASSIGN TO "file.seq"
001300            ORGANIZATION IS SEQUENTIAL.
001400*
001500 DATA DIVISION.
001600*
001700 FILE SECTION.
001800*
001900 FD  FILE-DA-LEGGERE
002000     LABEL RECORD IS STANDARD.
002100*
002200 01  RECORD-DA-LEGGERE.
002300     02  CODICE-FILE      PIC 9(10) COMP.
002400     02  TESTO            PIC X(75).
002500*
002600 WORKING-STORAGE SECTION.
002700*
002800 01  CAMPI-SCALARI.
002900     02  EOF              PIC 9     COMP VALUE IS 0.
003000     02  EOJ              PIC 9     COMP VALUE IS 0.
003100     02  CODICE-RECORD    PIC 9(10) COMP VALUE IS 0.
003200*
003300 PROCEDURE DIVISION.
003400*------------------------ LIVELLO 0 -----------------
003500 MAIN.
003600     OPEN INPUT FILE-DA-LEGGERE.
```

```
003700     READ FILE-DA-LEGGERE
003800          AT END MOVE 1 TO EOF.
003900     PERFORM DOMANDA UNTIL EOF = 1 OR EOJ = 1.
004000     CLOSE FILE-DA-LEGGERE.
004100     STOP RUN.
004200*------------------------ LIVELLO 1 -----------------
004300 DOMANDA.
004400     DISPLAY "INSERISCI IL CODICE DEL RECORD, ",
004450             "DI 10 CIFRE"
004500     ACCEPT CODICE-RECORD.
004600     IF CODICE-RECORD = 0
004700       THEN
004800           MOVE 1 TO EOJ.
004900     PERFORM RICERCA UNTIL EOF = 1 OR EOJ = 1.
005000     CLOSE FILE-DA-LEGGERE.
005100     MOVE ZERO TO EOF.
005200     OPEN INPUT FILE-DA-LEGGERE.
005300     READ FILE-DA-LEGGERE
005400          AT END MOVE 1 TO EOF.
005500*------------------------ LIVELLO 2 -----------------
005600 RICERCA.
005700     IF CODICE-FILE = CODICE-RECORD
005800       THEN
005900           DISPLAY CODICE-FILE, " ", TESTO.
006000     READ FILE-DA-LEGGERE
006100          AT END MOVE 1 TO EOF.
006200*
```

73.3.3 AGO-83-3: estensione di un file relativo

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/AGO-83-3.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   AGO-83-3.
```

```
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-03-20.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 INPUT-OUTPUT SECTION.
000900*
001000 FILE-CONTROL.
001100*
001200     SELECT FILE-DA-SCRIVERE ASSIGN TO "file.rel"
001300                             ORGANIZATION IS RELATIVE
001400                             ACCESS MODE IS SEQUENTIAL.
001500*
001600 DATA DIVISION.
001700*
001800 FILE SECTION.
001900*
002000 FD  FILE-DA-SCRIVERE
002100     LABEL RECORD IS STANDARD.
002200*
002300 01  RECORD-DA-SCRIVERE.
002400     02  TESTO            PIC X(80).
002500*
002600 WORKING-STORAGE SECTION.
002700*
002800 01  CAMPI-SCALARI.
002900     02  EOJ              PIC 9     COMP VALUE IS 0.
003000*
003100 PROCEDURE DIVISION.
003200*------------------------ LIVELLO 0 -----------------
003300 MAIN.
003400     OPEN EXTEND FILE-DA-SCRIVERE.
003500     PERFORM INSERIMENTO-DATI UNTIL EOJ = 1.
003600     CLOSE FILE-DA-SCRIVERE.
```

```
003700     STOP RUN.
003800*------------------------ LIVELLO 1 ----------------
003900 INSERIMENTO-DATI.
004000     DISPLAY "INSERISCI IL TESTO DEL RECORD"
004100     ACCEPT TESTO.
004200     IF TESTO = SPACES
004300       THEN
004400           MOVE 1 TO EOJ,
004500       ELSE
004600           WRITE RECORD-DA-SCRIVERE.
004700*
```

73.3.4 AGO-83-4: lettura di un file relativo ad accesso diretto

«

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/AGO-83-4.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   AGO-83-4.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 1983-08.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 INPUT-OUTPUT SECTION.
000900*
001000 FILE-CONTROL.
001100*
001200     SELECT FILE-DA-LEGGERE ASSIGN TO "file.rel"
001300                            ORGANIZATION IS RELATIVE
001400                            ACCESS MODE IS RANDOM
001500                            RELATIVE KEY IS N-RECORD.
001600*
001700 DATA DIVISION.
```

```
001800*
001900 FILE SECTION.
002000*
002100 FD  FILE-DA-LEGGERE
002200     LABEL RECORD IS STANDARD.
002300*
002400 01  RECORD-DA-LEGGERE.
002500     02  TESTO            PIC X(80).
002600*
002700 WORKING-STORAGE SECTION.
002800*
002900 01  CAMPI-SCALARI.
003000     02  INVALID-KEY      PIC 9     COMP VALUE IS 0.
003100     02  EOJ              PIC 9     COMP VALUE IS 0.
003200     02  N-RECORD         PIC 9(10) COMP VALUE IS 0.
003300*
003400 PROCEDURE DIVISION.
003500*------------------------ LIVELLO 0 -----------------
003600 MAIN.
003700     OPEN INPUT FILE-DA-LEGGERE.
003800     PERFORM ELABORA UNTIL EOJ = 1.
003900     CLOSE FILE-DA-LEGGERE.
004000     STOP RUN.
004100*------------------------ LIVELLO 1 -----------------
004200 ELABORA.
004300     DISPLAY "INSERISCI IL NUMERO DEL RECORD"
004400     ACCEPT N-RECORD.
004500     IF N-RECORD = 0
004600       THEN
004700           MOVE 1 TO EOJ;
004800       ELSE
004900           PERFORM LEGGI,
005000           IF INVALID-KEY = 1
005100             THEN
```

```
005200                 DISPLAY "INVALID KEY";
005300             ELSE
005400                 PERFORM VISUALIZZA.
005500*------------------------ LIVELLO 2 -----------------
005600 VISUALIZZA.
005700     DISPLAY N-RECORD, " ", TESTO.
005800*--------------------------------------------------
005900 LEGGI.
006000     MOVE ZERO TO INVALID-KEY.
006100     READ FILE-DA-LEGGERE
006200          INVALID KEY
006300                     MOVE 1 TO INVALID-KEY.
006400*
```

73.3.5 AGO-83-5: creazione di un file a indice

«

Questo esempio funziona con il compilatore TinyCOBOL 0.61. In questo caso, vengono creati due file: ‘`file.ind`’ e ‘`file.ind1`’, che insieme costituiscono lo stesso file logico.

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/AGO-83-5.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   AGO-83-5.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-03-20.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 INPUT-OUTPUT SECTION.
000900*
001000 FILE-CONTROL.
001100*
001200     SELECT FILE-DA-SCRIVERE
```

```
001250            ASSIGN TO "file.ind"
001300            ORGANIZATION IS INDEXED
001400            ACCESS MODE IS SEQUENTIAL
001500            RECORD KEY IS CHIAVE
001600            ALTERNATE RECORD KEY IS CHIAVE2
001700                                 WITH DUPLICATES.
001800*
001900 DATA DIVISION.
002000*
002100 FILE SECTION.
002200*
002300 FD  FILE-DA-SCRIVERE
002400     LABEL RECORD IS STANDARD.
002500*
002600 01  RECORD-DA-SCRIVERE.
002700     02  CHIAVE           PIC X(5).
002800     02  CHIAVE2          PIC X(5).
002900     02  TESTO            PIC X(70).
003000*
003100 WORKING-STORAGE SECTION.
003200*
003300 01  CAMPI-SCALARI.
003400     02  EOJ              PIC 9     COMP VALUE IS 0.
003500*
003600 PROCEDURE DIVISION.
003700*------------------------ LIVELLO 0 -----------------
003800 MAIN.
003900     OPEN OUTPUT FILE-DA-SCRIVERE.
004000     PERFORM INSERIMENTO-DATI UNTIL EOJ = 1.
004100     CLOSE FILE-DA-SCRIVERE.
004200     STOP RUN.
004300*------------------------ LIVELLO 1 -----------------
004400 INSERIMENTO-DATI.
004500     DISPLAY "INSERISCI IL RECORD: ",
```

```
004550              "I PRIMI CINQUE CARATTERI ",
004600              "COSTITUISCONO LA CHIAVE PRIMARIA ",
004700              "CHE DEVE ESSERE UNICA"
004800     ACCEPT RECORD-DA-SCRIVERE.
004900     IF RECORD-DA-SCRIVERE = SPACES
005000       THEN
005100           MOVE 1 TO EOJ,
005200       ELSE
005300           WRITE RECORD-DA-SCRIVERE
005400                 INVALID KEY
005500                                DISPLAY "LA CHIAVE ",
005550                                        CHIAVE,
005600                                        " E' DOPPIA,",
005650                                        " OPPURE ",
005700                                        "NON E' VALIDA".
005800*
```

73.3.6 AGO-83-6: lettura di un file a indice ad accesso diretto

«

Questo esempio funziona con il compilatore TinyCOBOL 0.61 e utilizza il file creato con l'esempio precedente.

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/AGO-83-6.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   AGO-83-6.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 1983-08.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 INPUT-OUTPUT SECTION.
000900*
```

```
001000 FILE-CONTROL.
001100*
001200     SELECT FILE-DA-LEGGERE
001250            ASSIGN TO "file.ind"
001300            ORGANIZATION IS INDEXED
001400            ACCESS MODE IS RANDOM
001500            RECORD KEY IS CHIAVE
001600            ALTERNATE RECORD KEY IS CHIAVE2
001700                      WITH DUPLICATES.
001800*
001900 DATA DIVISION.
002000*
002100 FILE SECTION.
002200*
002300 FD  FILE-DA-LEGGERE
002400     LABEL RECORD IS STANDARD.
002500*
002600 01  RECORD-DA-LEGGERE.
002700     02  CHIAVE           PIC X(5).
002800     02  CHIAVE2          PIC X(5).
002900     02  TESTO            PIC X(70).
003000*
003100 WORKING-STORAGE SECTION.
003200*
003300 01  CAMPI-SCALARI.
003400     02  EOJ              PIC 9     COMP VALUE IS 0.
003500     02  INV-KEY          PIC 9     COMP VALUE IS 0.
003600*
003700 PROCEDURE DIVISION.
003800*------------------------ LIVELLO 0 -----------------
003900 MAIN.
004000     OPEN INPUT FILE-DA-LEGGERE.
004100     PERFORM ELABORAZIONE UNTIL EOJ = 1.
004200     CLOSE FILE-DA-LEGGERE.
```

```
004300     STOP RUN.
004400*------------------------ LIVELLO 1 ----------------
004500 ELABORAZIONE.
004600     DISPLAY "INSERISCI LA CHIAVE PRIMARIA".
004700     ACCEPT CHIAVE.
004800     IF CHIAVE = SPACES
004900       THEN
005000           MOVE 1 TO EOJ,
005100       ELSE
005200           PERFORM LEGGI,
005300           IF INV-KEY = 1
005400             THEN
005500                 DISPLAY "INVALID KEY: ", CHIAVE,
005600             ELSE
005700                 DISPLAY CHIAVE, " ", CHIAVE2, " ",
005750                         TESTO.
005800*------------------------ LIVELLO 2 ----------------
005900 LEGGI.
006000     MOVE 0 TO INV-KEY.
006100     READ FILE-DA-LEGGERE
006200          INVALID KEY
006300                      MOVE 1 TO INV-KEY.
006400*
```

73.3.7 AGO-83-8: lettura di un file a indice ad accesso dinamico

«

Questo esempio funziona parzialmente con il compilatore TinyCOBOL 0.61 e utilizza il file già predisposto per quello precedente. Si osservi che si fa riferimento alla chiave secondaria del file, in modo da poter contare sulla presenza di chiavi doppie.

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/AGO-83-8.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   AGO-83-8.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 1983-08.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 INPUT-OUTPUT SECTION.
000900*
001000 FILE-CONTROL.
001100*
001200     SELECT FILE-DA-LEGGERE ASSIGN TO "file.ind"
001300                            ORGANIZATION IS INDEXED
001400                            ACCESS MODE IS DYNAMIC
001500                            RECORD KEY IS CHIAVE2.
001600*
001700 DATA DIVISION.
001800*
001900 FILE SECTION.
002000*
002100 FD  FILE-DA-LEGGERE
002200     LABEL RECORD IS STANDARD.
002300*
002400 01  RECORD-DA-LEGGERE.
002500     02  CHIAVE           PIC X(5).
002600     02  CHIAVE2          PIC X(5).
002700     02  TESTO            PIC X(70).
002800*
002900 WORKING-STORAGE SECTION.
003000*
003100 01  CAMPI-SCALARI.
003200     02  EOJ              PIC 9     COMP VALUE IS 0.
003300     02  EOF              PIC 9     COMP VALUE IS 0.
003400     02  INV-KEY          PIC 9     COMP VALUE IS 0.
```

```
003500     02  END-KEY          PIC 9     COMP VALUE IS 0.
003600     02  CHIAVE-W         PIC X(5).
003700*
003800 PROCEDURE DIVISION.
003900*------------------------ LIVELLO 0 ----------------
004000 MAIN.
004100     OPEN INPUT FILE-DA-LEGGERE.
004200     PERFORM ELABORAZIONE UNTIL EOJ = 1.
004300     CLOSE FILE-DA-LEGGERE.
004400     STOP RUN.
004500*------------------------ LIVELLO 1 ----------------
004600 ELABORAZIONE.
004700     DISPLAY "INSERISCI LA CHIAVE SECONDARIA".
004800     ACCEPT CHIAVE2.
004900     IF CHIAVE2 = SPACES
005000       THEN
005100           MOVE 1 TO EOJ,
005200       ELSE
005300           MOVE CHIAVE2 TO CHIAVE-W,
005400           PERFORM LEGGI,
005500           IF INV-KEY = 1
005600             THEN
005700                 DISPLAY "INVALID KEY: ", CHIAVE2,
005800             ELSE
005900                 PERFORM MOSTRA-LEGGI-NEXT
006000                         UNTIL END-KEY = 1
006100                            OR EOF     = 1.
006200*------------------------ LIVELLO 2 ----------------
006300 LEGGI.
006400     MOVE ZERO TO END-KEY.
006500     MOVE ZERO TO EOF.
006600     MOVE ZERO TO INV-KEY.
006700     READ FILE-DA-LEGGERE
006800          INVALID KEY MOVE 1 TO INV-KEY.
```

```
006900*-----------------------------------------------------
007000 MOSTRA-LEGGI-NEXT.
007100     DISPLAY CHIAVE, " ", CHIAVE2, " ", TESTO.
007200     READ FILE-DA-LEGGERE NEXT RECORD
007300          AT END MOVE 1 TO EOF.
007400     IF NOT CHIAVE-W = CHIAVE2
007500       THEN
007600           MOVE 1 TO END-KEY.
007700*
```

73.3.8 AGO-83-10: lettura di un file a indice ad accesso dinamico

Questo esempio funziona con il compilatore TinyCOBOL 0.61 e utilizza il file già predisposto per quello precedente. In questo caso si ritorna a utilizzare la chiave primaria.

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/AGO-83-10.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   AGO-83-10.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 1983-08.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 INPUT-OUTPUT SECTION.
000900*
001000 FILE-CONTROL.
001100*
001200     SELECT FILE-DA-LEGGERE ASSIGN TO "file.ind"
001300                            ORGANIZATION IS INDEXED
001400                            ACCESS MODE IS DYNAMIC
001500                            RECORD KEY IS CHIAVE.
```

```
001600*
001700 DATA DIVISION.
001800*
001900 FILE SECTION.
002000*
002100 FD  FILE-DA-LEGGERE
002200     LABEL RECORD IS STANDARD.
002300*
002400 01  RECORD-DA-LEGGERE.
002500     02  CHIAVE           PIC X(5).
002600     02  CHIAVE2          PIC X(5).
002700     02  TESTO            PIC X(70).
002800*
002900 WORKING-STORAGE SECTION.
003000*
003100 01  CAMPI-SCALARI.
003200     02  EOJ              PIC 9     COMP VALUE IS 0.
003300     02  EOF              PIC 9     COMP VALUE IS 0.
003400     02  INV-KEY          PIC 9     COMP VALUE IS 0.
003500     02  END-KEY          PIC 9     COMP VALUE IS 0.
003600     02  CHIAVE-INIZIALE  PIC X(5).
003700     02  CHIAVE-FINALE    PIC X(5).
003800     02  CHIAVE-SCAMBIO   PIC X(5).
003900*
004000 PROCEDURE DIVISION.
004100*------------------------ LIVELLO 0 -----------------
004200 MAIN.
004300     OPEN INPUT FILE-DA-LEGGERE.
004400     PERFORM ELABORAZIONE UNTIL EOJ = 1.
004500     CLOSE FILE-DA-LEGGERE.
004600     STOP RUN.
004700*------------------------ LIVELLO 1 -----------------
004800 ELABORAZIONE.
004900     DISPLAY "INSERISCI LA CHIAVE PRIMARIA ",
```

```
005000              "INIZIALE, POI QUELLA FINALE".
005100     ACCEPT CHIAVE-INIZIALE.
005200     ACCEPT CHIAVE-FINALE.
005300     IF CHIAVE-INIZIALE > CHIAVE-FINALE
005400       THEN
005500           MOVE CHIAVE-INIZIALE TO CHIAVE-SCAMBIO,
005600           MOVE CHIAVE-FINALE   TO CHIAVE-INIZIALE,
005700           MOVE CHIAVE-SCAMBIO  TO CHIAVE-FINALE.
005800     IF CHIAVE-INIZIALE = SPACES
005900       THEN
006000           MOVE 1 TO EOJ,
006100       ELSE
006200           MOVE CHIAVE-INIZIALE TO CHIAVE,
006300           PERFORM LEGGI,
006400           IF INV-KEY = 1
006500             THEN
006600                 DISPLAY "INVALID KEY: ", CHIAVE,
006700             ELSE
006800                 PERFORM MOSTRA-LEGGI-NEXT
006900                         UNTIL END-KEY = 1
007000                            OR EOF     = 1.
007100*------------------------ LIVELLO 2 -----------------
007200 LEGGI.
007300     MOVE ZERO TO END-KEY.
007400     MOVE ZERO TO EOF.
007500     MOVE ZERO TO INV-KEY.
007600     READ FILE-DA-LEGGERE
007700          INVALID KEY MOVE 1 TO INV-KEY.
007800*-----------------------------------------------------
007900 MOSTRA-LEGGI-NEXT.
008000     DISPLAY CHIAVE, " ", CHIAVE2, " ", TESTO.
008100     READ FILE-DA-LEGGERE NEXT RECORD
008200          AT END MOVE 1 TO EOF.
008300     IF CHIAVE > CHIAVE-FINALE
```

```
008400       THEN
008500           MOVE 1 TO END-KEY.
008600*
```

73.3.9 AGO-83-12: lettura di un file a indice ad accesso dinamico

«

Questo esempio funziona con il compilatore TinyCOBOL 0.61 e utilizza il file già predisposto per quello precedente. In questo caso si utilizza l'istruzione '**START**' per il posizionamento iniziale.

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/AGO-83-12.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   AGO-83-12.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 1983-08.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 INPUT-OUTPUT SECTION.
000900*
001000 FILE-CONTROL.
001100*
001200     SELECT FILE-DA-LEGGERE ASSIGN TO "file.ind"
001300                            ORGANIZATION IS INDEXED
001400                            ACCESS MODE IS DYNAMIC
001500                            RECORD KEY IS CHIAVE.
001600*
001700 DATA DIVISION.
001800*
001900 FILE SECTION.
002000*
002100 FD  FILE-DA-LEGGERE
```

```
002200     LABEL RECORD IS STANDARD.
002300*
002400 01  RECORD-DA-LEGGERE.
002500     02  CHIAVE           PIC X(5).
002600     02  CHIAVE2          PIC X(5).
002700     02  TESTO            PIC X(70).
002800*
002900 WORKING-STORAGE SECTION.
003000*
003100 01  CAMPI-SCALARI.
003200     02  EOJ              PIC 9     COMP VALUE IS 0.
003300     02  EOF              PIC 9     COMP VALUE IS 0.
003400     02  INV-KEY          PIC 9     COMP VALUE IS 0.
003500     02  END-KEY          PIC 9     COMP VALUE IS 0.
003600     02  CHIAVE-INIZIALE  PIC X(5).
003700     02  CHIAVE-FINALE    PIC X(5).
003800     02  CHIAVE-SCAMBIO   PIC X(5).
003900*
004000 PROCEDURE DIVISION.
004100*------------------------ LIVELLO 0 -----------------
004200 MAIN.
004300     OPEN INPUT FILE-DA-LEGGERE.
004400     PERFORM ELABORAZIONE UNTIL EOJ = 1.
004500     CLOSE FILE-DA-LEGGERE.
004600     STOP RUN.
004700*------------------------ LIVELLO 1 -----------------
004800 ELABORAZIONE.
004900     DISPLAY "INSERISCI LA CHIAVE PRIMARIA ",
005000             "INIZIALE, POI QUELLA FINALE".
005100     ACCEPT CHIAVE-INIZIALE.
005200     ACCEPT CHIAVE-FINALE.
005300     IF CHIAVE-INIZIALE > CHIAVE-FINALE
005400       THEN
005500           MOVE CHIAVE-INIZIALE TO CHIAVE-SCAMBIO,
```

```
005600           MOVE CHIAVE-FINALE   TO CHIAVE-INIZIALE,
005700           MOVE CHIAVE-SCAMBIO  TO CHIAVE-FINALE.
005800     IF CHIAVE-INIZIALE = SPACES
005900       THEN
006000           MOVE 1 TO EOJ,
006100       ELSE
006200           MOVE CHIAVE-INIZIALE TO CHIAVE,
006300           PERFORM START-LEGGI,
006400           IF INV-KEY = 1
006500             THEN
006600                 DISPLAY "INVALID KEY: ", CHIAVE,
006700             ELSE
006800                 PERFORM MOSTRA-LEGGI-NEXT
006900                         UNTIL END-KEY = 1
007000                            OR EOF     = 1.
007100*------------------------ LIVELLO 2 ----------------
007200 START-LEGGI.
007300     MOVE ZERO TO END-KEY.
007400     MOVE ZERO TO EOF.
007500     MOVE ZERO TO INV-KEY.
007600     START FILE-DA-LEGGERE KEY IS NOT < CHIAVE
007700          INVALID KEY MOVE 1 TO INV-KEY.
007800     IF NOT INV-KEY = 1
007900       THEN
008000           PERFORM LEGGI.
008100*-------------------------------------------------
008200 MOSTRA-LEGGI-NEXT.
008300     DISPLAY CHIAVE, " ", CHIAVE2, " ", TESTO.
008400     PERFORM LEGGI.
008500*------------------------ LIVELLO 3 ----------------
008600 LEGGI.
008700     READ FILE-DA-LEGGERE NEXT RECORD
008800          AT END MOVE 1 TO EOF.
008900     IF CHIAVE > CHIAVE-FINALE
```

```
009000       THEN
009100           MOVE 1 TO END-KEY.
009200*
```

73.3.10 AGO-83-13: creazione di un file sequenziale con dati da rielaborare

Questo esempio serve a creare un file sequenziale, contenente dei calcoli da eseguire, successivamente, con un altro programma.

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/AGO-83-13.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   AGO-83-13.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-03-22.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 INPUT-OUTPUT SECTION.
000900*
001000 FILE-CONTROL.
001100*
001200     SELECT FILE-DA-SCRIVERE ASSIGN TO "calc.seq"
001300            ORGANIZATION IS SEQUENTIAL.
001400*
001500 DATA DIVISION.
001600*
001700 FILE SECTION.
001800*
001900 FD  FILE-DA-SCRIVERE
002000     LABEL RECORD IS STANDARD.
002100*
002200 01  RECORD-DA-SCRIVERE.
```

```
002300     02  NUMERO-1          PIC 9(15).
002400     02  TIPO-CALCOLO      PIC X.
002500     02  NUMERO-2          PIC 9(15).
002600     02  FILLER            PIC X.
002700     02  RISULTATO         PIC 9(15).
002800     02  FILLER            PIC X.
002900     02  RESTO             PIC 9(15).
003000     02  NOTE              PIC X(18).
003100*
003200 WORKING-STORAGE SECTION.
003300*
003400 01  CAMPI-SCALARI.
003500     02  EOJ               PIC 9     COMP VALUE IS 0.
003600*
003700 PROCEDURE DIVISION.
003800*------------------------ LIVELLO 0 -----------------
003900 MAIN.
004000     OPEN EXTEND FILE-DA-SCRIVERE.
004100     PERFORM INSERIMENTO-DATI UNTIL EOJ = 1.
004200     CLOSE FILE-DA-SCRIVERE.
004300     STOP RUN.
004400*------------------------ LIVELLO 1 -----------------
004500 INSERIMENTO-DATI.
004600     DISPLAY "INSERISCI, IN SEQUENZA, ",
004650             "IL PRIMO NUMERO, ",
004700             "IL SIMBOLO DELL'OPERAZIONE, ",
004750             "IL SECONDO NUMERO".
004800     ACCEPT NUMERO-1.
004900     ACCEPT TIPO-CALCOLO.
005000     ACCEPT NUMERO-2.
005100     IF NUMERO-1 = 0 AND NUMERO-2 = 0
005150        AND TIPO-CALCOLO = SPACE
005200       THEN
005300           MOVE 1 TO EOJ,
```

```
005400       ELSE
005500           WRITE RECORD-DA-SCRIVERE.
005600*
```

73.3.11 AGO-83-14: lettura e riscrittura di un file sequenziale

Questo esempio legge e riscrive il file generato con l'esempio precedente, eseguendo i calcoli previsti e mostrando anche il risultato a video.

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/AGO-83-14.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   AGO-83-14.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 1983-08.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 INPUT-OUTPUT SECTION.
000900*
001000 FILE-CONTROL.
001100*
001200     SELECT FILE-DA-ELABORARE ASSIGN TO "calc.seq"
001300            ORGANIZATION IS SEQUENTIAL.
001400*
001500 DATA DIVISION.
001600*
001700 FILE SECTION.
001800*
001900 FD  FILE-DA-ELABORARE
002000     LABEL RECORD IS STANDARD.
002100*
002200 01  RECORD-DA-ELABORARE.
```

```
002300     02  NUMERO-1          PIC 9(15).
002400     02  TIPO-CALCOLO      PIC X.
002500     02  NUMERO-2          PIC 9(15).
002600     02  UGUALE            PIC X.
002700     02  RISULTATO         PIC 9(15).
002800     02  SEPARAZIONE       PIC X.
002900     02  RESTO             PIC 9(15).
003000     02  NOTE              PIC X(18).
003100*
003200 WORKING-STORAGE SECTION.
003300*
003400 01  CAMPI-SCALARI.
003500     02  EOF               PIC 9     COMP VALUE IS 0.
003600     02  EOJ               PIC 9     COMP VALUE IS 0.
003700*
003800 PROCEDURE DIVISION.
003900*------------------------ LIVELLO 0 -----------------
004000 MAIN.
004100     OPEN I-O FILE-DA-ELABORARE.
004200     READ FILE-DA-ELABORARE
004300          AT END MOVE 1 TO EOF.
004400     PERFORM ELABORAZIONE UNTIL EOF = 1.
004500     CLOSE FILE-DA-ELABORARE.
004600     STOP RUN.
004700*------------------------ LIVELLO 1 -----------------
004800 ELABORAZIONE.
004900     MOVE SPACES TO NOTE.
005000     MOVE ZERO   TO RESTO.
005100     IF      TIPO-CALCOLO = "+"
005200     THEN
005300         COMPUTE RISULTATO = NUMERO-1 + NUMERO-2;
005400     ELSE IF TIPO-CALCOLO = "-"
005500     THEN
005600         COMPUTE RISULTATO = NUMERO-1 - NUMERO-2;
```

```
005700      ELSE IF TIPO-CALCOLO = "*"
005800      THEN
005900          COMPUTE RISULTATO = NUMERO-1 * NUMERO-2;
006000      ELSE IF TIPO-CALCOLO = "/"
006100      THEN
006200          DIVIDE NUMERO-1 BY NUMERO-2 GIVING RISULTATO,
006300                 REMAINDER RESTO;
006400      ELSE
006500          MOVE ZERO TO RISULTATO,
006600          MOVE "CALCOLO ERRATO" TO NOTE.
006700
006800      MOVE "="    TO UGUALE.
006900      MOVE SPACE TO SEPARAZIONE.
007000      DISPLAY RECORD-DA-ELABORARE.
007100      REWRITE RECORD-DA-ELABORARE.
007200      READ FILE-DA-ELABORARE
007300           AT END MOVE 1 TO EOF.
007400*
```

73.3.12 AGO-83-15: estensione di un file sequenziale contenente aggiornamenti successivi

Questo esempio estende un file sequenziale con delle informazioni, che possono essere aggiornate in momenti successivi. I record si considerano contenere la stessa informazione, aggiornata, quando hanno la stessa chiave.

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/AGO-83-15.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.    AGO-83-15.
000300 AUTHOR.        DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-03-22.
```

```
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 INPUT-OUTPUT SECTION.
000900*
001000 FILE-CONTROL.
001100*
001200     SELECT FILE-AGGIORNAMENTI ASSIGN TO "agg.seq"
001300            ORGANIZATION IS SEQUENTIAL.
001400*
001500 DATA DIVISION.
001600*
001700 FILE SECTION.
001800*
001900 FD  FILE-AGGIORNAMENTI
002000     LABEL RECORD IS STANDARD.
002100*
002200 01  RECORD-AGGIORNAMENTI.
002300     02  CHIAVE           PIC X(5).
002400     02  DATI             PIC X(67).
002500     02  ANNO-MESE-GIORNO.
002600         03  ANNO         PIC 9999.
002700         03  MESE         PIC 99.
002800         03  GIORNO       PIC 99.
002900*
003000 WORKING-STORAGE SECTION.
003100*
003200 01  CAMPI-SCALARI.
003300     02  EOJ              PIC 9     COMP VALUE IS 0.
003400*
003500 PROCEDURE DIVISION.
003600*------------------------ LIVELLO 0 -----------------
003700 MAIN.
003800     OPEN EXTEND FILE-AGGIORNAMENTI.
```

```
003900     PERFORM INSERIMENTO-DATI UNTIL EOJ = 1.
004000     CLOSE FILE-AGGIORNAMENTI.
004100     STOP RUN.
004200*------------------------ LIVELLO 1 -----------------
004300 INSERIMENTO-DATI.
004400     DISPLAY "INSERISCI IN SEQUENZA: ",
004450             "LA CHIAVE, I DATI DEL ",
004500             "RECORD E LA DATA DI ",
004550             "INSERIMENTO. LA DATA SI ",
004600             "SCRIVE SECONDO IL FORMATO AAAAMMGG".
004700     ACCEPT CHIAVE.
004800     ACCEPT DATI.
004900     ACCEPT ANNO-MESE-GIORNO.
005000     IF CHIAVE = SPACES
005100       THEN
005200           MOVE 1 TO EOJ,
005300       ELSE
005400           WRITE RECORD-AGGIORNAMENTI.
005500*
```

73.3.13 AGO-83-16: aggiornamento di un file a indice

Questo esempio utilizza il file sequenziale del programma precedente, per aggiornare i record di un file a indice (che deve essere già esistente). Questo esempio funziona correttamente utilizzando il compilatore TinyCOBOL 0.61.

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/AGO-83-16.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   AGO-83-16.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-08.
```

```
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 INPUT-OUTPUT SECTION.
000900*
001000 FILE-CONTROL.
001100*
001200     SELECT FILE-AGGIORNAMENTI ASSIGN TO "agg.seq"
001300            ORGANIZATION IS SEQUENTIAL.
001400*
001500     SELECT FILE-DA-AGGIORNARE ASSIGN TO "agg.ind"
001600            ORGANIZATION IS INDEXED,
001700            ACCESS MODE IS RANDOM,
001800            RECORD KEY IS CHIAVE-K.
001900*
002000 DATA DIVISION.
002100*
002200 FILE SECTION.
002300*
002400 FD  FILE-AGGIORNAMENTI
002500     LABEL RECORD IS STANDARD.
002600*
002700 01  RECORD-AGGIORNAMENTI.
002800     02  CHIAVE           PIC X(5).
002900     02  DATI             PIC X(67).
003000     02  ANNO-MESE-GIORNO.
003100         03  ANNO         PIC 9999.
003200         03  MESE         PIC 99.
003300         03  GIORNO       PIC 99.
003400*
003500 FD  FILE-DA-AGGIORNARE
003600     LABEL RECORD IS STANDARD.
003700*
003800 01  RECORD-DA-AGGIORNARE.
```

```
003900     02  CHIAVE-K          PIC X(5).
004000     02  DATI              PIC X(67).
004100     02  ANNO-MESE-GIORNO.
004200         03  ANNO          PIC 9999.
004300         03  MESE          PIC 99.
004400         03  GIORNO        PIC 99.
004500*
004600 WORKING-STORAGE SECTION.
004700*
004800 01  CAMPI-SCALARI.
004900     02  EOF               PIC 9     COMP VALUE IS 0.
005000     02  INV-KEY           PIC 9     COMP VALUE IS 0.
005100*
005200 PROCEDURE DIVISION.
005300*------------------------ LIVELLO 0 -----------------
005400 MAIN.
005500     OPEN INPUT FILE-AGGIORNAMENTI.
005600     OPEN I-O   FILE-DA-AGGIORNARE.
005700     PERFORM LEGGI-FILE-AGGIORNAMENTI.
005800     PERFORM ELABORAZIONE
005900             UNTIL EOF = 1.
006000     CLOSE FILE-AGGIORNAMENTI.
006100     CLOSE FILE-DA-AGGIORNARE
006200     STOP RUN.
006300*------------------------ LIVELLO 1 -----------------
006400 ELABORAZIONE.
006500     MOVE ZERO TO INV-KEY.
006600     READ FILE-DA-AGGIORNARE
006700          INVALID KEY
006800                    MOVE 1 TO INV-KEY.
006900     IF INV-KEY = 1
007000       THEN
007100           PERFORM WRITE-FILE-DA-AGGIORNARE;
007200       ELSE
```

```
007300           IF ANNO-MESE-GIORNO
007350              OF RECORD-AGGIORNAMENTI
007400              > ANNO-MESE-GIORNO
007450                OF RECORD-DA-AGGIORNARE
007500             THEN
007600                 PERFORM REWRITE-FILE-DA-AGGIORNARE.
007700     PERFORM LEGGI-FILE-AGGIORNAMENTI.
007800*-------------------------------------------------
007900 LEGGI-FILE-AGGIORNAMENTI.
008000     READ FILE-AGGIORNAMENTI
008100          AT END MOVE 1 TO EOF.
008200     IF NOT EOF = 1
008300       THEN
008400           MOVE CHIAVE TO CHIAVE-K.
008500*------------------------ LIVELLO 2 ----------------
008600 WRITE-FILE-DA-AGGIORNARE.
008700     WRITE RECORD-DA-AGGIORNARE
008750           FROM RECORD-AGGIORNAMENTI
008800           INVALID KEY
008900                    DISPLAY "ERRORE NON PREVISTO 1".
009000*-------------------------------------------------
009100 REWRITE-FILE-DA-AGGIORNARE.
009200     REWRITE RECORD-DA-AGGIORNARE
009250             FROM RECORD-AGGIORNAMENTI
009300           INVALID KEY
009400                    DISPLAY "ERRORE NON PREVISTO 2".
009500*
```

73.3.14 AGO-83-18: fusione tra due file sequenziali ordinati

«

Il programma seguente richiede la presenza di due file sequenziali, ordinati, denominati rispettivamente ‘`file-ord-1.seq`’ e ‘`file-ord-2.seq`’. Per creare questi file si può usare il programma

‘**AGO-83-1**’, avendo cura di inserire una sequenza di record ordinati per codice, modificando poi il nome del file, una volta come ‘`file-ord-1.seq`’ e un’altra volta come ‘`file-ord-2.seq`’.

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/AGO-83-18.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   AGO-83-18.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 1983-06.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 INPUT-OUTPUT SECTION.
000900*
001000 FILE-CONTROL.
001100*
001200     SELECT FILE-ORD-1 ASSIGN TO "file-ord-1.seq"
001300                       ORGANIZATION IS SEQUENTIAL.
001400     SELECT FILE-ORD-2 ASSIGN TO "file-ord-2.seq"
001500                       ORGANIZATION IS SEQUENTIAL.
001600     SELECT FILE-MERGE ASSIGN TO "file-merge.seq"
001700                       ORGANIZATION IS SEQUENTIAL.
001800*
001900 DATA DIVISION.
002000*
002100 FILE SECTION.
002200*
002300 FD  FILE-ORD-1
002400     LABEL RECORD IS STANDARD.
002500*
002600 01  RECORD-ORD-1.
002700     02  CODICE-1          PIC 9(10) COMP.
002800     02  FILLER            PIC X(75).
002900*
```

```
003000 FD  FILE-ORD-2
003100     LABEL RECORD IS STANDARD.
003200*
003300 01  RECORD-ORD-2.
003400     02  CODICE-2          PIC 9(10) COMP.
003500     02  FILLER            PIC X(75).
003600*
003700 FD  FILE-MERGE
003800     LABEL RECORD IS STANDARD.
003900*
004000 01  RECORD-MERGE          PIC X(80).
004100*
004200 WORKING-STORAGE SECTION.
004300*
004400 01  CAMPI-SCALARI.
004500     02  EOF-1             PIC 9     COMP VALUE IS 0.
004600     02  EOF-2             PIC 9     COMP VALUE IS 0.
004700*
004800 PROCEDURE DIVISION.
004900*------------------------ LIVELLO 0 -----------------
005000 MAIN.
005100     OPEN INPUT  FILE-ORD-1.
005200     OPEN INPUT  FILE-ORD-2.
005300     OPEN OUTPUT FILE-MERGE.
005400     PERFORM LETTURA-FILE-ORD-1.
005500     PERFORM LETTURA-FILE-ORD-2.
005600     PERFORM ELABORAZIONE
005700             UNTIL EOF-1 = 1 AND EOF-2 = 1.
005800     CLOSE FILE-MERGE.
005900     CLOSE FILE-ORD-2.
006000     CLOSE FILE-ORD-1.
006100     STOP RUN.
006200*------------------------ LIVELLO 1 -----------------
006300 ELABORAZIONE.
```

```
006400     IF       (CODICE-1 <= CODICE-2 AND EOF-1 = 0)
006450              OR EOF-2 = 1
006500     THEN
006600         MOVE RECORD-ORD-1 TO RECORD-MERGE,
006700         WRITE RECORD-MERGE,
006800         PERFORM LETTURA-FILE-ORD-1;
006900     ELSE IF (CODICE-1 > CODICE-2 AND EOF-2 = 0)
006950              OR EOF-1 = 1
007000     THEN
007100         MOVE RECORD-ORD-2 TO RECORD-MERGE,
007200         WRITE RECORD-MERGE,
007300         PERFORM LETTURA-FILE-ORD-2;
007400     ELSE
007500         DISPLAY "ERRORE NON PREVISTO".
007600*------------------------ LIVELLO 2 -----------------
007700 LETTURA-FILE-ORD-1.
007800     READ FILE-ORD-1
007900          AT END
008000               MOVE 1 TO EOF-1.
008100*-------------------------------------------------
008200 LETTURA-FILE-ORD-2.
008300     READ FILE-ORD-2
008400          AT END
008500               MOVE 1 TO EOF-2.
008600*
```

73.3.15 AGO-83-20: riordino attraverso la fusione

Il programma seguente utilizza un file sequenziale, non ordinato, denominato ‘`file-in.seq`’, per generare il file ‘`file-out.seq`’ ordinato, utilizzando due file temporanei: ‘`file-tmp-1.seq`’ e ‘`file-tmp-2.seq`’. Per creare il file ‘`file-in.seq`’, si può usare

il programma '**AGO-83-1**', modificando poi il nome come richiesto in questo esempio.

Nella sezione 62.6.2 viene descritto il problema del riordino ottenuto attraverso la suddivisione in blocchi del file e la fusione successiva.

Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/AGO-83-20.cob* .

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.   AGO-83-20.
000300 AUTHOR.       DANIELE GIACOMINI.
000400 DATE-WRITTEN. 2005-03-29.
000500*
000600 ENVIRONMENT DIVISION.
000700*
000800 INPUT-OUTPUT SECTION.
000900*
001000 FILE-CONTROL.
001100*
001200     SELECT FILE-IN    ASSIGN TO "file-in.seq"
001300                       ORGANIZATION IS SEQUENTIAL.
001400     SELECT FILE-TMP-1 ASSIGN TO "file-tmp-1.seq"
001500                       ORGANIZATION IS SEQUENTIAL.
001600     SELECT FILE-TMP-2 ASSIGN TO "file-tmp-2.seq"
001700                       ORGANIZATION IS SEQUENTIAL.
001800     SELECT FILE-MERGE ASSIGN TO "file-out.seq"
001900                       ORGANIZATION IS SEQUENTIAL.
002000*
002100 DATA DIVISION.
002200*
002300 FILE SECTION.
002400*
002500 FD  FILE-IN
002600     LABEL RECORD IS STANDARD.
002700*
```

```
002800 01  RECORD-IN.
002900     02  CODICE-IN          PIC 9(10) COMP.
003000     02  FILLER             PIC X(75).
003100*
003200 FD  FILE-TMP-1
003300     LABEL RECORD IS STANDARD.
003400*
003500 01  RECORD-TMP-1.
003600     02  CODICE-T1          PIC 9(10) COMP.
003700     02  FILLER             PIC X(75).
003800*
003900 FD  FILE-TMP-2
004000     LABEL RECORD IS STANDARD.
004100*
004200 01  RECORD-TMP-2.
004300     02  CODICE-T2          PIC 9(10) COMP.
004400     02  FILLER             PIC X(75).
004500*
004600 FD  FILE-MERGE
004700     LABEL RECORD IS STANDARD.
004800*
004900 01  RECORD-MERGE.
005000     02  CODICE-MERGE       PIC 9(10) COMP.
005100     02  FILLER             PIC X(75).
005200*
005300 WORKING-STORAGE SECTION.
005400*
005500 01  CAMPI-SCALARI.
005600     02  EOF                PIC 9     COMP VALUE IS 0.
005700     02  EOF-1              PIC 9     COMP VALUE IS 0.
005800     02  EOF-2              PIC 9     COMP VALUE IS 0.
005900     02  EOB-1              PIC 9     COMP VALUE IS 0.
006000     02  EOB-2              PIC 9     COMP VALUE IS 0.
006100     02  BIFORCAZIONI       PIC 9(10) COMP VALUE IS 0.
```

```
006200     02  CODICE-ORIG      PIC 9(10) COMP VALUE IS 0.
006300     02  CODICE-ORIG-1    PIC 9(10) COMP VALUE IS 0.
006400     02  CODICE-ORIG-2    PIC 9(10) COMP VALUE IS 0.
006500     02  SCAMBIO          PIC 9     COMP VALUE IS 0.
006600*
006700 PROCEDURE DIVISION.
006800*------------------------ LIVELLO 0 -----------------
006900 MAIN.
007000     PERFORM COPIA-FILE-MERGE.
007100     PERFORM BIFORCAZIONE.
007200     IF BIFORCAZIONI > 0
007300       THEN
007400           PERFORM FUSIONE,
007500           PERFORM BIFORCAZIONE-E-FUSIONE
007600                   UNTIL BIFORCAZIONI <= 2.
007700     STOP RUN.
007800*------------------------ LIVELLO 1 -----------------
007900 COPIA-FILE-MERGE.
008000     OPEN INPUT  FILE-IN.
008100     OPEN OUTPUT FILE-MERGE.
008200     MOVE ZERO TO EOF.
008300     PERFORM LETTURA-FILE-IN.
008400     PERFORM COPIA-RECORD-FILE-MERGE
008500             UNTIL EOF = 1.
008600     CLOSE FILE-MERGE.
008700     CLOSE FILE-IN.
008800*-----------------------------------------------------
008900 BIFORCAZIONE-E-FUSIONE.
009000     PERFORM BIFORCAZIONE.
009100     PERFORM FUSIONE.
009200*------------------------ LIVELLO 2 -----------------
009300 COPIA-RECORD-FILE-MERGE.
009400     MOVE RECORD-IN TO RECORD-MERGE.
009500     WRITE RECORD-MERGE.
```

```
009600     PERFORM LETTURA-FILE-IN.
009700*-----------------------------------------------
009800 BIFORCAZIONE.
009900     MOVE ZERO TO BIFORCAZIONI.
010000     OPEN INPUT  FILE-MERGE.
010100     OPEN OUTPUT FILE-TMP-1.
010200     OPEN OUTPUT FILE-TMP-2.
010300     MOVE ZERO TO EOF.
010400     MOVE 1 TO SCAMBIO.
010500     PERFORM LETTURA-FILE-MERGE.
010600     IF EOF = 0
010700       THEN
010800           ADD 1 TO BIFORCAZIONI,
010900           MOVE RECORD-MERGE TO RECORD-TMP-1,
011000           WRITE RECORD-TMP-1,
011100           MOVE CODICE-MERGE TO CODICE-ORIG,
011200           PERFORM LETTURA-FILE-MERGE.
011300     PERFORM BIFORCAZIONE-SUCCESSIVA
011400             UNTIL EOF = 1.
011500     CLOSE FILE-TMP-2.
011600     CLOSE FILE-TMP-1.
011700     CLOSE FILE-MERGE.
011800*-----------------------------------------------
011900 FUSIONE.
012000     OPEN INPUT  FILE-TMP-1.
012100     OPEN INPUT  FILE-TMP-2.
012200     OPEN OUTPUT FILE-MERGE.
012300     MOVE ZERO TO EOF-1.
012400     MOVE ZERO TO EOF-2.
012500     MOVE ZERO TO EOB-1.
012600     MOVE ZERO TO EOB-2.
012700     PERFORM LETTURA-FILE-TMP-1.
012800     IF EOF-1 = 0 AND EOB-1 = 0
012900       THEN
```

```
013000           MOVE CODICE-T1 TO CODICE-ORIG-1.
013100     PERFORM LETTURA-FILE-TMP-2.
013200     IF EOF-2 = 0 AND EOB-2 = 0
013300       THEN
013400           MOVE CODICE-T2 TO CODICE-ORIG-2.
013500     PERFORM FUSIONE-SUCCESSIVA
013600             UNTIL EOF-1 = 1 AND EOF-2 = 1.
013700     CLOSE FILE-MERGE.
013800     CLOSE FILE-TMP-2.
013900     CLOSE FILE-TMP-1.
014000*------------------------ LIVELLO 3 -----------------
014100 BIFORCAZIONE-SUCCESSIVA.
014200     IF CODICE-MERGE >= CODICE-ORIG
014300       THEN
014400           IF SCAMBIO = 1
014500             THEN
014600                 MOVE RECORD-MERGE TO RECORD-TMP-1,
014700                 WRITE RECORD-TMP-1,
014800                 MOVE CODICE-MERGE TO CODICE-ORIG,
014900                 PERFORM LETTURA-FILE-MERGE;
015000             ELSE
015100                 MOVE RECORD-MERGE TO RECORD-TMP-2,
015200                 WRITE RECORD-TMP-2,
015300                 MOVE CODICE-MERGE TO CODICE-ORIG,
015400                 PERFORM LETTURA-FILE-MERGE;
015500       ELSE
015600           ADD 1 TO BIFORCAZIONI,
015700           MOVE CODICE-MERGE TO CODICE-ORIG,
015800           IF SCAMBIO = 1
015900             THEN
016000                 MOVE 2 TO SCAMBIO;
016100             ELSE
016200                 MOVE 1 TO SCAMBIO.
016300*-----------------------------------------------
```

```
016400 FUSIONE-SUCCESSIVA.
016500     PERFORM FUSIONE-BLOCCO
016600             UNTIL EOB-1 = 1 AND EOB-2 = 1.
016700     IF NOT EOF-1 = 1
016800       THEN
016900           MOVE ZERO TO EOB-1.
017000     IF NOT EOF-2 = 1
017100       THEN
017200           MOVE ZERO TO EOB-2.
017300*------------------------ LIVELLO 4 -----------------
017400 FUSIONE-BLOCCO.
017500     IF EOB-1 = 1
017600       THEN
017700           MOVE RECORD-TMP-2 TO RECORD-MERGE,
017800           PERFORM LETTURA-FILE-TMP-2;
017900       ELSE
018000           IF EOB-2 = 1
018100             THEN
018200                 MOVE RECORD-TMP-1 TO RECORD-MERGE,
018300                 PERFORM LETTURA-FILE-TMP-1;
018400             ELSE
018500                 IF CODICE-T1 < CODICE-T2
018600                   THEN
018700                       MOVE RECORD-TMP-1
018750                            TO RECORD-MERGE,
018800                       PERFORM LETTURA-FILE-TMP-1;
018900                       IF EOF-1 = 0 AND EOB-1 = 0
019000                         THEN
019100                             IF CODICE-T1
019150                                >= CODICE-ORIG-1
019200                               THEN
019300                                   MOVE CODICE-T1
019400                                     TO CODICE-ORIG-1;
019500                               ELSE
```

```
019600                                    MOVE 1 TO EOB-1;
019700                              ELSE
019800                                  NEXT SENTENCE;
019900                         ELSE
020000                             MOVE RECORD-TMP-2
020050                                  TO RECORD-MERGE,
020100                             PERFORM LETTURA-FILE-TMP-2;
020200                             IF EOF-2 = 0 AND EOB-2 = 0
020300                               THEN
020400                                   IF CODICE-T2
020450                                      >= CODICE-ORIG-2
020500                                     THEN
020600                                         MOVE CODICE-T2
020700                                           TO CODICE-ORIG-2;
020800                                     ELSE
020900                                         MOVE 1 TO EOB-2.
021000     WRITE RECORD-MERGE.
021200*------------------------ LIVELLO 5 -----------------
021300 LETTURA-FILE-IN.
021400     READ FILE-IN
021500          AT END
021600               MOVE 1 TO EOF.
021700*-------------------------------------------------
021800 LETTURA-FILE-MERGE.
021900     READ FILE-MERGE
022000          AT END
022100               MOVE 1 TO EOF.
022200*-------------------------------------------------
022300 LETTURA-FILE-TMP-1.
022400     READ FILE-TMP-1
022500          AT END
022600               MOVE 1 TO EOF-1,
022700               MOVE 1 TO EOB-1.
022800*-------------------------------------------------
```

```
022900 LETTURA-FILE-TMP-2.
023000     READ FILE-TMP-2
023100          AT END
023200               MOVE 1 TO EOF-2,
023300               MOVE 1 TO EOB-2.
023400*
```

73.4 Approfondimento: una tecnica per simulare la ricorsione in COBOL

Questa sezione contiene la ricostruzione di un documento con lo stesso nome, concluso nel mese di giugno del 1985, dopo un periodo di studio sul linguaggio COBOL. Il COBOL è un linguaggio procedurale che offre esclusivamente la gestione di variabili globali, pertanto non consente di realizzare la ricorsione; tuttavia, qui, come esercizio, si descrive una tecnica per arrivare a ottenere un risultato simile alla ricorsione comune.

Si fa riferimento a tre algoritmi noti: torre di Hanoi, quicksort e permutazioni. Questi algoritmi sono descritti nella sezione 62.2.

Al termine è riportata la bibliografia dello studio originale. Tutti gli esempi originali con il linguaggio MPL II sono omessi, anche se nella bibliografia questo linguaggio viene citato.

73.4.1 Il concetto di locale e di globale

Niklaus Wirth **[1]** spiega molto bene la differenza tra il concetto di ***locale*** e di ***globale*** all'interno di un programma:

> *Se un oggetto –una costante, una variabile, una procedura, una funzione o un tipo– è significativo solo all'interno di*

una parte determinata del programma, viene chiamato «locale». Spesso conviene rappresentare questa parte mediante una procedura; gli oggetti locali vengono allora indicati nel titolo della procedura. Dato che le procedure stesse possono essere locali, può accadere che più indicazioni di procedura siano innestate l'una nell'altra.

Nell'ambito della procedura si possono quindi riconoscere due tipi di oggetti: gli oggetti «locali» e gli oggetti «non locali». Questi ultimi sono oggetti definiti nel programma (o nella procedura) in cui è inserita la procedura («ambiente» della procedura). Se sono definiti nel programma principale, sono detti «globali». In una procedura il campo di influenza degli oggetti locali corrisponde al corpo della procedura. In particolare, terminata l'esecuzione della procedura, le variabili locali saranno ancora disponibili per indicare dei nuovi valori; chiaramente, in una chiamata successiva della stessa procedura, i valori delle variabili locali saranno diversi da quelli della chiamata precedente.

È essenziale che i nomi degli oggetti locali non debbano dipendere dall'ambiente della procedura. Ma, in tal modo, può accadere che un nome «x», scelto per un oggetto locale della procedura «P», sia identico a quello di un oggetto definito nel programma ambiente di «P». Questa situazione però è corretta solo se la grandezza non locale «x» non è significativa per «P», cioè non viene applicata in «P». Si adotta quindi la «regola fondamentale» che «x» denoti entro «P» la grandezza locale e fuori da «P» quella non locale.

73.4.2 La ricorsione

«La ricorsione», come spiegano Ledgard, Nagin e Hueras **[2]**, «è un metodo di definizione in cui l'oggetto della definizione è usato all'interno della definizione». Per esempio si può considerare la seguente definizione della parola «discendente»:

> Un discendente di una persona è il figlio o la figlia di quella persona, o un discendente del figlio o della figlia.

Quindi, come scrive Lawrie Moore **[3]**, un sottoprogramma ricorsivo «è un sottoprogramma che corrisponde direttamente e utilizza una definizione ricorsiva». Ovvero, molto più semplicemente come dicono Aho, Hopcroft e Ullman **4**: «Una procedura che chiama se stessa, direttamente o indirettamente, si dice essere ricorsiva».

Moore **[3]** inoltre aggiunge quanto segue: «La chiamata genera un nuovo blocco di programma, con il suo proprio ambito, il suo proprio spazio di lavoro, la sua propria esistenza virtuale. [...] Questo processo prende luogo al momento dell'esecuzione del programma (run-time). Al momento della compilazione né la macchina, né l'intelligenza umana possono dire quante volte la procedura sarà richiamata al momento dell'esecuzione. Perciò, la creazione di un nuovo blocco di programma al momento dell'esecuzione è un processo dinamico. La creazione ricorsiva di nuovi blocchi di programma è una struttura di programmazione dinamica».

73.4.3 Proprietà del linguaggio ricorsivo

«

La definizione di procedura ricorsiva data da Aho, Hopcroft e Ullman è una condizione necessaria ma non sufficiente perché un linguaggio di programmazione possa definirsi ricorsivo. Infatti, è tale quel linguaggio che oltre a permettere la chiamata di una procedura da parte di se stessa, permette una dichiarazione locale delle variabili, ovvero permette l'allocazione dinamica delle variabili stesse.

Non vi è dubbio che il linguaggio COBOL non sia ricorsivo, eppure ammette che all'interno di un paragrafo si faccia la chiamata dello stesso paragrafo tramite l'istruzione '**`PERFORM`**'. In effetti non si parla di ricorsione proprio perché il COBOL gestisce solo variabili globali.

73.4.4 Descrizione della tecnica per simulare la ricorsione in COBOL

«

Le variabili di scambio di un sottoprogramma possono collegarsi all'esterno, a seconda del contesto del programma, in tre modi: in input, in output o in input-output, a seconda che importi che i dati entrino nel sottoprogramma ma non escano, che i dati escano soltanto oppure che i dati debbano prima entrare e poi uscire modificati.

La pseudocodifica utilizzata per mostrare gli esempi, prima di presentare la trasformazione in COBOL, si rifà al linguaggio MPL II Burroughs, dove le variabili di scambio di una procedura vengono semplicemente nominate a fianco del nome della procedura tra parentesi. Ciò corrisponde a una dichiarazione implicita di quelle variabili con ambito locale e con caratteristiche identiche a quelle

usate nelle chiamate relative. In particolare, se nella chiamata vengono usate costanti alfanumeriche, la variabile corrispondente sarà di tipo alfanumerico di lunghezza pari alla costante trasmittente, se di tipo numerico, la variabile corrispondente sarà di tipo numerico opportuno: intero o a virgola mobile.
Quindi, in questo tipo di pseudocodifica non sono permesse le variabili di scambio in output.
Le variabili di scambio di questa pseudocodifica si collegano per posizione.

Il problema della simulazione della ricorsione si risolve utilizzando una pila (*stack*) per ogni variabile locale.
La tecnica è indicata molto semplicemente da Jerrold L. Wagener **[5]**. Una volta determinato a priori qual è il numero massimo di livelli della ricorsione, occorre associare a ogni variabile locale, che non sia collegata con l'esterno in input-output, una pila con dimensioni pari a quel numero. Quindi, a una variabile scalare viene associato un vettore, a un vettore viene associata una matrice a due dimensioni e così di seguito. L'indice della pila (*stack pointer*) viene indicato con '**`SP`**'.
La simulazione si divide in due fasi: la prima deve essere effettuata subito prima della chiamata ricorsiva e consiste nella conservazione delle varie pile dei valori delle variabili di scambio che non sono in input-output con un'operazione di inserimento (*push*); la seconda deve essere effettuata subito dopo la chiamata ricorsiva e consiste nel recupero dalle varie pile dei valori originali delle variabili con un'operazione di estrazione (*pop*).

Figura 73.39. Confronto tra una procedura ricorsiva e la sua trasformazione non ricorsiva, attraverso la pseudocodifica.

```
#
# Procedura ricorsiva
#
PROC1 (V, G, Z)
    .
    .
    .
    .
    .
    .
    .
    # G è una variabile in
    # input-output
    PROC1 (V, G, Z-1)
    .
    .
    .
    .
    .
    .
    .
END PROC1
```

```
#
# Trasformazione non ricorsiva
#
PROC1
    .
    .
    .
    # push
    SP := SP + 1
    SAVEV(SP) := V
    SAVEZ(SP) := Z
    # chiamata
    Z := Z -1
    PROC1
    # pop
    V := SAVEV(SP)
    Z := SAVEZ(SP)
    SP := SP - 1
    .
    .
    .
END PROC1
```

È bene precisare che la tecnica è valida solo se all'interno di una procedura ricorsiva tutte le iterazioni che contengono una chiamata (diretta o indiretta) alla stessa procedura sono a loro volta espresse in forma ricorsiva (si veda il problema delle permutazioni).

73.4.5 Torre di Hanoi

Segue la descrizione dell'algoritmo attraverso la pseudocodifica in forma ricorsiva. Nella sezione 62.5.3 viene descritto il problema della torre di Hanoi.

Variabile	Descrizione
`N`	È la dimensione della torre espressa in numero di anelli: gli anelli sono numerati da 1 a '**N**'.
`P1`	È il numero del piolo su cui si trova inizialmente la pila di '**N**' anelli.
`P2`	È il numero del piolo su cui deve essere spostata la pila di anelli.
`6-P1-P2`	È il numero dell'altro piolo. Funziona così se i pioli sono numerati da 1 a 3.

```
HANOI (N, P1, P2)
    IF N > 0
        THEN
            HANOI (N-1, P1, 6-P1-P2)
            scrivi: "Muovi l'anello" N "dal piolo" P1 "al piolo" P2
            HANOI (N-1, 6-P1-P2, P2)
    END IF
END HANOI
```

Segue la descrizione della trasformazione in modo tale da simulare la ricorsione.

Variabile	Descrizione
`SAVEN`	È il vettore utilizzato per conservare il valore di '**N**'.
`SAVEP1`	È il vettore utilizzato per conservare il valore di '**P1**'.

Variabile	Descrizione
`SAVEP2`	È il vettore utilizzato per conservare il valore di '**`P2`**'.
`SP`	È l'indice dei vettori usati per salvare i valori (*stack pointer*).

```
HANOI (N, P1, P2)
    IF N > 0
        THEN
            SP := SP + 1
            SAVEN(SP) := N
            SAVEP2(SP) := P2
            N := N - 1
            P2 := 6 - P1 - P2
            HANOI
            N := SAVEN(SP)
            P2 := SAVEP2(SP)
            SP = SP - 1
            scrivi: "Muovi l'anello" N "dal piolo" P1 "al piolo" P2
            SP := SP + 1
            SAVEN(SP) := N
            SAVEP1(SP) := P1
            N := N - 1
            P1 := 6 - P1 - P2
            HANOI
            N := SAVEN(SP)
            P1 := SAVEP1(SP)
            SP = SP - 1
    END IF
END HANOI
```

Listato 73.44. Soluzione in COBOL del problema della torre di Hanoi, con la simulazione della ricorsione. Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/HC04.cob* .

```
000600 IDENTIFICATION DIVISION.
000700 PROGRAM-ID.   HC04.
000800 AUTHOR.       DANIELE GIACOMINI.
000900 DATE-WRITTEN. 1984-08-18.
001000
001100
001200 ENVIRONMENT DIVISION.
001300
001400
001500 DATA DIVISION.
001600
001700
001800 WORKING-STORAGE SECTION.
001900
002000 01  RECORD-STACKS.
002100     02  SAVEN  OCCURS 100 TIMES PIC 99.
002200     02  SAVEP1 OCCURS 100 TIMES PIC 9.
002300     02  SAVEP2 OCCURS 100 TIMES PIC 9.
002400
002500 01  STACK-POINTER.
002600     02  SP                      PIC 99 VALUE 0.
002700
002800 01  VARIABILI-SCALARI.
002900     02  N                       PIC 99.
003000     02  P1                      PIC 9.
003100     02  P2                      PIC 9.
003200
003300
003400 PROCEDURE DIVISION.
003500
003600 MAIN.
```

```
003700
003800     DISPLAY "INSERISCI LA DIMENSIONE DELLA TORRE".
003900     DISPLAY "(DUE CARATTERI)".
004000     ACCEPT N.
004100
004200     DISPLAY "INSERISCI LA POSIZIONE INIZIALE ",
004250             "DELLA TORRE".
004300     DISPLAY "(UN CARATTERE)".
004400     ACCEPT P1.
004500
004600     DISPLAY "INSERISCI LA DESTINAZIONE DELLA TORRE".
004700     DISPLAY "(UN CARATTERE)".
004800     ACCEPT P2.
004900
005000     PERFORM HANOI.
005100
005200     STOP RUN.
005300
005400 HANOI.
005500
005600     IF N > 0
005700       THEN
005800*
005900*          push per conservare le variabili di scambio
006000*
006100           COMPUTE SP = SP + 1,
006200           COMPUTE SAVEN(SP) = N,
006300           COMPUTE SAVEP2(SP) = P2,
006400*
006500*          cambiamenti alle variabili di scambio prima
006600*          della chiamata
006700*
006800           COMPUTE N = N - 1,
006900           COMPUTE P2 = 6 - P1 - P2,
```

```
007000*
007100*         chiamata della procedura
007200*
007300          PERFORM HANOI,
007400*
007500*         pop per recuperare i valori delle variabili
007550*         di scambio
007600*
007700          COMPUTE P2 = SAVEP2(SP),
007800          COMPUTE N = SAVEN(SP),
007900          COMPUTE SP = SP - 1,
008000
008100          DISPLAY "MUOVI L'ANELLO ", N,
008150                  " DAL PIOLO ", P1,
008200                  " AL PIOLO ", P2,
008300
008400*
008500*         push per conservare le variabili di scambio
008600*
008700          COMPUTE SP = SP + 1,
008800          COMPUTE SAVEN(SP) = N,
008900          COMPUTE SAVEP1(SP) = P1,
009000*
009100*         modifica dei valori delle variabili di
009159*         scambio
009200*
009300          COMPUTE N = N - 1,
009400          COMPUTE P1 = 6 - P1 - P2,
009500*
009600*         chiamata della procedura
009700*
009800          PERFORM HANOI,
009900*
010000*         pop per recuperare i valori delle variabili
```

```
010050*          di scambio
010100*
010200           COMPUTE P1 = SAVEP1(SP),
010300           COMPUTE N = SAVEN(SP),
010400           COMPUTE SP = SP - 1.
010500
```

73.4.6 Quicksort (ordinamento non decrescente)

«

Segue la descrizione dell'algoritmo attraverso la pseudocodifica in forma ricorsiva; si ricorda che l'algoritmo del Quicksort si risolve con due subroutine: una serve a suddividere il vettore; l'altra esegue le chiamate ricorsive. Nella sezione 62.5.4 viene descritto il problema del Quicksort in modo dettagliato.

Variabile	Descrizione
`LISTA`	L'array da ordinare in modo crescente.
`A`	L'indice inferiore del segmento di array da ordinare.
`Z`	L'indice superiore del segmento di array da ordinare.
`CF`	Sta per «collocazione finale» ed è l'indice che cerca e trova la posizione giusta di un elemento nell'array.
`I`	È l'indice che insieme a '**`CF`**' serve a ripartire l'array.

```
PART (LISTA, A, Z)

    LOCAL I INTEGER
    LOCAL CF INTEGER

    # si assume che A < U
```

```
I := A + 1
CF := Z

WHILE TRUE # ciclo senza fine.

    WHILE TRUE

        # sposta I a destra

        IF (LISTA[I] > LISTA[A]) OR I >= CF
            THEN
                BREAK
            ELSE
                I := I + 1
        END IF

    END WHILE

    WHILE TRUE

        # sposta CF a sinistra

        IF (LISTA[CF] <= LISTA[A])
            THEN
                BREAK
            ELSE
                CF := CF - 1
        END IF

    END WHILE

    IF CF <= I
        THEN
```

```
                # è avvenuto l'incontro tra I e CF
                BREAK
            ELSE
                # vengono scambiati i valori
                LISTA[CF] :==: LISTA[I]
                I := I + 1
                CF := CF - 1
        END IF

    END WHILE

    # a questo punto LISTA[A:Z] è stata ripartita e CF è
    # la collocazione di LISTA[A]

    LISTA[CF] :==: LISTA[A]

    # a questo punto, LISTA[CF] è un elemento (un valore)
    # nella giusta posizione

    RETURN CF

END PART
```

```
QSORT (LISTA, A, Z)

    LOCAL CF INTEGER

    IF Z > A
        THEN
            CF := PART (@LISTA, A, Z)
            QSORT (@LISTA, A, CF-1)
            QSORT (@LISTA, CF+1, Z)
    END IF
END QSORT
```

Vale la pena di osservare che l'array viene indicato nelle chiamate in modo che alla subroutine sia inviato un riferimento a quello originale, perché le variazioni fatte all'interno delle subroutine devono riflettersi sull'array originale.

La subroutine '**QSORT**' è quella che richiede la trasformazione per la simulazione della ricorsione; tuttavia, anche la subroutine deve essere adattata in modo tale da gestire la variabile '**CF**' come variabile globale (non potendo gestire variabili di '**output**'). Segue la descrizione di tali adattamenti.

Variabile	Descrizione
SAVEA	È il vettore utilizzato per conservare il valore di '**A**'.
SAVEZ	È il vettore utilizzato per conservare il valore di '**Z**'.
SP	È l'indice dei vettori usati per salvare i valori (*stack pointer*).

```
PART (LISTA, A, Z)

    LOCAL I INTEGER

    # si assume che A < U

    I := A + 1
    CF := Z

    WHILE TRUE # ciclo senza fine.

        WHILE TRUE

            # sposta I a destra
```

```
        IF (LISTA[I] > LISTA[A]) OR I >= CF
            THEN
                BREAK
            ELSE
                I := I + 1
        END IF

    END WHILE

    WHILE TRUE

        # sposta CF a sinistra

        IF (LISTA[CF] <= LISTA[A])
            THEN
                BREAK
            ELSE
                CF := CF - 1
        END IF

    END WHILE

    IF CF <= I
        THEN
            # è avvenuto l'incontro tra I e CF
            BREAK
        ELSE
            # vengono scambiati i valori
            LISTA[CF] :==: LISTA[I]
            I := I + 1
            CF := CF - 1
    END IF
```

```
    END WHILE

    # a questo punto LISTA[A:Z] è stata ripartita e CF è
    # la collocazione di LISTA[A]

    LISTA[CF] :==: LISTA[A]

    # a questo punto, LISTA[CF] è un elemento (un valore)
    # nella giusta posizione

END PART
```

```
QSORT
    IF Z > A
        THEN
            PART
            SP := SP + 1
            SAVEZ(SP) := Z
            Z := CF - 1
            QSORT
        #   SP := SP - 1
        #   SP := SP + 1
            SAVEA(SP) := A
            A := CF + 1
            QSORT
            A := SAVEA(SP)
            SP := SP - 1
    END IF
END QSORT
```

Listato 73.51. Soluzione in COBOL del problema del Quicksort, con la simulazione della ricorsione. Si osservi che '**CF**' è una parola riservata del linguaggio, pertanto viene sostituita con '**C-F**'. Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/HC06.cob* .

```
000600 IDENTIFICATION DIVISION.
000700 PROGRAM-ID.   HC06.
000800 AUTHOR.       DANIELE GIACOMINI.
000900 DATE-WRITTEN. 1984-08-22.
001000
001100
001200 ENVIRONMENT DIVISION.
001300
001400
001500 DATA DIVISION.
001600
001700
001800 WORKING-STORAGE SECTION.
001900
002000 01  RECORD-STACKS.
002100     02  SAVEA   OCCURS 100 TIMES PIC 999.
002200     02  SAVEZ   OCCURS 100 TIMES PIC 999.
002300
002400 01  STACK-POINTER.
002500     02  SP                       PIC 999.
002600
002700 01  VARIABILI-SCALARI.
002800     02  C-F                      PIC 999.
002900     02  A                        PIC 999.
003000     02  Z                        PIC 999.
003100     02  TEMP                     PIC X(15).
003200     02  I                        PIC 999.
003300     02  J                        PIC 999.
```

```
003400
003500 01  RECORD-TABELLA.
003600     02  TABELLA OCCURS 100 TIMES PIC X(15).
003700
003800 PROCEDURE DIVISION.
003900
004000 MAIN.
004100
004200     DISPLAY "INSERISCI IL NUMERO DI ELEMENTI ",
004250             "DA ORDINARE".
004300     DISPLAY "(TRE CIFRE)".
004400     ACCEPT Z.
004500     IF Z > 100
004600       THEN
004700           STOP RUN.
004800
004900     COMPUTE A = 1.
005000
005100     PERFORM INSERIMENTO-ELEMENTI
005150             VARYING J FROM 1 BY 1
005200             UNTIL   J > Z.
005300
005400     PERFORM QSORT.
005500
005600     PERFORM OUTPUT-DATI VARYING J FROM 1 BY 1
005700                         UNTIL   J > Z.
005800
005900     STOP RUN.
006000
006100
006200 INSERIMENTO-ELEMENTI.
006300
006400     DISPLAY "INSERISCI L'ELEMENTO ", J,
006450             " DELLA TABELLA".
```

```
006500     ACCEPT TABELLA(J).
006600
006700
006800 PART.
006900
007000*
007100*    si assume che A < Z
007200*
007300     COMPUTE I = A + 1.
007400     COMPUTE C-F = Z.
007500
007600     PERFORM PART-TESTA-MAINLOOP.
007700     PERFORM PART-MAINLOOP UNTIL C-F < I
007800                               OR C-F = I.
007900
008000     MOVE TABELLA(C-F) TO TEMP.
008100     MOVE TABELLA(A)   TO TABELLA(C-F).
008200     MOVE TEMP         TO TABELLA(A).
008300
008400
008500 PART-TESTA-MAINLOOP.
008600
008700     PERFORM SPOSTA-I-A-DESTRA
008750             UNTIL TABELLA(I) > TABELLA(A)
008800                   OR I > C-F
008900                   OR I = C-F.
009000
009100     PERFORM SPOSTA-C-F-A-SINISTRA
009200             UNTIL TABELLA(C-F) < TABELLA(A)
009300                OR TABELLA(C-F) = TABELLA(A).
009400
009500
009600 PART-MAINLOOP.
009700
```

```
009800     MOVE TABELLA(C-F) TO TEMP.
009900     MOVE TABELLA(I)   TO TABELLA(C-F).
010000     MOVE TEMP         TO TABELLA(I).
010100
010200     COMPUTE I = I + 1.
010300     COMPUTE C-F = C-F - 1.
010400
010500     PERFORM SPOSTA-I-A-DESTRA
010550             UNTIL TABELLA(I) > TABELLA(A)
010600                   OR I > C-F
010700                   OR I = C-F.
010800
010900     PERFORM SPOSTA-C-F-A-SINISTRA
011000             UNTIL TABELLA(C-F) < TABELLA(A)
011100                OR TABELLA(C-F) = TABELLA(A).
011200
011300
011400 SPOSTA-I-A-DESTRA.
011500
011600     COMPUTE I = I + 1.
011700
011800
011900 SPOSTA-C-F-A-SINISTRA.
012000
012100     COMPUTE C-F = C-F - 1.
012200
012300
012400 QSORT.
012500
012600     IF Z > A
012700       THEN
012800*
012900*          le variabili che riguardano PART sono tutte
012950*          in I-O
```

```
013000*
013100           PERFORM PART,
013200*
013300*          push
013400*
013500           COMPUTE SP = SP + 1,
013600           COMPUTE SAVEZ(SP) = Z,
013700*
013800*          cambiamenti alle variabili di scambio
013900*
014000           COMPUTE Z = C-F - 1,
014100*
014200*          chiamata
014300*
014400           PERFORM QSORT,
014500*
014600*          pop
014700*
014800           COMPUTE Z = SAVEZ(SP),
014900           COMPUTE SP = SP - 1,
015000*
015100*          push
015200*
015300           COMPUTE SP = SP + 1,
015400           COMPUTE SAVEA(SP) = A,
015500*
015600*          cambiamenti alle variabili di scambio
015700*
015800           COMPUTE A = C-F + 1,
015900*
016000*          chiamata
016100*
016200           PERFORM QSORT,
016300*
```

```
016400*           pop
016500*
016600            COMPUTE A = SAVEA(SP),
016700            COMPUTE SP = SP - 1.
016800
016900
017000 OUTPUT-DATI.
017100
017200      DISPLAY "TABELLA(", J, ") = ", TABELLA(J).
017300
```

73.4.7 Permutazioni

La permutazione degli elementi di un vettore si risolve generalmente attraverso un algoritmo iterativo normale; segue la descrizione dell'algoritmo iterativo in forma di pseudocodifica. Nella sezione 62.5.5 viene descritto il problema delle permutazioni in modo dettagliato.

Variabile	Descrizione
`LISTA`	L'array da permutare.
`A`	L'indice inferiore del segmento di array da permutare.
`Z`	L'indice superiore del segmento di array da permutare.
`K`	È l'indice che serve a scambiare gli elementi.

```
PERMUTA (LISTA, A, Z)

    LOCAL K INTEGER
    LOCAL N INTEGER
```

```
    IF (Z - A) >= 1
        # Ci sono almeno due elementi nel segmento di array.
        THEN
            FOR K := Z; K >= A; K--

                LISTA[K] :==: LISTA[Z]

                PERMUTA (LISTA, A, Z-1)

                LISTA[K] :==: LISTA[Z]

            END FOR
        ELSE
            scrivi LISTA
    END IF

END PERMUTA
```

Per esercizio, l'algoritmo iterativo viene trasformato in modo ricorsivo:

```
PERMUTA (LISTA, A, Z)

    LOCAL K INTEGER
    LOCAL N INTEGER

    SCAMBIO_CHIAMATA_SCAMBIO (LISTA, A, Z, K)
        IF K >= A
            THEN
                LISTA[K] :==: LISTA[Z]
                PERMUTA (LISTA, A, Z-1)
                LISTA[K] :==: LISTA[Z]
                SCAMBIO_CHIAMATA_SCAMBIO (LISTA, A, Z, K - 1)
        END IF
```

```
    END SCAMBIO_CHIAMATA_SCAMBIO

    IF Z > A
        THEN
            SCAMBIO_CHIAMATA_SCAMBIO (LISTA, A, Z, Z)
        ELSE
            scrivi LISTA
    END IF

END PERMUTA
```

Segue l'adattamento della pseudocodifica appena mostrata, in modo da simulare la ricorsione, utilizzando variabili globali:

Variabile	Descrizione
`SAVEZ`	È il vettore utilizzato per conservare il valore di '**`Z`**'.
`SAVEK`	È il vettore utilizzato per conservare il valore di '**`K`**'.
`SP`	È l'indice dei vettori usati per salvare i valori (*stack pointer*).

```
PERMUTA (LISTA, A, Z)

    SCAMBIO_CHIAMATA_SCAMBIO
        IF K >= A
            THEN
                LISTA[K] :==: LISTA[Z]
                SP := SP + 1
                SAVEZ(SP) := Z
                Z := Z - 1
                PERMUTA
                Z := SAVEZ(SP)
                SP := SP - 1
```

```
                LISTA[K] :==: LISTA[Z]
                SP := SP + 1
                SAVEK(SP) := K
                K := K - 1
                SCAMBIO_CHIAMATA_SCAMBIO
                K := SAVEK(SP)
                SP := SP - 1
        END IF
    END SCAMBIO_CHIAMATA_SCAMBIO

    IF Z > A
        THEN
            SP := SP + 1
            SAVEK(SP) := K
            K := N
            SCAMBIO_CHIAMATA_SCAMBIO
            K := SAVEK(SP)
            SP := SP - 1
        ELSE
            scrivi LISTA
    END IF
END PERMUTA
```

Listato 73.57. Soluzione in COBOL del problema delle permutazioni, con la simulazione della ricorsione. Una copia di questo file dovrebbe essere disponibile presso *allegati/cobol/HC07.cob* .

```
000600 IDENTIFICATION DIVISION.
000700 PROGRAM-ID.   HC07.
000800 AUTHOR.       DANIELE GIACOMINI.
000900 DATE-WRITTEN. 1985-06-19.
001000
001100
001200 ENVIRONMENT DIVISION.
001300
```

```
001400
001500 DATA DIVISION.
001600
001700
001800 WORKING-STORAGE SECTION.
001900
002000 01  RECORD-STACKS.
002100     02  SAVEZ   OCCURS 100 TIMES PIC 9.
002200     02  SAVEK   OCCURS 100 TIMES PIC 9.
002300
002400 01  STACK-POINTER.
002500     02  SP                       PIC 999.
002600
002700 01  VARIABILI-SCALARI.
002800     02  A                        PIC 9   VALUE 1.
002900     02  Z                        PIC 9.
003000     02  K                        PIC 9.
003100     02  TEMP                     PIC 9.
003200     02  J                        PIC 99.
003300
003400 01  RECORD-LISTA.
003500     02  LISTA   OCCURS 10 TIMES PIC 9.
003600
003700
003800 PROCEDURE DIVISION.
003900
004000 MAIN.
004100
004200     DISPLAY "INSERISCI IL NUMERO DI ELEMENTI ",
004250             "DA PERMUTARE".
004300     DISPLAY "(UNA CIFRA)".
004400     ACCEPT Z.
004500*
004600*    si genera la prima permutazione con numeri in
```

```
004650*    ordine crescente
004800*
004900     MOVE SPACES TO RECORD-LISTA.
005000     PERFORM GEN-PRIMA-PERMUTAZIONE
005050             VARYING J FROM 1 BY 1
005100             UNTIL J > Z.
005200
005300     PERFORM PERMUTA.
005400
005500     STOP RUN.
005600
005700
005800 GEN-PRIMA-PERMUTAZIONE.
005900
006000     MOVE J TO LISTA(J).
006100
006200
006300 PERMUTA.
006400
006500     IF Z > A
006600       THEN
006700*
006800*         push
006900*
007000           COMPUTE SP = SP + 1,
007100           COMPUTE SAVEK(SP) = K,
007200*
007300*          chiamata
007400*
007500           COMPUTE K = Z,
007600           PERFORM SCAMBIO-CHIAMATA-SCAMBIO,
007700*
007800*          pop
007900*
```

```
008000           COMPUTE K = SAVEK(SP),
008100           COMPUTE SP = SP - 1,
008200
008300       ELSE
008400
008500           DISPLAY RECORD-LISTA.
008600
008700
008800 SCAMBIO-CHIAMATA-SCAMBIO.
008900
009000     IF K >= A
009100       THEN
009200*
009300*      scambio di LISTA(K) con LISTA(Z)
009400*
009500       MOVE LISTA(K) TO TEMP,
009600       MOVE LISTA(Z) TO LISTA (K),
009700       MOVE TEMP     TO LISTA (Z),
009800*
009900*      push
010000*
010100       COMPUTE SP = SP + 1,
010200       COMPUTE SAVEZ(SP) = Z,
010300*
010400*      chiamata
010500*
010600       COMPUTE Z = Z - 1,
010700       PERFORM PERMUTA,
010800*
010900*      pop
011000*
011100       COMPUTE Z = SAVEZ(SP),
011200       COMPUTE SP = SP - 1,
011300*
```

```
011400*       scambio di LISTA(K) con LISTA(Z)
011500*
011600        MOVE LISTA(K) TO TEMP,
011700        MOVE LISTA(Z) TO LISTA (K),
011800        MOVE TEMP     TO LISTA (Z),
011900*
012000*       push
012100*
012200        COMPUTE SP = SP + 1,
012300        COMPUTE SAVEK(SP) = K,
012400*
012500*       chiamata
012600*
012700        COMPUTE K = K - 1,
012800        PERFORM SCAMBIO-CHIAMATA-SCAMBIO,
012900*
013000*       pop
013100*
013200        COMPUTE K = SAVEK(SP),
013300        COMPUTE SP = SP - 1.
013400
```

73.4.8 Bibliografia

«

- Wagener J. L., *FORTRAN 77 Principles of Programming*, Wiley, 1980, pagine 228..229. **[5]**
- Knuth D. E., *The Art of Computer Programming - Volume 3 Sorting and Searching*, Addison-Wesley, 1973, capitolo 5.
- Dijkstra E. W., *A Discipline of Programming*, Prentice-Hall, 1976, capitolo 13.

Il concetto di locale e di globale: ambito delle variabili

- Wirth N., *Principi di programmazione strutturata*, ISEDI, 1977, capitolo 12. **[1]**
- Moore L., *Foundations of Programming with Pascal*, Ellis Horwood Limited, 1980, capitolo 10. **[3]**
- Ledgard, Nagin, Hueras, *Pascal with Style*, Hayden, 1979, pagine 126..134. **[2]**
- Dijkstra E. W., *A Discipline of Programming*, Prentice-Hall, 1976, capitolo 10.
- Nicholls J. E., *The Structure and Design of Programming Languages*, Addison-Wesley, 1975, capitolo 12.

La ricorsione

- Arsac J., *La construction de programmes structures*, DUNOD, 1977, capitoli 2..5.
- Moore L., *Foundations of Programming with Pascal*, Ellis Horwood Limited, 1980, capitolo 14.
- Aho, Hopcroft, Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974, pagine 55..60. **[4]**
- Ledgard, Nagin, Hueras, *Pascal with Style*, Hayden, 1979, pagine 134..139.
- Wirth N., *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976, capitolo 3.
- Wagener J. L., *FORTRAN 77 Principles of Programming*, Wiley, 1980, capitolo 11. **[5]**

I linguaggi

- Burroughs Corporation, *Computer Management System COBOL*, codice 2007266.

• Burroughs Corporation, *Computer Management System Message Processing Language (MPLII) - reference manual*, codice 2007563.

Figura 73.58. Ultima foto del 1988 di un elaboratore Burroughs B91, prima della sua dismissione completa. Alla destra appaiono le unità a disco; in fondo il B91, preso dal lato posteriore, assieme a un terminale MT. Il materiale infiammabile a cui si riferisce la scritta sull'armadio era una bottiglietta di alcool, utilizzato come solvente per pulire manualmente i dischi (sia le unità rimovibili, sia i piatti del disco fisso) a seguito dei continui atterraggi delle testine. I piatti dei dischi venivano sfruttati fino a quando la traccia iniziale non risultava raschiata completamente, arrivando a volte anche a rimontarli fuori asse, allo scopo di utilizzare una superficie ancora efficiente per tale traccia. Le testine delle unità a disco dovevano compiere un tragitto molto lungo per raggiungere tutte le tracce del disco (con tutti i problemi che ne derivano a causa della dilatazione termica) e spesso il loro motorino si incagliava: per fare riprendere l'attività all'elaboratore occorreva colpire le unità sullo stesso asse delle testine, per sbloccare il loro movimento.

73.5 Riferimenti

«

- *TinyCOBOL*, *http://tiny-cobol.sourceforge.net*
- *OpenCOBOL*, *http://www.opencobol.org*

[1] **TinyCOBOL** GNU GPL e GNU LGPL

[2] **OpenCOBOL** GNU GPL e GNU LGPL

Parte v

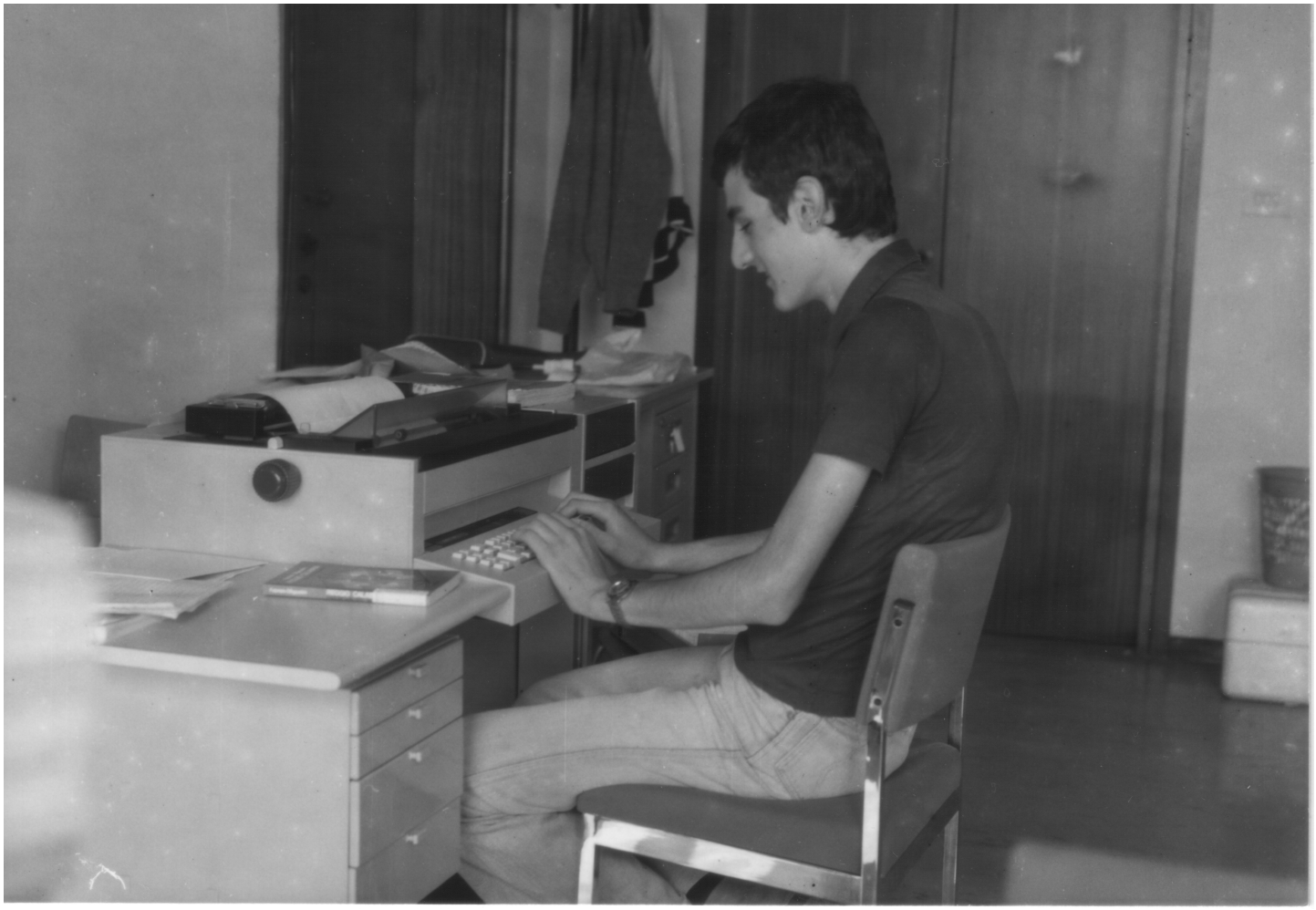
Basi di dati

74 DBMS e SQL 1863
74.1 Introduzione ai DBMS 1864
74.2 Questioni organizzative generali dei DBMS comuni 1885
74.3 Introduzione a SQL 1890
74.4 DDL 1893
74.5 DML 1915
74.6 DCL 1938
74.7 Riferimenti 1946
75 PostgreSQL 1949
75.1 Struttura e preparazione 1950
75.2 Gestione del DBMS 1972
75.3 Il linguaggio 2000
75.4 Accesso attraverso PgAccess 2027
75.5 Accesso attraverso WWW-SQL 2041
75.6 Riferimenti 2065
76 MySQL 2067
76.1 Struttura e preparazione 2067
76.2 Gestione del DBMS 2090
76.3 Riferimenti 2102
77 SQLite 2103

77.1 Utilizzo generale 2104
77.2 Esempi comuni 2109
77.3 Riferimenti 2119
78 ODBC 2121
78.1 DSN 2121
78.2 unixODBC 2121
78.3 ODBCConfig 2124
78.4 Accesso a ODBC tramite «isql» o «iusql» 2129
78.5 Riferimenti 2130
79 SQL: lezioni pratiche e verifiche 2131
79.1 Creazione ed eliminazione delle relazioni 2140
79.2 Interrogazione semplice di una relazione 2155
79.3 Interrogazione ordinata di una relazione 2164
79.4 Interrogazione selettiva di una relazione 2170
79.5 Interrogazioni simultanee di più relazioni 2179
79.6 Interrogazioni simultanee di più relazioni e alias . . . 2187
79.7 Viste 2192
79.8 Modifica del contenuto delle tuple 2202
79.9 Eliminazione delle tuple 2211
79.10 Grilletti per il controllo del dominio degli attributi 2217
79.11 Grilletti per il controllo della validità esterna 2227
79.12 Selezione di attributi virtuali, ottenuti da un'espressione 2240
79.13 Aggregazioni 2250

79.14 Inserimento automatico del costo medio 2262

Figura v.1. Uno studente all'opera con una macchina TES-501 Olivetti, nel 1979, per la catalogazione dei libri della biblioteca scolastica.



1862

Capitolo 74

DBMS e SQL

74.1 Introduzione ai DBMS 1864
74.1.1 Caratteristiche fondamentali 1865
74.1.2 Modello relazionale 1867
74.1.3 Gestione delle relazioni 1874
74.2 Questioni organizzative generali dei DBMS comuni . . 1885
74.2.1 Configurazione e basi di dati amministrative 1885
74.2.2 Copie di sicurezza delle basi di dati 1886
74.2.3 Comunicazione con il DBMS e accesso alle basi di dati 1887
74.2.4 Utenze e privilegi 1888
74.2.5 ODBC 1890
74.3 Introduzione a SQL 1890
74.3.1 Concetti fondamentali 1891
74.3.2 Terminologia 1892
74.4 DDL 1893
74.4.1 Tipi di dati 1893
74.4.2 Operatori, funzioni ed espressioni 1902
74.4.3 Le relazioni dal punto di vista di SQL 1907
74.5 DML 1915
74.5.1 Inserimento, eliminazione e modifica dei dati 1915
74.5.2 Interrogazioni di relazioni 1919

74.5.3 Trasferimento di dati in un'altra relazione 1932
74.5.4 Viste 1934
74.5.5 Cursori 1935
74.6 DCL 1938
74.6.1 Gestione delle utenze 1938
74.6.2 Gestione delle basi di dati 1941
74.6.3 Gestione dei privilegi standard 1942
74.6.4 Controllo delle transazioni 1944
74.7 Riferimenti 1946

`ALTER TABLE` 1914 `ALTER USER` 1938 `CLOSE` 1938 `COMMIT` 1944 `CREATE DATABASE` 1941 `CREATE TABLE` 1908 `CREATE USER` 1938 `CREATE VIEW` 1934 `DECLARE` 1935 `DELETE FROM` 1919 `DROP TABLE` 1915 `DROP USER` 1938 `DROP VIEW` 1934 `FETCH` 1937 `GRANT` 1942 `INSERT INTO` 1916 1933 `OPEN` 1935 `REVOKE` 1942 `ROLLBACK` 1944 `SELECT` 1919 `UPDATE` 1918

74.1 Introduzione ai DBMS

«

Un DBMS (*Data base management system*) è, letteralmente, un sistema di gestione di ***basi di dati***, che per attuare questa gestione utilizza il software. Queste «basi di dati» sono dei contenitori atti a immagazzinare una grande quantità di dati, per i quali il «sistema di gestione» è necessario a permetterne la fruizione (diverso è il problema della semplice archiviazione dei dati).

74.1.1 Caratteristiche fondamentali

Un DBMS, per essere considerato tale, deve avere caratteristiche determinate. Le più importanti che permettono di comprenderne il significato sono elencate di seguito.

- Un DBMS è fatto per gestire grandi quantità di dati.

  Convenzionalmente si può intendere che un gruppo di informazioni sia di grandi dimensioni quando questo non possa essere contenuto tutto simultaneamente nella memoria centrale dell'elaboratore. In generale un DBMS non dovrebbe porre limiti alle dimensioni, tranne quelle imposte dai supporti fisici in cui devono essere memorizzate le informazioni.

- I dati devono poter essere condivisibili.

  L'idea che sta alla base dei sistemi di gestione dei dati è quella di accentrare le informazioni in un sistema di amministrazione unico. In tal senso è poi necessario che questi dati siano condivisibili da diverse applicazioni e da diversi utenti.

- I dati devono essere persistenti e affidabili.

  I dati sono persistenti quando continuano a esistere dopo lo spegnimento della macchina con cui vengono elaborati; sono affidabili quando gli eventi per cui si possono produrre alterazioni accidentali sono estremamente limitati.

- L'accesso ai dati deve essere controllabile.

  Dovendo trattare una grande mole di dati in modo condiviso, è indispensabile che esistano dei sistemi di controllo degli accessi, per evitare che determinate informazioni possano essere ottenute

da chi non è autorizzato, oppure che vengano modificate da chi non ne è il responsabile.

74.1.1.1 Livelli di astrazione dei dati

I dati gestiti da un DBMS devono essere organizzati a diversi livelli di astrazione. Generalmente si distinguono tre livelli: esterno, logico e interno.

- Il livello interno è quello usato effettivamente per la memorizzazione dei dati. In pratica è rappresentato dai file che contengono effettivamente le informazioni e dal modo con cui questi file vengono utilizzati. Il livello interno non è importante per la progettazione di una base di dati e nemmeno per la scrittura di programmi che devono interagire con il DBMS.
- Il livello logico, o concettuale, è quello che descrive i dati secondo la filosofia del DBMS particolare con cui si ha a che fare.
- Lo schema esterno è un'astrazione aggiuntiva che permette di definire dei punti di vista differenti dei dati descritti a livello logico. L'accesso ai dati avviene solo a questo livello, anche se di fatto può coincidere con il livello logico.

74.1.1.2 Ruoli e sigle standard

Lo studio sui DBMS ha generato degli acronimi che rappresentano persone o componenti essenziali di un sistema del genere. Questi acronimi devono essere conosciuti perché se ne fa uso abitualmente nella documentazione riferita ai DBMS.

L'organizzazione di una base di dati è compito del suo amministratore, definito DBA (*Data base administrator*). Eventualmente può trattarsi anche di più persone; in ogni caso, chi ha la responsabilità dell'amministrazione di uno o più basi di dati è un DBA.

Il concetto di DBA non è molto preciso, perché include assieme ruoli che possono essere differenti. Si va dall'amministratore dell'intero DBMS, fino a coloro che hanno la responsabilità di una base di dati singola.

La definizione della struttura dei dati (sia a livello logico, sia a livello esterno) viene fatta attraverso un linguaggio di programmazione definito DDL (*Data definition language*), la gestione dei dati avviene attraverso un altro linguaggio, detto DML (*Data manipulation language*), infine, la gestione della sicurezza viene fatta attraverso un linguaggio DCL (*Data control language*). Nella pratica, DDL, DML e DCL possono coincidere, come nel caso del linguaggio SQL.

74.1.2 Modello relazionale

Una base di dati può essere impostata secondo diversi tipi di modelli (logici) di rappresentazione. Quello più comune, che è anche il più semplice dal punto di vista umano, è il modello relazionale. In tal senso, un DBMS relazionale viene anche definito semplicemente come RDBMS.

Nel modello relazionale, i dati sono raccolti all'interno di ***relazioni***. Ogni relazione è una raccolta di nessuna o più ***tuple*** di tipo omogeneo. La tupla rappresenta una singola informazione completa, in rapporto alla relazione a cui appartiene, suddivisa in ***attributi***. Le in-

formazioni che possono essere inserite in ogni singolo attributo, dipendono da un ***dominio***. Una relazione, nella sua definizione, non ha una «forma» particolare, tuttavia questo concetto si presta a una rappresentazione tabellare: gli attributi sono rappresentati dalle colonne e le tuple dalle righe. Si osservi l'esempio della figura 74.1.

Figura 74.1. Relazione '**`Indirizzi (Cognome, Nome, Indirizzo, Telefono)`**'.

Indirizzi			
Cognome	**Nome**	**Indirizzo**	**Telefono**
Pallino	Pinco	Via Biglie 1	0222,222222
Tizi	Tizio	Via Tazi 5	0555,555555
Cai	Caio	Via Caini 1	0888,888888
Semproni	Sempronio	Via Sempi 7	0999,999999

In una relazione, le tuple non hanno una posizione particolare, sono semplicemente contenute nell'insieme della relazione stessa. Se l'ordine ha una rilevanza per le informazioni contenute, questo elemento dovrebbe essere aggiunto tra gli attributi, senza essere determinato da un'ipotetica collocazione fisica. Osservando l'esempio, si intende che l'ordine delle righe non ha importanza per le informazioni che si vogliono trarre; al massimo, un elenco ordinato può facilitare la lettura umana, quando si è alla ricerca di un nome particolare, ma ciò non ha rilevanza nella struttura che deve avere la relazione corrispondente.

Il fatto che la posizione delle tuple all'interno della relazione non

sia importante, significa che non è necessario poterle identificare: le tuple si distinguono in base al loro contenuto. In questo senso, una relazione non può contenere due tuple uguali: la presenza di doppioni non avrebbe alcun significato.

A differenza delle tuple, gli attributi devono essere identificati attraverso un nome. Infatti, il semplice contenuto delle tuple non è sufficiente a stabilire di quale attributo si tratti. Osservando la prima riga dell'esempio, diventa difficile distinguere quale sia il nome e quale il cognome:

Pallino	Pinco	Via Biglie 1	0222,222222

Assegnando agli attributi (cioè alle colonne) un nome, diventa indifferente la disposizione fisica di questi all'interno delle tuple.

74.1.2.1 Relazioni collegate

Generalmente, una relazione da sola non è sufficiente a rappresentare tutti i dati riferiti a un problema o a un interesse della vita reale. Quando una relazione contiene tante volte le stesse informazioni, è opportuno scinderla in due o più relazioni più piccole, collegate in qualche modo attraverso dei riferimenti. Si osservi il caso delle relazioni rappresentate dalle tabelle che si vedono nella figura 74.3.

Figura 74.3. Relazioni di un'ipotetica gestione del magazzino.

Articoli

Codice	Descrizione	Fornitore1	Fornitore2
vite30	Vite 3 mm	123	126
vite40	Vite 4 mm	126	127
dado30	Dado 3 mm	122	123
dado40	Dado 4 mm	126	127
rond50	Rondella 5 mm	123	126

Movimenti

Codice	Data	Carico	Scarico	CodFor	CodCli
vite40	01/01/2012	1200		124	
vite30	01/01/2012		800		825
vite30	02/01/2012	1000		954	
vite30	03/01/2012	2000		127	
rond50	03/01/2012		500		954

Fornitori

CodFor	Ditta	Indirizzo	Telefono
127	Vitoni spa	Via Ferri 2	0123,45678
122	Ferroni spa	Via Metalli 34	0234,5678
126	Nuova Metal	Via Industrie	0345,6789
123	Viti e Bulloni	Via di sopra 7	0567,9875

Clienti

CodCli	Ditta	Indirizzo	Telefono
925	Tendoni Max	Via di sotto 2	0113,44578
825	Arti Plus	Via di lato 45	0765,23456

La prima relazione, '**`Articoli`**', rappresenta l'anagrafica del magazzino di un grossista di ferramenta. Ogni articolo di magazzino viene codificato e descritto, inoltre vengono annotati i riferimenti ai codici di possibili fornitori. La seconda relazione, '**`Movimenti`**', elenca le operazioni di carico e di scarico degli articoli di magazzino, specificando solo il codice dell'articolo, la data, la quantità caricata o scaricata e il codice del fornitore o del cliente da cui è stato acquistato o a cui è stato venduto l'articolo. Infine seguono le relazioni che descrivono i codici dei fornitori e quelli dei clienti.

Si può intendere che una sola relazione non avrebbe potuto essere utilizzata utilmente per esprimere tutte queste informazioni.

È importante stabilire che, nel modello relazionale, il collegamento tra le tuple delle varie relazioni avviene attraverso dei valori e non

attraverso dei puntatori. Infatti, nella relazione '**Articoli**' l'attributo '**Fornitore1**' contiene il valore 123 e questo significa solo che i dati di quel fornitore sono rappresentati da quel valore. Nella relazione '**Fornitori**', la tupla il cui attributo '**CodFor**' contiene il valore 123 è quella che contiene i dati di quel particolare fornitore. Quindi, «123» non rappresenta un puntatore, ma solo una tupla che contiene quel valore nell'attributo «giusto». In questo senso si ribadisce l'indifferenza della posizione delle tuple all'interno delle relazioni.

74.1.2.2 Tipi di dati, domini e informazioni mancanti

Nelle relazioni, ogni attributo contiene una singola informazione elementare di un certo tipo, per il quale esiste un dominio determinato di valori possibili. Ogni attributo di ogni tupla deve contenere un valore ammissibile, nell'ambito del proprio dominio.

Spesso capitano situazioni in cui i valori di uno o più attributi di una tupla non sono disponibili per qualche motivo. In tal caso si pone il problema di assegnare a questi attributi un valore che definisca in modo non ambiguo questo stato di indeterminatezza. Questo valore viene definito come '**NULL**' ed è ammissibile per tutti i tipi di attributi possibili.

74.1.2.3 Vincoli di validità

I dati contenuti in una o più relazioni sono utili in quanto «sensati» in base al contesto a cui si riferiscono. Per esempio, considerando la relazione '**Movimenti**', vista precedentemente, questa deve contenere sempre un codice valido nell'attributo '**Codice**'. Se così non fosse, la registrazione data da quella tupla che dovesse avere un riferimen-

to a un codice di articolo non valido, non avrebbe alcun senso, perché mancherebbe proprio l'informazione più importante: l'articolo caricato o scaricato.

Il controllo sulla validità dei dati può avvenire a diversi livelli, a seconda della circostanza. Si possono distinguere vincoli che riguardano:

1. il dominio dell'attributo stesso -- quando si tratta di definire se l'attributo può assumere il valore '`NULL`' o meno e quando si stabilisce l'intervallo dei valori ammissibili;
2. gli altri attributi della stessa tupla -- quando dal valore contenuto in altri attributi della stessa tupla dipende l'intervallo dei valori ammissibili per l'attributo in questione;
3. gli attributi di altre tuple -- quando dal valore contenuto negli attributi delle altre tuple della stessa relazione dipende l'intervallo dei valori ammissibili per l'attributo in questione;
4. gli attributi di tuple di altre relazioni -- quando altre relazioni condizionano la validità di un attributo determinato.

I vincoli di tupla, ovvero quelli che riguardano i primi due punti dell'elenco appena indicato, sono i più semplici da esprimere perché non occorre conoscere altre informazioni esterne alla tupla stessa. Per esempio, un attributo che esprime un prezzo potrebbe essere definito in modo tale che non sia ammissibile un valore negativo; nello stesso modo, un attributo che esprime uno sconto su un prezzo potrebbe ammettere un valore positivo diverso da zero solo se il prezzo a cui si riferisce, contenuto nella stessa tupla, supera un valore determinato.

Il caso più importante di un vincolo interno alla relazione, che coinvolge più tuple, è quello che riguarda le ***chiavi***. In certe situazioni, un attributo, o un gruppo particolare di attributi di una relazione, deve essere unico per ogni tupla. Quindi, questo attributo, o questo gruppo di attributi, è valido solo quando non è già presente in un'altra tupla della stessa relazione.

Quando le informazioni sono distribuite fra più relazioni, i dati sono validi solo quando tutti i riferimenti sono validi. Volendo riprendere l'esempio della gestione di magazzino, visto precedentemente, una tupla della relazione '**`Movimenti`**' che dovesse contenere un codice di un fornitore o di un cliente inesistente, si troverebbe, in pratica, senza questa informazione.

74.1.2.4 Chiavi

Nella sezione precedente si è accennato alle ***chiavi***. Questo concetto merita un po' di attenzione. In precedenza è stato affermato che una relazione contiene una raccolta di tuple che contano per il loro contenuto e non per la loro posizione. In questo senso non è ammissibile una relazione contenente due tuple identiche. Una ***chiave*** di una relazione è un gruppo di attributi che permette di identificare univocamente le tuple in essa contenute; per questo, tali attributi devono contenere dati differenti per ogni tupla.

Stabilendo quali attributi devono costituire una chiave per una certa relazione, si comprende intuitivamente che questi attributi non possono mai contenere un valore indeterminato.

Nella definizione di relazioni collegate attraverso dei riferimenti, l'oggetto di questi riferimenti deve essere una chiave per la relazio-

ne di destinazione. Diversamente non si otterrebbe un riferimento univoco a una tupla particolare.

74.1.3 Gestione delle relazioni

«

Prima di affrontare l'utilizzo pratico di una base di dati relazionale, attraverso un linguaggio di manipolazione dei dati, è opportuno considerare a livello teorico alcuni tipi di operazioni che si possono eseguire con le relazioni.

Inizialmente è stato affermato che una relazione è un insieme di tuple... Dalla teoria degli insiemi derivano molte delle operazioni che riguardano le relazioni.

74.1.3.1 Unione, intersezione e differenza

«

Quando si maneggiano relazioni contenenti gli stessi attributi, hanno senso le operazioni fondamentali sugli insiemi: unione, intersezione e differenza. Il significato è evidente: l'unione genera una relazione composta da tutte le tuple distinte delle relazioni di origine; l'intersezione genera una relazione composta dalle tuple presenti simultaneamente in tutte le relazioni di origine; la differenza genera una relazione contenente le tuple che compaiono esclusivamente nella prima delle relazioni di origine.

L'esempio rappresentato dalle relazioni della figura 74.4 dovrebbe chiarire il senso di queste affermazioni.

Figura 74.4. Unione, intersezione e differenza tra relazioni.

Laureati

Codice	Nominativo	...
1245	Tizi Tizio	...
1745	Cai Caio	...
1655	Semproni Sempronio	...

Magazzinieri

Codice	Nominativo	...
1745	Cai Caio	...
1986	Pallino Pinco	...
1245	Tizi Tizio	...

Laureati UNITO Magazzinieri

Codice	Nominativo	...
1245	Tizi Tizio	...
1745	Cai Caio	...
1655	Semproni Sempronio	...
1986	Pallino Pinco	...

Laureati INTERSECATO Magazzinieri

Codice	Nominativo	...
1245	Tizi Tizio	...
1745	Cai Caio	...

Laureati MENO Magazzinieri

Codice	Nominativo	...
1655	Semproni Sempronio	...

74.1.3.2 Ridenominazione degli attributi

«

L'elaborazione dei dati contenuti in una relazione può avvenire previa modifica dei nomi di alcuni attributi. La modifica dei nomi genera di fatto una nuova relazione temporanea, per il tempo necessario a eseguire l'elaborazione conclusiva.

Le situazioni in cui la ridenominazione degli attributi può essere conveniente possono essere varie. Nel caso delle operazioni sugli insiemi visti nella sezione precedente, la ridenominazione può rendere compatibili relazioni i cui attributi, pur essendo compatibili, hanno nomi differenti.

74.1.3.3 Selezione, proiezione e congiunzione

«

La ***selezione*** e la ***proiezione*** sono operazioni che si eseguono su una sola relazione e generano una relazione che contiene una porzione dei dati di quella di origine. La selezione permette di estrarre alcune tuple dalla relazione, mentre la proiezione estrae parte degli attributi di tutte le tuple. Il primo caso, quello della selezione, non richiede considerazioni particolari, mentre la proiezione ha delle implicazioni importanti.

Attraverso la proiezione, utilizzando solo parte degli attributi, si genera una relazione in cui si potrebbero perdere delle tuple, a causa della possibilità che risultino dei doppioni. Per esempio, si consideri la relazione mostrata nella figura 74.5.

Figura 74.5. Relazione '**`Utenti (UID, Nominativo, Cognome, Nome, Ufficio)`**'.

Utenti				
UID	**Nominativo**	**Cognome**	**Nome**	**Ufficio**
0	root	Pallino	Pinco	CED
515	rmario	Rossi	Mario	Contabilità
501	bbianco	Bianchi	Bianco	Magazzino
502	rrosso	Rossi	Rosso	Contabilità

La figura mostra una relazione contenente le informazioni sugli utenti di un centro di elaborazione dati. Si può osservare che sia l'attributo '**`UID`**', sia l'attributo '**`Nominativo`**', possono essere da soli una chiave per la relazione. Se da questa relazione si vuole ottenere una proiezione contenente solo gli attributi '**`Cognome`**' e '**`Ufficio`**', non essendo questi due una chiave della relazione, si perdono delle tuple.

Figura 74.6. Proiezione degli attributi '**`Cognome`**' e '**`Ufficio`**' della relazione '**`Utenti`**'.

Cognome	**Ufficio**
Pallino	CED
Rossi	Contabilità
Bianchi	Magazzino

La figura 74.6 mostra la proiezione della relazione '**`Utenti`**', in cui

sono stati estratti solo gli attributi '**`Cognome`**' e '**`Ufficio`**'. In tal modo, le tuple che prima corrispondevano al numero '**`UID`**' 515 e 502 si sono ridotte a una sola, quella contenente il cognome «Rossi» e l'ufficio «Contabilità».

Da questo esempio si dovrebbe intendere che la proiezione ha senso, prevalentemente, quando gli attributi estratti costituiscono una chiave della relazione originaria

La ***congiunzione*** di relazioni, o *join*, è un'operazione in cui due o più relazioni vengono unite a formare una nuova relazione. Questo congiungimento implica la creazione di tuple formate dall'unione di tuple provenienti dalle relazioni di origine. Se per semplicità si pensa solo alla congiunzione di due relazioni: si va da una congiunzione minima in cui nessuna tupla della prima relazione risulta abbinata ad altre tuple della seconda, fino a un massimo in cui si ottengono tutti gli abbinamenti possibili delle tuple della prima relazione con quelle della seconda. Tra questi estremi si pone la situazione tipica, quella in cui ogni tupla della prima relazione viene collegata solo a una tupla corrispondente della seconda.

La ***congiunzione naturale*** si ottiene quando le relazioni oggetto di tale operazione vengono collegate in base ad attributi aventi lo stesso nome. Per osservare di cosa si tratta, vale la pena di riprendere l'esempio della gestione di magazzino già descritto in precedenza. Nella figura 74.7 ne viene mostrata nuovamente solo una parte.

Figura 74.7. Le relazioni '**`Articoli`**' e '**`Movimenti`**' dell'esempio sulla gestione del magazzino.

Articoli		
Codice	**Descrizione**	**...**
vite30	Vite 3 mm	...
vite40	Vite 4 mm	...
dado30	Dado 3 mm	...
dado40	Dado 4 mm	...
rond50	Rondella 5 mm	...

Movimenti				
Codice	**Data**	**Carico**	**Scarico**	**...**
vite40	01/01/2012	1200		...
vite30	01/01/2012		800	...
vite30	02/01/2012	1000		...
vite30	03/01/2012	2000		...
rond50	03/01/2012		500	...

La congiunzione naturale delle relazioni '**`Movimenti`**' e '**`Articoli`**', basata sulla coincidenza del contenuto dell'attributo '**`Codice`**', genera una relazione in cui appaiono tutti gli attributi delle due relazioni di origine, con l'eccezione dell'attributo '**`Codice`**' che appare una volta sola (figura 74.8).

Tabella 74.8. Il join naturale tra le relazioni '**Movimenti**' e '**Articoli**'.

Codice	**Data**	**Carico**	**Scarico**	**...**	**Descrizione**	**...**
vite40	01/01/2012	1200		...	Vite 4 mm	...
vite30	01/01/2012		800	...	Vite 3 mm	...
vite30	02/01/2012	1000		...	Vite 3 mm	...
vite30	03/01/2012	2000		...	Vite 3 mm	...
rond50	03/01/2012		500	...	Rondella 5 mm	...

Nel caso migliore, ogni tupla di una relazione trova una tupla corrispondente dell'altra; nel caso peggiore, nessuna tupla ha una corrispondente nell'altra relazione. L'esempio mostra che tutte le tuple della relazione '**Movimenti**' hanno trovato una corrispondenza nella relazione '**Articoli**', mentre solo alcune tuple della relazione '**Articoli**' hanno una corrispondenza dall'altra parte. Queste tuple, quelle che non hanno una corrispondenza, sono dette «penzolanti», o *dangling*, e di fatto vengono perdute dopo la congiunzione.

Quando una congiunzione genera corrispondenze per tutte le tuple delle relazioni coinvolte, si parla di congiunzione completa. La dimensione (espressa in quantità di tuple) della relazione risultante in presenza di una congiunzione completa è pari alla dimensione massima delle varie relazioni.

Quando si vuole eseguire la congiunzione di relazioni che non hanno attributi in comune si ottiene il collegamento di ogni tupla di una relazione con ogni tupla delle altre. Si può osservare l'esempio della

figura 74.9 che riprende il solito problema del magazzino, con delle semplificazioni opportune.

Figura 74.9. Le relazioni '**`Articoli`**' e '**`Fornitori`**' dell'esempio sulla gestione del magazzino.

Articoli			
Codice	**Descrizione**	**Fornitore1**	**...**
vite30	Vite 3 mm	127	...
vite40	Vite 4 mm	127	...
dado30	Dado 3 mm	122	...

Fornitori		
CodFor	**Ditta**	**...**
127	Vitoni spa	...
122	Ferroni spa	...

Nessuno degli attributi delle due relazioni coincide, quindi si ottiene un «prodotto» tra le due relazioni, in pratica, una relazione che contiene il prodotto delle tuple contenute nelle relazioni originali (figura 74.10).

Figura 74.10. Il prodotto tra le relazioni '**Articoli**' e '**Fornitori**'.

Codice	Descrizione	Fornitore1	...	CodFor	Ditta	...
vite30	Vite 3 mm	127	...	127	Vitoni spa	...
vite40	Vite 4 mm	127	...	127	Vitoni spa	...
dado30	Dado 3 mm	122	...	127	Vitoni spa	...
vite30	Vite 3 mm	127	...	122	Ferroni spa	...
vite40	Vite 4 mm	127	...	122	Ferroni spa	...
dado30	Dado 3 mm	122	...	122	Ferroni spa	...

Quando si esegue un'operazione del genere, è normale che molte delle tuple risultanti siano prive di significato per gli scopi che ci si prefigge. Di conseguenza, quasi sempre, si applica poi una selezione attraverso delle condizioni. Nel caso dell'esempio, sarebbe ragionevole porre come condizione di selezione l'uguaglianza tra i valori dell'attributo '**Fornitore1**' e '**CodFor**'.

Figura 74.11. La selezione delle tuple che rispettano la condizione di uguaglianza tra gli attributi '**Fornitore1**' e '**CodFor**'.

Codice	Descrizione	Fornitore1	...	CodFor	Ditta	...
vite30	Vite 3 mm	127	...	127	Vitoni spa	...
vite40	Vite 4 mm	127	...	127	Vitoni spa	...
dado30	Dado 3 mm	122	...	122	Ferroni spa	...

Generalmente, nella pratica, non esiste la possibilità di definire una congiunzione basato sull'uguaglianza dei nomi degli attributi. Di solito si esegue una congiunzione che genera un prodotto tra le relazioni, quindi si applicano delle condizioni di selezione come nell'esempio mostrato. Quando la selezione in questione è del tipo visto nell'esempio, cioè basata sull'uguaglianza del contenuto di attributi delle diverse relazioni (anche se il nome di questi attributi è differente), si parla di ***equi-giunzione*** (*equi-join*).

74.1.3.4 Gestione dei valori nulli

Si è accennato in precedenza alla possibilità che gli attributi di una relazione possano contenere anche il valore indeterminato, o '**`NULL`**'. Con questo valore indeterminato non si possono fare comparazioni con valori determinati e di solito nemmeno con altri valori indeterminati. Per esempio, non si può dire che '**`NULL`**' sia maggiore o minore di qualcosa; una comparazione di questo tipo genera solo un risultato indeterminato. '**`NULL`**' è solo uguale a se stesso ed è diverso da ogni altro valore, compreso un altro valore '**`NULL`**'.

Per verificare la presenza o l'assenza di un valore indeterminato si utilizzano generalmente operatori specifici, come in SQL:

- '**`IS NULL`**' -- che si avvera quando il valore controllato è indeterminato;
- '**`IS NOT NULL`**' -- che si avvera quando il valore controllato è determinato, quindi diverso da indeterminato.

Nel momento in cui si eseguono delle espressioni logiche, utilizzando i soliti operatori AND, OR e NOT, si pone il problema di stabilire cosa accade quando si presentano valori indeterminati. La soluzione

è intuitiva: quando non si può fare a meno di conoscere il valore che si presenta come indeterminato, il risultato è indeterminato. Questo concetto deriva dalla cosiddetta logica *fuzzy*.

Figura 74.12. Tabella della verità degli operatori AND e OR quando sono coinvolti valori indefiniti.

```
? AND ? = ?      ? OR ? = ?
? AND F = F      ? OR F = ?
? AND T = ?      ? OR T = T
F AND ? = F      F OR ? = ?
F AND F = F      F OR F = F
F AND T = F      F OR T = T
T AND ? = ?      T OR ? = T
T AND F = F      T OR F = T
T AND T = T      T OR T = T
```

74.1.3.5 Relazioni derivate e viste

«

Precedentemente si è accennato al fatto che la rappresentazione finale dei dati può essere diversa da quella logica. Nel modello relazionale è possibile ottenere delle relazioni derivate da altre, attraverso una funzione determinata che stabilisce il modo con cui ottenere queste derivazioni. Si distingue fondamentalmente tra:

- relazioni derivate virtuali, o ***viste***, che non generano nuove relazioni memorizzate nella base di dati, il cui contenuto viene generato al volo al momento della necessità;
- relazioni derivate materializzate, che generano una nuova relazione nella base di dati.

Il primo dei due casi è semplice da gestire, perché i dati sono sempre allineati correttamente, ma è pesante dal punto di vista elaborativo;

il secondo ha invece i pregi e i difetti opposti. Con il termine «vista» si intende fare riferimento alle relazioni derivate virtuali.

74.2 Questioni organizzative generali dei DBMS comuni

Dopo la teoria astratta, l'organizzazione di un DBMS richiede una realizzazione pratica, che implica delle scelte. In questa sezione si descrivono alcune questioni «pratiche» che è bene conoscere inizialmente, prima di affrontare lo studio di un DBMS specifico.

74.2.1 Configurazione e basi di dati amministrative

Un DBMS, per funzionare, deve poter annotare l'esistenza e i contenuti delle proprie basi di dati, così come l'esistenza e i privilegi dei propri utenti. Per conservare queste informazioni, può gestire dei file di configurazione, oppure, più spesso, impiegare una o più basi di dati amministrative, la cui gestione avviene in modo quasi trasparente.

La presenza di queste basi di dati amministrative implica il fatto che non possano esserne create delle altre con gli stessi nomi, ma ci sono naturalmente anche altre implicazioni più importanti.

Quando si installa il software di gestione del DBMS per la prima volta, è necessario provvedere a costruire le basi di dati amministrative, oltre che, eventualmente, a sistemare altri file di configurazione. Per questo, di solito il software del DBMS include un programma che predispone tali basi di dati speciali con una configurazione iniziale predefinita.

Quando si aggiorna il software di gestione del DBMS, si ha generalmente la necessità di conservare i dati preesistenti. Questo signi-

fica preservare le basi di dati che sono state create e le informazioni sulle utenze, con i privilegi rispettivi. Il problema sta nel fatto che il software aggiornato potrebbe avere un'organizzazione differente nel modo di gestire i file che contengono le informazioni sulle basi di dati, pertanto è poi necessario provvedere a una conversione, con l'ausilio di strumenti realizzati appositamente per quel DBMS. Naturalmente, per evitare circoli viziosi, un software aggiornato dovrebbe essere in grado di accedere a basi di dati di qualche versione precedente.

Generalmente, prima di un aggiornamento del software di gestione del DBMS, si consiglia di eseguire una copia dei dati in una forma indipendente dalla versione, che può essere acquisita successivamente, dopo l'aggiornamento; tuttavia, rimane il problema delle basi di dati amministrative: se dovesse fallire la procedura di aggiornamento automatica, si renderebbe necessaria, nuovamente, la creazione delle basi di dati (vuote) e delle utenze.

74.2.2 Copie di sicurezza delle basi di dati

«

Generalmente, le copie di sicurezza del contenuto di una basi di dati, si fanno in modo di generare del codice SQL, contenente le istruzioni per la creazione delle relazioni e per l'inserimento delle tuple. Questo viene ottenuto tramite programmi o script del software del DBMS e il codice che si ottiene è specifico di quel tipo di DBMS, perciò non è universale.

Ci possono essere dei DBMS che consentono l'acquisizione di dati molto complessi in un solo attributo (dove per «attributo» si intende una cella di una riga di una tabella), ma poi, questi dati non possono essere rappresentati in modo testuale in un codice SQL. In tali casi,

l'archiviazione in forma di codice SQL si deve limitare ai dati consueti, ignorando il resto; pertanto si devono usare formati differenti per l'archiviazione completa dei dati.

74.2.3 Comunicazione con il DBMS e accesso alle basi di dati

I programmi accedono alle basi di dati attraverso un protocollo di comunicazione con il DBMS. Il protocollo in questione dipende dal DBMS, ma generalmente consente di trasportare delle istruzioni SQL.

Nei sistemi Unix, la comunicazione con il DBMS avviene tipicamente attraverso socket di dominio Unix, per le comunicazioni locali, e socket di dominio Internet per quelle remote. Pertanto, il DBMS deve disporre di un demone in attesa di dati da un file di tipo socket e in ascolto di una porta TCP o UDP (di solito si tratta del protocollo TCP).

Naturalmente, per consentire l'accesso alle basi di dati, il DBMS deve avere un modo per «riconoscere» chi vuole accedere.

Un DBMS che consente esclusivamente collegamenti di tipo locale, avrebbe la possibilità di individuare gli accessi in base al numero UID associato al processo elaborativo del programma che tenta il contatto. In questo caso particolare, il riconoscimento delle utenze può essere demandata al sistema operativo.

74.2.4 Utenze e privilegi

Le utenze di un DBMS servono a distinguere le competenze al suo interno. Generalmente si distingue la presenza di un amministratore con poteri illimitati nell'ambito della gestione complessiva del DBMS e di conseguenza di ogni base di dati. Il nome, dal punto del DBMS, di questo amministratore, non è standardizzato. A titolo di esempio, nel caso di PostgreSQL si tratta normalmente dell'utente '**`postgres`**', mentre con MySQL si usa il nome '**`root`**'.

Generalmente, il DBMS riconosce gli utenti attraverso una parola d'ordine, che deve essere fornita all'inizio di ogni collegamento; tuttavia, dovrebbe esistere anche la possibilità di definire utenze senza parola d'ordine (sulla fiducia), oppure dovrebbe essere possibile definire dei contesti per cui l'accesso non debba richiedere questa formalità.

Il problema di evitare l'obbligo di inserire la parola d'ordine si sente in particolare proprio per l'accesso in qualità di amministratore, quando si vogliono realizzare degli script per svolgere certe funzioni amministrative. Le forme di raggiro dipendono dalle caratteristiche del DBMS.

Quando il DBMS è in grado di riconoscere un accesso locale, in quanto proveniente da un utente che ha lo stesso nome usato nell'ambito del sistema operativo, potrebbe accettarlo senza richiesta di una parola d'ordine, perché in sostanza l'autenticazione è già avvenuta. Questo meccanismo viene usato in particolare con PostgreSQL, dove l'utente '**`postgres`**' viene aggiunto anche nel file '`/etc/passwd`'; in tal modo, ammesso che la configurazione di PostgreSQL lo consenta, l'utente '**`root`**' del sistema operativo

può impersonare facilmente l'utente '**postgres**', attraverso il comando '**su**', quindi può accedere localmente al DBMS venendo riconosciuto implicitamente.

Quando non c'è la possibilità di sfruttare il sistema operativo per il riconoscimento dell'utente in modo implicito, si può arrivare ad annotare la parola d'ordine (in chiaro) in un file che solo l'utente '**root**' del sistema operativo può leggere, così che uno script con i privilegi necessari possa leggere questa parola d'ordine, usarla per collegarsi al DBMS e svolgere il suo compito.

Generalmente, le utenze vengono considerate nel DBMS soltanto come nominativi puri e semplici, senza distinguerne la provenienza. Il DBMS potrebbe disporre di una configurazione ulteriore in cui si specifica il metodo di riconoscimento richiesto, in base alla provenienza degli accessi (PostgreSQL), oppure potrebbe arrivare a considerare le utenze come l'unione del nome al nodo di origine, come se si trattasse di utenti distinti. Questo secondo caso riguarda in particolare MySQL e vale la pena di considerarlo con attenzione, perché si possono creare degli equivoci; infatti, se un'utenza è abbinata all'origine '**localhost**' e tale utente accede sì dall'elaboratore locale (come indica convenzionalmente il nome '**localhost**'), ma il file '`/etc/hosts`' non abbina correttamente l'indirizzo locale a tale nome, l'accesso fallisce; inoltre, se l'utenza fosse abbinata all'origine '**127.0.0.1**' e l'utente cercasse di accedere localmente, potrebbe succedergli di non essere riconosciuto se il sistema operativo traduce poi l'indirizzo nel nome.

Per quanto riguarda la gestione delle utenze, c'è da considerare che esistono anche dei DBMS semplificati che, concedendo esclusivamente accessi locali, invece di gestire le utenze in proprio si affi-

dano alla gestione del sistema operativo. In questi casi, i permessi di accesso alle basi di dati vengono regolati tramite la gestione dei permessi corrispondenti ai file che rappresentano le basi di dati stesse.

74.2.5 ODBC

«

ODBC, ovvero *Open database connectivity* è un metodo standardizzato per l'accesso ai DBMS. In pratica, si inserisce un servizio intermedio, tra i DBMS e le applicazioni che devono accedere ai dati: le applicazioni comunicano con il servizio ODBC; il servizio ODBC comunica con i DBMS sottostanti, preoccupandosi di adattarsi alle loro particolarità. In questo modo, invece di scrivere applicazioni che comunicano solo con un certo DBMS, le applicazioni fatte per ODBC, possono utilizzare qualsiasi DBMS che il servizio ODBC è in grado di gestire.

74.3 Introduzione a SQL

«

SQL è l'acronimo di *Structured query language* e identifica un linguaggio di interrogazione (gestione) per basi di dati relazionali. Le sue origini risalgono alla fine degli anni 1970 e questo giustifica la sua sintassi prolissa e verbale tipica dei linguaggi dell'epoca, come il COBOL.

Allo stato attuale, data la sua evoluzione e standardizzazione, l'SQL rappresenta un riferimento fondamentale per la gestione di una base di dati relazionale.

A parte il significato originale dell'acronimo, SQL è un linguaggio completo per la gestione di una base di dati relazionale, includendo

le funzionalità di un DDL (*Data definition language*), di un DML (*Data manipulation language*) e di un DCL (*Data control language*).

Data l'età e la conseguente evoluzione di questo linguaggio, si sono definiti nel tempo diversi livelli di standard. I più importanti sono SQL89, SQL92 e SQL99, noti anche come SQL1, SQL2 e SQL3 rispettivamente.

L'aderenza dei vari sistemi DBMS allo standard SQL92 non è mai completa e perfetta, per questo sono stati definiti dei sottolivelli di questo standard per definire il grado di compatibilità di un DBMS. Si tratta di: *entry SQL*, *intermediate SQL* e *full SQL*. Si può intendere che il primo sia il livello di compatibilità minima e l'ultimo rappresenti la compatibilità totale. Lo standard di fatto è rappresentato prevalentemente dal primo livello, che coincide fondamentalmente con lo standard precedente, SQL89. Da questo si comprende che lo stato di assimilazione di SQL99 è ancora più arretrato.

74.3.1 Concetti fondamentali

Convenzionalmente, le istruzioni di questo linguaggio sono scritte con tutte le lettere maiuscole. Si tratta solo di una tradizione di quell'epoca. SQL non distingue tra lettere minuscole e maiuscole nelle parole chiave delle istruzioni e nemmeno nei nomi di relazioni, attributi e altri oggetti. Solo quando si tratta di definire il contenuto di una variabile, allora le differenze contano.

In questo capitolo e nel resto del documento, quando si fa riferimento a istruzioni SQL, queste vengono indicate utilizzando solo lettere maiuscole, come richiede la tradizione.

I nomi degli oggetti (relazioni e altro) possono essere composti utilizzando lettere, numeri e il trattino basso; il primo carattere deve

essere una lettera oppure il trattino basso.

Le istruzioni SQL possono essere distribuite su più righe, senza una regola precisa. Si distingue la fine di un'istruzione dall'inizio di un'altra attraverso la presenza di almeno una riga vuota. Generalmente, i sistemi SQL richiedono l'uso di un simbolo di terminazione delle righe, che di norma è costituito dal punto e virgola.

L'SQL standard prevede la possibilità di inserire commenti; per questo si può usare un trattino doppio ('--') seguito dal commento desiderato, fino alla fine della riga. Tuttavia, si osservi che per ottenere la massima compatibilità con i DBMS esistenti, conviene lasciare almeno uno spazio dopo il trattino doppio, prima di inserire il commento vero e proprio.

74.3.2 Terminologia

«

Il linguaggio SQL utilizza una propria terminologia per distinguere gli «oggetti» che manipola. Per la precisione, si utilizzano normalmente i termini «tabella», «riga» e «colonna», al posto di «relazione», «tupla» e «attributo».

Esistono delle buone ragioni per utilizzare una terminologia differente nel linguaggio SQL, soprattutto in considerazione del fatto che in questo caso sono ammissibili situazioni che nella teoria generale delle basi di dati relazionali non lo sarebbero (per esempio la possibilità di avere tuple doppie). Tuttavia, si osservi che in questo documento si cerca di mantenere una certa uniformità nei termini, seguendo la tradizione della teoria delle basi di dati, anche a costo di rischiare una contraddizione con questa.

Specchietto 74.13. Associazione tra i termini relativi alla gestione delle basi di dati. Ogni riga dello specchietto, rappresenta un contesto differente, mentre le colonne individuano la traduzione dei termini in base al contesto.

classi	istanze	attributi	tipi di dati contenibili negli attributi
relazioni	tuple	attributi	domini
tabelle	righe	colonne	tipi di dati contenibili nelle colonne

74.4 DDL

DDL, ovvero *Data definition language*, è il linguaggio usato per definire la struttura dei dati (in una base di dati). In questa sezione viene trattato il linguaggio SQL per ciò che riguarda specificatamente i dati, la loro creazione e la loro distruzione.

74.4.1 Tipi di dati

I tipi di dati gestibili con il linguaggio SQL sono molti. Fondamentalmente si possono distinguere tipi contenenti: valori numerici, stringhe e informazioni data-orario. Nelle sezioni seguenti vengono descritti solo alcuni dei tipi definiti dallo standard.

74.4.1.1 Stringhe di caratteri

Si distinguono due tipi di stringhe di caratteri in SQL: quelle a dimensione fissa, completate a destra dal carattere spazio, e quelle a dimensione variabile.

Sintassi	Descrizione
`CHARACTER` `CHARACTER(`***dimensione***`)` `CHAR` `CHAR(`***dimensione***`)`	Queste sono le varie sintassi alternative che possono essere utilizzate per definire una stringa di dimensione fissa. Se non viene indicata la dimensione tra parentesi, si intende una stringa di un solo carattere.
`CHARACTER VARYING(`***dimensione***`)` `CHAR VARYING(`***dimensione***`)` `VARCHAR(`***dimensione***`)`	Una stringa di dimensione variabile può essere definita attraverso uno dei tre modi appena elencati. È necessario specificare la dimensione massima che questa stringa può avere. Il minimo è rappresentato dalla stringa nulla.

Le costanti stringa si esprimono delimitandole attraverso apici singoli, oppure apici doppi, come nell'esempio seguente:

```
'Questa è una stringa letterale per SQL'
"Anche questa è una stringa letterale per SQL"
```

Non tutti i sistemi SQL accettano entrambi i tipi di delimitatori di stringa. In caso di dubbio è bene limitarsi all'uso degli apici singoli; eventualmente, per inserire un apice singolo in una stringa delimitata con apici singoli, dovrebbe essere sufficiente il suo raddoppio. In pratica, per scrivere una stringa del tipo «l'albero», dovrebbe essere possibile scrivere:

```
'l''albero'
```

74.4.1.2 Valori numerici

I tipi numerici si distinguono in ***esatti*** e ***approssimati***, intendendo con la prima definizione quelli di cui si conosce il numero massimo di cifre numeriche intere e decimali, mentre con la seconda si fa riferimento ai tipi a virgola mobile. In ogni caso, le dimensioni massime o la precisione massima che possono avere tali valori dipende dal sistema in cui vengono utilizzati.

Sintassi	Descrizione
`NUMERIC` `NUMERIC(`***precisione*** [`,` ***scala***] `)`	Il tipo '**`NUMERIC`**' permette di definire un valore numerico composto da un massimo di tante cifre numeriche quante indicate dalla precisione, cioè il primo argomento tra parentesi. Se viene specificata anche la scala, si intende riservare quella parte di cifre per quanto appare dopo la virgola. Per esempio, con '**`NUMERIC(5,2)`**' si possono rappresentare valori da +999,99 a −999,99. Se non viene specificata la scala, si intende che si tratti solo di valori interi; se non viene specificata nemmeno la precisione, viene usata la definizione predefinita per questo tipo di dati, che dipende dalle caratteristiche del DBMS.

Sintassi	Descrizione
`DECIMAL` `DECIMAL(`***precisione*** [`,` ***scala***] `)` `DEC` `DEC(`***precisione*** [`,` ***scala***] `)`	Il tipo '**`DECIMAL`**' è simile al tipo '**`NUMERIC`**', con la differenza che le caratteristiche della precisione e della scala rappresentano le esigenze minime, mentre il sistema può fornire una rappresentazione con precisione o scala maggiore.
`INTEGER` `INT` `SMALLINT`	I tipi '**`INTEGER`**' e '**`SMALLINT`**' rappresentano tipi interi la cui dimensione dipende generalmente dalle caratteristiche del sistema operativo e dall'hardware utilizzato. L'unico riferimento sicuro è che il tipo '**`SMALLINT`**' permette di rappresentare interi con una dimensione inferiore o uguale al tipo '**`INTEGER`**'.
`FLOAT` `FLOAT(`***precisione***`)`	Il tipo '**`FLOAT`**' definisce un tipo numerico approssimato (a virgola mobile) con una precisione binaria pari o superiore di quella indicata tra parentesi (se non viene indicata, dipende dal sistema).
`REAL` `DOUBLE PRECISION`	Il tipo '**`REAL`**' e il tipo '**`DOUBLE PRECISION`**' sono due tipi a virgola mobile con una precisione prestabilita. Questa precisione dipende dal sistema, ma in generale, il secondo dei due tipi deve essere più preciso dell'altro.

I valori numerici costanti vengono espressi attraverso la semplice indicazione del numero senza delimitatori. La virgola di separazione della parte intera da quella decimale si esprime normalmente attraverso il punto (‘.’), a meno che sia prevista una forma di adattamento alle caratteristiche di configurazione locale.

74.4.1.3 Valori Data-orario

I valori data-orario sono di tre tipi e servono rispettivamente a memorizzare un giorno particolare, un orario normale e un’informazione data-ora completa.

Sintassi	Descrizione
`DATE`	Il tipo ‘**`DATE`**’ permette di rappresentare delle date composte dall’informazione anno-mese-giorno.
`TIME` `TIME(`***precisione***`)` `TIME WITH TIME ZONE` `TIME(`***precisione***`) WITH TIME ZONE`	Il tipo ‘**`TIME`**’ permette di rappresentare un orario particolare, composto da ore-minuti-secondi ed eventualmente frazioni di secondo. Se viene specificata la precisione, si intende definire un numero di cifre per la parte frazionaria dei secondi, altrimenti si intende che non debbano essere memorizzate le frazioni di secondo.

Sintassi	Descrizione
`TIMESTAMP` `TIMESTAMP(`***precisione***`)` `TIMESTAMP WITH TIME ZONE` `TIMESTAMP(`***precisione***`) WITH TIME ZONE`	Il tipo '**`TIMESTAMP`**' è un'informazione oraria più completa del tipo '**`TIME`**' in quanto prevede tutte le informazioni, dall'anno ai secondi, oltre alle eventuali frazioni di secondo.

Se viene specificata la precisione, si intende definire un numero di cifre per la parte frazionaria dei secondi, altrimenti si intende che non debbano essere memorizzate le frazioni di secondo.

L'aggiunta dell'opzione '**`WITH TIME ZONE`**' serve a specificare un tipo orario differente, che assieme all'informazione oraria aggiunge lo scostamento, espresso in ore e minuti, dell'ora locale dal tempo universale (UTC). Per esempio, 22:05:10+1:00 rappresenta le 22.05 e 10 secondi dell'ora locale italiana (durante l'inverno), mentre il tempo universale corrispondente sarebbe invece 21:05:10+0:00.

Le costanti che rappresentano informazioni data-orario sono espresse come le stringhe, delimitate tra apici. Il sistema DBMS potrebbe ammettere più forme differenti per l'inserimento di queste, ma i modi più comuni dovrebbero essere quelli espressi dagli esempi seguenti.

```
'2012-12-31'
'12/31/2012'
'31.12.2012'
```

Questi tre esempi rappresentano la stessa data: il 31 dicembre 1999. Per una questione di uniformità, dovrebbe essere preferibile il primo

di questi formati, corrispondente allo stile ISO 8601. Anche gli orari che si vedono sotto, sono aderenti allo stile ISO 8601; in particolare per il fatto che il fuso orario viene indicato attraverso lo scostamento dal tempo universale, invece che attraverso una parola chiave che definisca il fuso dell'ora locale.

```
'12:30:50+1.00'
'12:30:50.10'
'12:30:50'
'12:30'
```

Il primo di questa serie di esempi rappresenta un orario composto da ore, minuti e secondi, oltre all'indicazione dello scostamento dal tempo universale (per ottenere il tempo universale deve essere sottratta un'ora). Il secondo esempio mostra un orario composto da ore, minuti, secondi e centesimi di secondo. Il terzo e il quarto sono rappresentazioni normali, in particolare nell'ultimo è stata omessa l'indicazione dei secondi.

```
'2012-12-31 12:30:50+1.00'
'2012-12-31 12:30:50.10'
'2012-12-31 12:30:50'
'2012-12-31 12:30'
```

Gli esempi mostrano la rappresentazione di informazioni data-orario complete per il tipo '**`TIMESTAMP`**'. La data è separata dall'ora da uno spazio.

74.4.1.4 Intervalli di tempo

Quanto mostrato nella sezione precedente rappresenta un valore che indica un momento preciso nel tempo: una data o un orario, o entrambe le cose. Per rappresentare una durata, si parla di intervalli. Per l'SQL si possono gestire gli intervalli a due livelli di precisione:

anni e mesi; oppure giorni, ore, minuti, secondi ed eventualmente anche le frazioni di secondo. L'intervallo si indica con la parola chiave '**INTERVAL**', seguita eventualmente dalla precisione con cui questo deve essere rappresentato:

```
INTERVAL [unità_di_misura_data_orario [TO unità_di_misura_data_orario]]
```

In pratica, si può indicare che si tratta di un intervallo, senza specificare altro, oppure si possono definire una o due unità di misura che limitano la precisione di questo (pur restando nei limiti a cui si è già accennato). Tanto per fare un esempio concreto, volendo definire un intervallo che possa esprimere solo ore e minuti, si potrebbe dichiarare con: '**INTERVAL HOUR TO MINUTE**'. La tabella 74.22 elenca le parole chiave che rappresentano queste unità di misura.

Tabella 74.22. Elenco delle parole chiave che esprimono unità di misura data-orario.

Parola chiave	Significato
YEAR	Anni
MONTH	Mesi
DAY	Giorni
HOUR	Ore
MINUTE	Minuti
SECOND	Secondi

Si osservino i due esempi seguenti:

```
INTERVAL '12 HOUR 30 MINUTE 50 SECOND'
INTERVAL '12:30:50'
```

Queste due forme rappresentano entrambe la stessa cosa: una durata di 12 ore, 30 minuti e 50 secondi. In generale, dovrebbe essere preferibile la seconda delle due forme di rappresentazione.

```
INTERVAL '10 DAY 12 HOUR 30 MINUTE 50 SECOND'
INTERVAL '10 DAY 12:30:50'
```

Come prima, i due esempi che si vedono sopra sono equivalenti. Intuitivamente, si può osservare che non ci può essere un altro modo di esprimere una durata in giorni, senza specificarlo attraverso la parola chiave '**DAY**'.

Per completare la serie di esempi, si aggiungono anche i casi in cui si rappresentano esplicitamente quantità molto grandi, che di conseguenza sono approssimate al mese (come richiede lo standard SQL92):

```
INTERVAL '10 YEAR 11 MONTH'
INTERVAL '10 YEAR'
```

Gli intervalli di tempo possono servire per indicare un tempo trascorso rispetto al momento attuale. Per specificare espressamente questo fatto, si indica l'intervallo come un valore negativo, aggiungendo all'inizio un trattino (il segno meno).

```
INTERVAL '- 10 YEAR 11 MONTH'
```

L'esempio che si vede sopra, esprime precisamente 10 anni e 11 mesi fa.

74.4.2 Operatori, funzioni ed espressioni

«

SQL, pur non essendo un linguaggio di programmazione completo, mette a disposizione una serie di operatori e di funzioni utili per la realizzazione di espressioni di vario tipo.

74.4.2.1 Operatori aritmetici

«

Gli operatori che intervengono su valori numerici sono elencati nella tabella 74.27.

Tabella 74.27. Elenco degli operatori aritmetici.

Operatore e operandi	Descrizione
−op	Inverte il segno dell'operando.
op1 + *op2*	Somma i due operandi.
op1 − *op2*	Sottrae dal primo il secondo operando.
op1 * *op2*	Moltiplica i due operandi.
op1 / *op2*	Divide il primo operando per il secondo.
op1 % *op2*	Modulo: il resto della divisione tra il primo e il secondo operando.

Nelle espressioni, tutti i tipi numerici esatti e approssimati possono essere usati senza limitazioni. Dove necessario, il sistema provvede a eseguire le conversioni di tipo.

74.4.2.2 Operazioni con i valori data-orario e con intervalli di tempo

Le operazioni che si possono compiere utilizzando valori data-orario e valori che esprimono intervalli di tempo, hanno significato solo in alcune circostanze. La tabella 74.28 elenca le operazioni possibili e il tipo di risultato che si ottiene in base al tipo di operatori utilizzato.

Tabella 74.28. Operatori e operandi validi quando si utilizzano valori data-orario e valori che esprimono intervalli di tempo.

Operatore e operandi	Risultato
data_orario − ***data_orario***	Intervallo
data_orario + ***intervallo***	Data-orario
data_orario − ***intervallo***	Data-orario
intervallo + ***data_orario***	Data-orario
intervallo + ***intervallo***	Intervallo
intervallo − ***intervallo***	Intervallo
intervallo * ***numerico***	Intervallo
intervallo / ***numerico***	Intervallo
numerico * ***intervallo***	Intervallo

74.4.2.3 Operatori di confronto e operatori logici

«

Gli operatori di confronto determinano la relazione tra due operandi. Il risultato dell'espressione composta da due operandi posti a confronto è di tipo booleano: *Vero* o *Falso*. Gli operatori di confronto sono elencati nella tabella 74.29.

Tabella 74.29. Elenco degli operatori di confronto.

Operatore e operandi	Descrizione
op1 = ***op2***	*Vero* se gli operandi si equivalgono.
op1 <> ***op2***	*Vero* se gli operandi sono differenti.
op1 < ***op2***	*Vero* se il primo operando è minore del secondo.
op1 > ***op2***	*Vero* se il primo operando è maggiore del secondo.
op1 <= ***op2***	*Vero* se il primo operando è minore o uguale al secondo.
op1 >= ***op2***	*Vero* se il primo operando è maggiore o uguale al secondo.

Quando si vogliono combinare assieme diverse espressioni logiche si utilizzano gli operatori logici. Come in tutti i linguaggi di programmazione, si possono usare le parentesi tonde per raggruppare le espressioni logiche in modo da chiarire l'ordine di risoluzione. Gli operatori logici sono elencati nella tabella 74.30.

Tabella 74.30. Elenco degli operatori logici.

Operatore e operandi	Descrizione
`NOT` ***op***	Inverte il risultato logico dell'operando.
op1 `AND` ***op2***	*Vero* se entrambi gli operandi restituiscono il valore *Vero*.
op1 `OR` ***op2***	*Vero* se almeno uno degli operandi restituisce il valore *Vero*.

Il meccanismo di confronto tra due operandi numerici è evidente, mentre può essere meno evidente con le stringhe di caratteri. Per la precisione, il confronto tra due stringhe avviene senza tenere conto degli spazi finali, per cui, le stringhe '**`'ciao'`**' e '**`'ciao '`**' dovrebbero risultare uguali attraverso il confronto di uguaglianza con l'operatore '**`=`**'.

Con le stringhe, tuttavia, si possono eseguire dei confronti basati su modelli, attraverso gli operatori '**`IS LIKE`**' e '**`IS NOT LIKE`**'. Il modello può contenere dei metacaratteri rappresentati dal trattino basso ('_'), che rappresenta un carattere qualsiasi, e dal simbolo di percentuale ('`%`'), che rappresenta una sequenza qualsiasi di caratteri. La tabella 74.31 riassume quanto affermato.

Tabella 74.31. Espressioni sulle stringhe di caratteri.

Espressioni e modelli	Descrizione
stringa `IS LIKE` ***modello***	Restituisce *Vero* se il modello corrisponde alla stringa.
stringa `IS NOT LIKE` ***modello***	Restituisce *Vero* se il modello non corrisponde alla stringa.
_	Rappresenta un carattere qualsiasi.

Espressioni e modelli	Descrizione
`%`	Rappresenta una sequenza indeterminata di caratteri.

La presenza di valori indeterminati richiede la disponibilità di operatori di confronto in grado di determinarne l'esistenza. La tabella 74.32 riassume gli operatori ammissibili in questi casi.

Tabella 74.32. Espressioni di verifica dei valori indeterminati.

Operatori	Descrizione
espressione `IS NULL`	Restituisce *Vero* se l'espressione genera un risultato indeterminato.
espressione `IS NOT NULL`	Restituisce *Vero* se l'espressione non genera un risultato indeterminato.

Infine, occorre considerare una categoria particolare di espressioni che permettono di verificare l'appartenenza di un valore a un intervallo o a un elenco di valori. La tabella 74.33 riassume gli operatori utilizzabili.

Tabella 74.33. Espressioni per la verifica dell'appartenenza di un valore a un intervallo o a un elenco.

Operatori e operandi	Descrizione
op1 `IN (`***elenco***`)`	*Vero* se il primo operando è contenuto nell'elenco.
op1 `NOT IN (`***elenco***`)`	*Vero* se il primo operando non è contenuto nell'elenco.
op1 `BETWEEN` ***op2*** `AND` ***op3***	*Vero* se il primo operando è compreso tra il secondo e il terzo.
op1 `NOT BETWEEN` ***op2*** `AND` ***op3***	*Vero* se il primo operando non è compreso nell'intervallo.

74.4.3 Le relazioni dal punto di vista di SQL

SQL tratta le relazioni attraverso il modello tabellare; di conseguenza si adegua tutta la sua filosofia e il modo di esprimere i concetti nella sua documentazione. Pertanto, le relazioni per SQL sono delle «tabelle», che vengono definite nel modo seguente dalla documentazione standard.

- La tabella è un insieme di più righe. Una riga è una sequenza non vuota di valori. Ogni riga della stessa tabella ha la stessa cardinalità e contiene un valore per ogni colonna di quella tabella. L'i-esimo valore di ogni riga di una tabella è un valore dell'i-esima colonna di quella tabella. La riga è l'elemento che costituisce la più piccola unità di dati che può essere inserita in una tabella e cancellata da una tabella.
- Il grado di una tabella è il numero di colonne della stessa. In ogni momento, il grado della tabella è lo stesso della cardinalità di ognuna delle sue righe; la cardinalità della tabella (cioè il numero delle righe contenute) è la stessa della cardinalità di ognuna delle sue colonne. Una tabella la cui cardinalità sia zero viene definita come vuota.

In pratica, la tabella è un contenitore di informazioni organizzato in righe e colonne. La tabella viene identificata per nome, così anche le colonne, mentre le righe vengono identificate attraverso il loro contenuto.

Nel modello di SQL, le colonne sono ordinate, anche se ciò non è sempre un elemento indispensabile, dal momento che si possono identificare per nome. Inoltre sono ammissibili tabelle contenenti righe duplicate.

Si osservi, comunque, che nel resto di questo documento si preferisce la terminologia generale delle basi di dati, dove, al posto di tabelle, righe e colonne, si parla piuttosto di relazioni, tuple e attributi.

74.4.3.1 Creazione di una relazione

La creazione di una relazione avviene attraverso un'istruzione che può assumere un'articolazione molto complessa, a seconda delle caratteristiche particolari che da questa relazione si vogliono ottenere. La sintassi più semplice è quella seguente:

```
CREATE TABLE nome_relazione ( specifiche )
```

Tuttavia, sono proprio le specifiche indicate tra le parentesi tonde che possono tradursi in un sistema molto confuso. La creazione di una relazione elementare può essere espressa con la sintassi seguente:

```
CREATE TABLE nome_relazione (nome_attributo tipo [, ...])
```

In questo modo, all'interno delle parentesi vengono semplicemente elencati i nomi degli attributi seguiti dal tipo di dati che in essi possono essere contenuti. L'esempio seguente rappresenta l'istruzione necessaria a creare una relazione composta da cinque attributi, contenenti rispettivamente informazioni su: codice, cognome, nome, indirizzo e numero di telefono.

```
CREATE TABLE Indirizzi (
        Codice          integer,
        Cognome         char(40),
        Nome            char(40),
        Indirizzo       varchar(60),
        Telefono        varchar(40)
    )
```

74.4.3.2 Valori predefiniti

Quando si inseriscono delle tuple all'interno della relazione, in linea di principio è possibile che i valori corrispondenti ad attributi particolari non siano inseriti esplicitamente. Se si verifica questa situazione (purché ciò sia consentito dai vincoli), viene assegnato a questi elementi mancanti un valore predefinito. Questo può essere stabilito all'interno delle specifiche di creazione della relazione; in mancanza di tale definizione, viene assegnato '**NULL**', corrispondente al valore indefinito.

La sintassi necessaria a creare una relazione contenente le indicazioni sui valori predefiniti da utilizzare è la seguente:

```
CREATE TABLE nome_relazione (
                    nome_attributo tipo
                        [DEFAULT espressione]
                    [,…]
        )
```

L'esempio seguente crea la stessa relazione già vista nell'esempio precedente, specificando come valore predefinito per l'indirizzo, la stringa di caratteri: «sconosciuto».

```
CREATE TABLE Indirizzi (
        Codice          integer,
        Cognome         char(40),
        Nome            char(40),
        Indirizzo       varchar(60)   DEFAULT 'sconosciuto',
        Telefono        varchar(40)
    )
```

74.4.3.3 Vincoli interni alla relazione

«

Può darsi che in certe situazioni, determinati valori all'interno di una tupla non siano ammissibili, a seconda del contesto a cui si riferisce la relazione. I vincoli interni alla relazione sono quelli che possono essere risolti senza conoscere informazioni esterne alla relazione stessa.

Il vincolo più semplice da esprimere è quello di non ammissibilità dei valori indefiniti. La sintassi seguente ne mostra il modo.

```
CREATE TABLE nome_relazione (
                    nome_attributo  tipo
                        [NOT NULL]
                    [,…]
        )
```

L'esempio seguente crea la stessa relazione già vista negli esempi precedenti, specificando che il codice, il cognome, il nome e il telefono non possono essere indeterminati.

```
CREATE TABLE Indirizzi (
        Codice          integer         NOT NULL,
        Cognome         char(40)        NOT NULL,
        Nome            char(40)        NOT NULL,
        Indirizzo       varchar(60)     DEFAULT 'sconosciuto',
        Telefono        varchar(40)     NOT NULL
    )
```

Un altro vincolo importante è quello che permette di definire che un gruppo di attributi deve rappresentare dati unici in ogni tupla, cioè che non siano ammissibili tuple che per quel gruppo di attributi abbiano dati uguali. Segue lo schema sintattico relativo:

```
CREATE TABLE nome_relazione (
                nome_attributo tipo
                [,…],
                UNIQUE ( nome_attributo [,…] )
                [,…]
        )
```

L'indicazione dell'unicità può riguardare più gruppi di attributi in modo indipendente. Per ottenere questo si possono indicare più opzioni '**UNIQUE**'.

È il caso di osservare che il vincolo '**UNIQUE**' non è sufficiente per impedire che i dati possano essere indeterminati. Infatti, il valore indeterminato, '**NULL**', è diverso da ogni altro '**NULL**'.

L'esempio seguente crea la stessa relazione già vista negli esempi precedenti, specificando che i dati dell'attributo del codice devono

essere unici per ogni tupla.

```
CREATE TABLE Indirizzi (
        Codice          integer         NOT NULL,
        Cognome         char(40)        NOT NULL,
        Nome            char(40)        NOT NULL,
        Indirizzo       varchar(60)     DEFAULT 'sconosciuto',
        Telefono        varchar(40)     NOT NULL,
        UNIQUE (Codice)
    )
```

Quando un attributo, o un gruppo di attributi, costituisce un riferimento importante per identificare le varie tuple che compongono la relazione, si può utilizzare il vincolo '**`PRIMARY KEY`**', che può essere utilizzato una sola volta. Questo vincolo stabilisce anche che i dati contenuti, oltre a non poter essere doppi, non possono essere indefiniti.

```
CREATE TABLE nome_relazione (
                    nome_attributo tipo
                    [,...],
                    PRIMARY KEY ( nome_attributo [,...] )
        )
```

L'esempio seguente crea la stessa relazione già vista negli esempi precedenti specificando che l'attributo del codice deve essere considerato come chiave primaria.

```
CREATE TABLE Indirizzi (
        Codice          integer,
        Cognome         char(40)        NOT NULL,
        Nome            char(40)        NOT NULL,
        Indirizzo       varchar(60)     DEFAULT 'sconosciuto',
        Telefono        varchar(40)     NOT NULL,
        PRIMARY KEY (Codice)
    )
```

74.4.3.4 Vincoli esterni alla relazione

I vincoli esterni alla relazione riguardano principalmente la connessione con altre relazioni e la necessità che i riferimenti a queste siano validi. La definizione formale di questa connessione è molto complessa e qui non viene descritta. Si tratta, in ogni caso, dell'opzione '**`FOREIGN KEY`**' seguita da '**`REFERENCES`**'.

Vale la pena però di considerare i meccanismi che sono coinvolti. Infatti, nel momento in cui si inserisce un valore, il sistema può impedire l'operazione perché non valida in base all'assenza di quel valore in un'altra relazione esterna specificata. Il problema nasce però nel momento in cui nella relazione esterna viene eliminata o modificata una tupla che è oggetto di un riferimento da parte della prima. Si pongono le alternative seguenti.

Vincolo	Descrizione
`CASCADE`	Se nella relazione esterna il dato a cui si fa riferimento è stato cambiato, viene cambiato anche il riferimento nella relazione di partenza; se nella relazione esterna la tupla corrispondente viene rimossa, viene rimossa anche la tupla della relazione di partenza.
`SET NULL`	Se viene a mancare l'oggetto a cui si fa riferimento, viene modificato il dato attribuendo il valore indefinito.

Vincolo	Descrizione
`SET DEFAULT`	Se viene a mancare l'oggetto a cui si fa riferimento, viene modificato il dato attribuendo il valore predefinito.
`NO ACTION`	Se viene a mancare l'oggetto a cui si fa riferimento, non viene modificato il dato contenuto nella relazione di partenza.

Le azioni da compiere si possono distinguere in base all'evento che ha causato la rottura del riferimento: cancellazione della tupla della relazione esterna o modifica del suo contenuto.

74.4.3.5 Modifica della struttura della relazione

La modifica della struttura di una relazione riguarda principalmente la sua organizzazione in attributi. Le cose più semplici che si possono desiderare di fare sono l'aggiunta di nuovi attributi e l'eliminazione di attributi già esistenti. Vedendo il problema in questa ottica, la sintassi si riduce ai due casi seguenti:

```
ALTER TABLE nome_relazione ADD [COLUMN]
            nome_attributo tipo [altre_caratteristiche]
```

```
ALTER TABLE nome_relazione DROP [COLUMN] nome_attributo
```

Nel primo caso si aggiunge un attributo, del quale si deve specificare il nome, il tipo ed eventualmente i vincoli; nel secondo si tratta solo di indicare l'attributo da eliminare. A livello di singolo attributo può essere eliminato o assegnato un valore predefinito.

```
ALTER TABLE nome_relazione ALTER [COLUMN]
            nome_attributo DROP DEFAULT
```

```
ALTER TABLE nome_relazione ALTER [COLUMN]
            nome_attributo SET DEFAULT valore_predefinito
```

74.4.3.6 Eliminazione di una relazione

L'eliminazione di una relazione, con tutto il suo contenuto, è un'operazione semplice che dovrebbe essere autorizzata solo all'utente che l'ha creata.

```
DROP TABLE nome_relazione
```

74.5 DML

DML, ovvero *Data manipulation language*, è il linguaggio usato per inserire, modificare e accedere ai dati. In questa sezione viene trattato il linguaggio SQL per ciò che riguarda specificatamente l'inserimento, la lettura e la modifica del contenuto delle relazioni.

74.5.1 Inserimento, eliminazione e modifica dei dati

L'inserimento, l'eliminazione e la modifica dei dati di una relazione è un'operazione che interviene sempre a livello delle tuple. Infatti, come già definito, la tupla è l'elemento che costituisce l'unità di dati più piccola che può essere inserita o cancellata da una relazione.

74.5.1.1 Inserimento di tuple

«

L'inserimento di una nuova tupla all'interno di una relazione viene eseguito attraverso l'istruzione '**INSERT**'. Dal momento che nel modello di SQL gli attributi sono ordinati, è sufficiente indicare ordinatamente l'elenco dei valori della tupla da inserire, come mostra la sintassi seguente:

```
INSERT INTO nome_relazione VALUES (espressione_1 [,...espressione_n])
```

Per esempio, l'inserimento di una tupla nella relazione '**Indirizzi**' già mostrata in precedenza, potrebbe avvenire nel modo seguente:

```
INSERT INTO Indirizzi
    VALUES (
        01,
        'Pallino',
        'Pinco',
        'Via Biglie 1',
        '0222,222222'
    )
```

Se i valori inseriti sono meno del numero degli attributi della relazione, i valori mancanti, in coda, ottengono quanto stabilito come valore predefinito, o '**NULL**' in sua mancanza (sempre che ciò sia concesso dai vincoli della relazione).

L'inserimento dei dati può avvenire in modo più chiaro e sicuro elencando prima i nomi degli attributi, in modo da evitare di dipendere dalla sequenza degli attributi memorizzata nella relazione. La sintassi seguente mostra il modo di ottenere questo.

```
INSERT INTO nome_relazione (attributo_1 [,...attributo_n]) ]
    VALUES (espressione_1 [,...espressione_n] )
```

L'esempio già visto potrebbe essere tradotto nel modo seguente, più prolisso, ma anche più chiaro:

```
INSERT INTO Indirizzi (
        Codice,
        Cognome,
        Nome,
        Indirizzo,
        Telefono
    )
    VALUES (
        01,
        'Pallino',
        'Pinco',
        'Via Biglie 1',
        '0222,222222'
    )
```

Questo modo esplicito di fare riferimento agli attributi garantisce anche che eventuali modifiche di lieve entità nella struttura della relazione non debbano necessariamente riflettersi nei programmi. L'esempio seguente mostra l'inserimento di alcuni degli attributi della tupla, lasciando che gli altri ottengano l'assegnamento di un valore predefinito.

```
INSERT INTO Indirizzi (
        Codice,
        Cognome,
        Nome,
        Telefono
```

```
    )
    VALUES (
        01,
        'Pinco',
        'Pallino',
        '0222,222222'
    )
```

74.5.1.2 Aggiornamento delle tuple

La modifica delle tuple può avvenire attraverso una scansione della relazione, dalla prima all'ultima tupla, eventualmente controllando la modifica in base all'avverarsi di determinate condizioni. La sintassi per ottenere questo risultato, leggermente semplificata, è la seguente:

```
UPDATE relazione
    SET attributo_1=espressione_1 [ ,...attributo_n=espressione_n ]
    [WHERE condizione ]
```

L'istruzione '**UPDATE**' esegue tutte le sostituzioni indicate dalle coppie ***attributo=espressione***, per tutte le tuple in cui la condizione posta dopo la parola chiave '**WHERE**' si avvera. Se tale condizione manca, l'effetto delle modifiche si riflette su tutte le tuple della relazione.

L'esempio seguente aggiunge un attributo alla relazione degli indirizzi, per contenere il nome del comune di residenza; successivamente viene inserito il nome del comune «Sferopoli» in base al prefisso telefonico.

```
ALTER TABLE Indirizzi ADD COLUMN Comune char(30)
```

```
UPDATE Indirizzi
    SET Comune='Sferopoli'
    WHERE Telefono >= '022' AND Telefono < '023'
```

Eventualmente, al posto dell'espressione si può indicare la parola chiave '**DEFAULT**' che fa in modo di assegnare il valore predefinito per quel attributo.

74.5.1.3 Eliminazione di tuple

La cancellazione di tuple da una relazione è un'operazione molto semplice. Richiede solo l'indicazione del nome della relazione e la condizione in base alla quale le tuple devono essere cancellate.

```
DELETE FROM relazione [WHERE condizione]
```

Se la condizione non viene indicata, si cancellano tutte le tuple!

74.5.2 Interrogazioni di relazioni

L'interrogazione di una relazione è l'operazione con cui si ottengono i dati contenuti al suo interno, in base a dei criteri di filtro determinati. L'interrogazione consente anche di combinare assieme dati provenienti da relazioni differenti, in base a dei «collegamenti» che possono intercorrere tra queste.

74.5.2.1 Interrogazioni elementari

La forma più semplice di esprimere la sintassi necessaria a interrogare **una** sola relazione è quella espressa dallo schema seguente:

```
SELECT espress_col_1 [ , ...espress_col_n ]
    FROM relazione
    [WHERE condizione ]
```

In questo modo è possibile definire gli attributi che si intendono utilizzare per il risultato, mentre le tuple si specificano, eventualmente, con la condizione posta dopo la parola chiave '**WHERE**'. L'esempio seguente mostra la proiezione degli attributi del cognome e nome della relazione di indirizzi già vista negli esempi delle altre sezioni, senza porre limiti alle tuple.

```
SELECT Cognome, Nome FROM Indirizzi
```

Quando si vuole ottenere una selezione composta dagli stessi attributi della relazione originale, nel suo stesso ordine, si può utilizzare un carattere jolly particolare, l'asterisco ('*'). Questo rappresenta l'elenco di tutti gli attributi della relazione indicata.

```
SELECT * FROM Indirizzi
```

È bene osservare che gli attributi si esprimono attraverso un'espressione, questo significa che gli attributi a cui si fa riferimento sono quelle del risultato finale, cioè della relazione che viene restituita come selezione o proiezione della relazione originale. L'esempio seguente emette un solo attributo contenente un ipotetico prezzo scontato del 10 %, in pratica viene moltiplicato il valore di un attributo contenente il prezzo per 0,90, in modo da ottenerne il 90 % (100 % meno lo sconto).

```
SELECT Prezzo * 0.90 FROM Listino
```

In questo senso si può comprendere l'utilità di assegnare esplicita-

mente un nome agli attributi del risultato finale, come indicato dalla sintassi seguente:

```
SELECT espress_col_1 AS nome_col_1 [,...espress_col_n AS nome_col_n]
    FROM relazione
    [WHERE condizione]
```

In questo modo, l'esempio precedente può essere trasformato come segue, dando un nome all'attributo generato e chiarendone così il contenuto.

```
SELECT Prezzo * 0.90 AS Prezzo_Scontato FROM Listino
```

Finora è stata volutamente ignorata la condizione che controlla le tuple da selezionare. Anche se potrebbe essere evidente, è bene chiarire che la condizione posta dopo la parola chiave '**WHERE**' può fare riferimento solo ai dati originali della relazione da cui si attingono. Quindi, non è valida una condizione che utilizza un riferimento a un nome che appare dopo la parola chiave '**AS**' abbinata alle espressioni degli attributi.

Per qualche motivo che viene chiarito in seguito, può essere conveniente associare un alias alla relazione da cui estrarre i dati. Anche in questo caso si utilizza la parola chiave '**AS**', come indicato dalla sintassi seguente:

```
SELECT specificazione_dell'attributo_1 [,...specificazione_dell'attributo_n]
    FROM relazione AS alias
    [WHERE condizione]
```

Quando si vuole fare riferimento al nome di un attributo, se per qual-

che motivo questo nome dovesse risultare ambiguo, si può aggiungere anteriormente il nome della relazione a cui appartiene, separandolo attraverso l'operatore punto ('.'). L'esempio seguente è la proiezione dei cognomi e dei nomi della solita relazione degli indirizzi. In questo caso, le espressioni degli attributi rappresentano solo gli attributi corrispondenti della relazione originaria, con l'aggiunta dell'indicazione esplicita del nome della relazione stessa.

```
SELECT Indirizzi.Cognome, Indirizzi.Nome FROM Indirizzi
```

A questo punto, se al nome della relazione viene abbinato un alias, si può esprimere la stessa cosa indicando il nome dell'alias al posto di quello della relazione, come nell'esempio seguente:

```
SELECT Ind.Cognome, Ind.Nome FROM Indirizzi AS Ind
```

74.5.2.2 Interrogazioni ordinate

«

Per ottenere un elenco ordinato in base a qualche criterio, si utilizza l'istruzione '**SELECT**' con l'indicazione di un'espressione in base alla quale effettuare l'ordinamento. Questa espressione è preceduta dalle parole chiave '**ORDER BY**':

```
SELECT espress_col_1 [,...espress_col_n]
    FROM relazione
    [WHERE condizione]
    ORDER BY espressione [ASC|DESC] [,...]
```

L'espressione può essere il nome di un attributo, oppure un'espressione che genera un risultato da uno o più attributi; l'aggiunta eventuale della parola chiave '**ASC**', o '**DESC**', permette di specificare un

ordinamento crescente, o discendente. Come si vede, le espressioni di ordinamento possono essere più di una, separate con una virgola.

```
SELECT Cognome, Nome FROM Indirizzi ORDER BY Cognome
```

L'esempio mostra un'applicazione molto semplice del problema, in cui si ottiene un elenco dei soli attributi '**Cognome**' e '**Nome**', della relazione '**Indirizzi**', ordinato per '**Cognome**'.

```
SELECT Cognome, Nome FROM Indirizzi ORDER BY Cognome, Nome
```

Questo esempio, aggiunge l'indicazione del nome nella chiave di ordinamento, in modo che in presenza di cognomi uguali, la scelta venga fatta in base al nome.

```
SELECT Cognome, Nome
    FROM Indirizzi
    ORDER BY TRIM( Cognome ), TRIM( Nome )
```

Questo ultimo esempio mostra l'utilizzo di due espressioni come chiave di ordinamento. Per la precisione, la funzione ***TRIM()***, usata in questo modo, serve a eliminare gli spazi iniziali e finali superflui. In questo modo, se i nomi e i cognomi sono stati inseriti con degli spazi iniziali, questi non vanno a influire sull'ordinamento.

74.5.2.3 Interrogazioni simultanee di più relazioni

Se dopo la parola chiave '**FROM**' si indicano più relazioni (ciò vale anche se si indica più volte la stessa relazione), si intende fare riferimento a una relazione generata dal prodotto di queste. Se per esempio si vogliono abbinare due relazioni, una di tre tuple con due attributi e un'altra di due tuple con due attributi, quello che si ottiene è una relazione con quattro attributi composta da sei tuple. Infatti,

ogni tupla della prima relazione risulta abbinata con ogni tupla della seconda.

```
SELECT specificazione_dell'attributo_1 [ , ...specificazione_dell'attributo_n ]
    FROM specificazione_della_relazione_1 [ , ...specificazione_della_relazione_n ]
    [WHERE condizione ]
```

Viene proposto un esempio banalizzato, con il quale poi si vuole eseguire un'elaborazione (figura 74.54).

Figura 74.54. Relazioni '**Articoli**' e '**Movimenti**' di una gestione del magazzino ipotetica.

Articoli	
Codice	**Descrizione**
vite30	Vite 3 mm
dado30	Dado 3 mm
rond50	Rondella 5 mm

Movimenti			
Codice	**Data**	**Carico**	**Scarico**
dado30	01/01/2012	1200	
vite30	01/01/2012		800
vite30	03/01/2012	2000	
rond50	03/01/2012		500

Da questa situazione si vuole ottenere la congiunzione della relazione '**Movimenti**' con tutte le informazioni corrispondenti della relazione '**Articoli**', basando il riferimento sull'attributo '**Codice**'. In pratica si vuole ottenere la relazione della figura 74.55.

Tabella 74.55. Risultato del join che si intende ottenere tra la relazione '**Movimenti**' e la relazione '**Articoli**'.

Codice	Data	Carico	Scarico	Descrizione
dado30	01/01/2012	1200		Dado 3 mm
vite30	01/01/2012	2000		Vite 3 mm
vite30	03/01/2012		800	Vite 3 mm
rond50	03/01/2012		500	Rondella 5 mm

Considerato che da un'istruzione '**SELECT**' contenente il riferimento a più relazioni si genera il prodotto tra queste, si pone poi il problema di eseguire una proiezione degli attributi desiderati e, soprattutto, di selezionare le tuple. In questo caso, la selezione deve essere basata sulla corrispondenza tra l'attributo '**Codice**' della prima relazione, con lo stesso attributo della seconda. Dovendo fare riferimento a due attributi di relazioni differenti, aventi però lo stesso nome, diviene indispensabile indicare i nomi degli attributi prefissandoli con i nomi delle relazioni rispettive.

```
SELECT
    Movimenti.Codice,
    Movimenti.Data,
    Movimenti.Carico,
    Movimenti.Scarico,
    Articoli.Descrizione
    FROM Movimenti, Articoli
    WHERE Movimenti.Codice = Articoli.Codice;
```

L'interrogazione simultanea di più relazioni si presta anche per elaborazioni della stessa relazione più volte. In tal caso, diventa obbligatorio l'uso degli alias. Si osservi il caso seguente:

```
SELECT Ind1.Cognome, Ind1.Nome
    FROM Indirizzi AS Ind1, Indirizzi AS Ind2
    WHERE
        Ind1.Cognome = Ind2.Cognome
    AND Ind1.Nome <> Ind2.Nome
```

Il senso di questa interrogazione, che utilizza la stessa relazione degli indirizzi per due volte con due alias differenti, è quello di ottenere l'elenco delle persone che hanno lo stesso cognome, avendo però un nome differente.

Esiste anche un'altra situazione in cui si ottiene l'interrogazione simultanea di più relazioni: l'***unione***. Si tratta semplicemente di attaccare il risultato di un'interrogazione su una relazione con quello di un'altra relazione, quando gli attributi finali appartengono allo stesso tipo di dati.

```
SELECT specificazione_attributo_1 [ , ...specificazione_attributo_n ]
        FROM specificazione_relazione_1 [ , ...specificazione_relazione_n ]
        [WHERE condizione ]
    UNION
        SELECT specificatore_attributo_1 [ , ...specificazione_attributo_n ]
            FROM specificazione_relazione_1 [ , ...specificazione_relazione_n ]
            [WHERE condizione ]
```

Lo schema sintattico dovrebbe essere abbastanza esplicito: si uniscono due istruzioni '**SELECT**' in un risultato unico, attraverso la parola chiave '**UNION**'.

74.5.2.4 Condizioni

La condizione che esprime la selezione delle tuple può essere composta come si vuole, purché il risultato sia di tipo logico e i dati a cui si fa riferimento provengano dalle relazioni di partenza. Quindi si possono usare anche altri operatori di confronto, funzioni e operatori booleani.

È bene ricordare che il valore indefinito, rappresentato da '**NULL**', è diverso da qualunque altro valore, compreso un altro valore indefinito. Per verificare che un valore sia o non sia indefinito, si deve usare l'operatore '**IS NULL**' oppure '**IS NOT NULL**'.

74.5.2.5 Aggregazioni

L'aggregazione è una forma di interrogazione attraverso cui si ottengono risultati riepilogativi del contenuto di una relazione, in forma di relazione contenente una sola tupla. Per questo si utilizzano delle funzioni speciali al posto dell'espressione che esprime gli attributi del risultato. Queste funzioni restituiscono un solo valore e come tali concorrono a creare un'unica tupla. Le funzioni di aggregazione sono: ***COUNT()***, ***SUM()***, ***MAX()***, ***MIN()***, ***AVG()***. Per intendere il problema, si osservi l'esempio seguente:

```
SELECT COUNT(*) FROM Movimenti WHERE …
```

In questo caso, quello che si ottiene è solo il numero di tuple della relazione '**Movimenti**' che soddisfano la condizione posta dopo la parola chiave '**WHERE**' (qui non è stata indicata). L'asterisco posto co-

me parametro della funzione ***COUNT()*** rappresenta effettivamente l'elenco di tutti i nomi degli attributi della relazione '**Movimenti**'.

Quando si utilizzano funzioni di questo tipo, occorre considerare che l'elaborazione si riferisce alla relazione virtuale generata dopo la selezione posta da '**WHERE**'.

La funzione ***COUNT()*** può essere descritta attraverso la sintassi seguente:

```
COUNT( * )
```

```
COUNT( [DISTINCT|ALL] lista_attributi)
```

Utilizzando la forma già vista, quella dell'asterisco, si ottiene solo il numero delle tuple della relazione. L'opzione '**DISTINCT**', seguita da una lista di nomi di attributi, fa in modo che vengano contate le tuple contenenti valori differenti per quel gruppo di attributi. L'opzione '**ALL**' è implicita quando non si usa '**DISTINCT**' e indica semplicemente di contare tutte le tuple.

Il conteggio delle tuple esclude in ogni caso quelle in cui il contenuto di tutti gli attributi selezionati è indefinito ('**NULL**').

Le altre funzioni aggreganti non prevedono l'asterisco, perché fanno riferimento a un'espressione che genera un risultato per ogni tupla ottenuta dalla selezione.

```
SUM( [DISTINCT|ALL] espressione)
```

```
MAX( [DISTINCT|ALL] espressione)
```

```
MIN( [DISTINCT|ALL] espressione)
```

```
AVG( [DISTINCT|ALL] espressione)
```

In linea di massima, per tutti questi tipi di funzioni aggreganti, l'espressione deve generare un risultato numerico, sul quale calcolare la sommatoria, ***SUM()***, il valore massimo, ***MAX()***, il valore minimo, ***MIN()***, la media, ***AVG()***.

L'esempio seguente calcola lo stipendio medio degli impiegati, ottenendo i dati da un'ipotetica relazione '**`Emolumenti`**', limitandosi ad analizzare le tuple riferite a un certo settore.

```
SELECT AVG( Stipendio ) FROM Emolumenti
    WHERE Settore = 'Amministrazione'
```

L'esempio seguente è una variante in cui si estraggono rispettivamente lo stipendio massimo, medio e minimo.

```
SELECT MAX( Stipendio ), AVG( Stipendio ), MIN( Stipendio )
    FROM Emolumenti WHERE Settore = 'Amministrazione'
```

L'esempio seguente è invece volutamente **errato**, perché si mescolano funzioni aggreganti assieme a espressioni di attributi normali.

```
-- Esempio errato
SELECT MAX( Stipendio ), Settore FROM Emolumenti
    WHERE Settore = 'Amministrazione'
```

74.5.2.6 Raggruppamenti

«

Le aggregazioni possono essere effettuate in riferimento a gruppi di tuple, distinguibili in base al contenuto di uno o più attributi. In questo tipo di interrogazione si può generare solo una relazione composta da tanti attributi quanti sono quelli presi in considerazione dall'opzione di raggruppamento, assieme ad altre contenenti solo espressioni di aggregazione.

Alla sintassi normale già vista nelle sezioni precedenti, si aggiunge la clausola '**GROUP BY**'.

```
SELECT specificazione_dell'attributo_1 [ , ...specificazione_dell'attributo_n ]
    FROM specificazione_della_relazione_1 [ , ...specificazione_della_relazione_n ]
    [WHERE condizione ]
    GROUP BY attributo_1 [ , ... ]
```

Per comprendere l'effetto di questa sintassi, si deve scomporre idealmente l'operazione di selezione da quella di raggruppamento:

1. la relazione ottenuta dall'istruzione '**SELECT**...**FROM**' viene filtrata dalla condizione '**WHERE**';
2. la relazione risultante viene riordinata in modo da raggruppare le tuple in cui i contenuti degli attributi elencati dopo l'opzione '**GROUP BY**' sono uguali;
3. su questi gruppi di tuple vengono valutate le funzioni di aggregazione.

Figura 74.62. Carichi e scarichi in magazzino.

Movimenti				
Codice	**Data**	**Carico**	**Scarico**	**...**
vite40	01/01/2012	1200		...
vite30	01/01/2012		800	...
vite40	01/01/2012	1500		...
vite30	02/01/2012		1000	...
vite30	03/01/2012	2000		...
rond50	03/01/2012		500	...
vite40	04/01/2012	2200		...

Si osservi la relazione riportata in figura 74.62, mostra la solita sequenza di carichi e scarichi di magazzino. Si potrebbe porre il problema di conoscere il totale dei carichi e degli scarichi per ogni articolo di magazzino. La richiesta può essere espressa con l'istruzione seguente:

```
SELECT Codice, SUM( Carico ), SUM( Scarico ) FROM Movimenti
    GROUP BY Codice
```

Quello che si ottiene appare nella figura 74.64.

Figura 74.64. Carichi e scarichi totali.

Codice	**SUM(Carico)**	**SUM(Scarico)**
vite40	4900	
vite30	2000	1800
rond50		500

Volendo si possono fare i raggruppamenti in modo da avere i totali distinti anche in base al giorno, come nell'istruzione seguente:

```
SELECT Codice, Data, SUM( Carico ), SUM( Scarico )
    FROM Movimenti GROUP BY Codice, Data
```

Come già affermato, la condizione posta dopo la parola chiave '**WHERE**' serve a filtrare inizialmente le tuple da considerare nel raggruppamento. Se quello che si vuole è filtrare ulteriormente il risultato di un raggruppamento, occorre usare la clausola '**HAVING**'.

```
SELECT specificazione_dell'attributo_1 [, ...specificazione_dell'attributo_n]
    FROM specificazione_della_relazione_1 [, ...specificazione_della_relazione_n]
    [WHERE condizione]
    GROUP BY attributo_1 [, ...]
    HAVING condizione
```

L'esempio seguente serve a ottenere il raggruppamento dei carichi e scarichi degli articoli, limitando però il risultato a quelli per i quali sia stata fatta una quantità di scarichi consistente (superiore a 1000 unità).

```
SELECT Codice, SUM( Carico ), SUM( Scarico ) FROM Movimenti
    GROUP BY Codice
    HAVING SUM( Scarico ) > 1000
```

Dall'esempio già visto in figura 74.64 risulterebbe escluso l'articolo '**rond50**'.

74.5.3 Trasferimento di dati in un'altra relazione

«

Alcune forme particolari di interrogazioni SQL possono essere utilizzate per inserire dati in relazioni esistenti o per crearne di nuove.

74.5.3.1 Creazione di una nuova relazione a partire da altre

L'istruzione '**SELECT**' può servire per creare una nuova relazione a partire dai dati ottenuti dalla sua interrogazione.

```
SELECT specificazione_dell'attributo_1 [ , ...specificazione_dell'attributo_n ]
    INTO TABLE relazione_da_generare
    FROM specificazione_della_relazione_1 [ , ...specificazione_della_relazione_n ]
    [WHERE condizione ]
```

L'esempio seguente crea la relazione '**Mia_prova**' come risultato della fusione delle relazioni '**Indirizzi**' e '**Presenze**'.

```
SELECT
    Presenze.Giorno,
    Presenze.Ingresso,
    Presenze.Uscita,
    Indirizzi.Cognome,
    Indirizzi.Nome
    INTO TABLE Mia_prova
    FROM Presenze, Indirizzi
    WHERE Presenze.Codice = Indirizzi.Codice;
```

74.5.3.2 Inserimento in una relazione esistente

L'inserimento di dati in una relazione esistente prelevando da dati contenuti in altre, può essere fatta attraverso l'istruzione '**INSERT**' sostituendo la clausola '**VALUES**' con un'interrogazione ('**SELECT**').

```
INSERT INTO nome_relazione [(attributo_1...attributo_n)]
    SELECT espressione_1, ... espressione_n
        FROM relazioni_di_origine
        [WHERE condizione]
```

L'esempio seguente aggiunge alla relazione dello storico delle presenze le registrazioni vecchie che poi vengono cancellate.

```
INSERT INTO PresenzeStorico (
        PresenzeStorico.Codice,
        PresenzeStorico.Giorno,
        PresenzeStorico.Ingresso,
        PresenzeStorico.Uscita
    )
    SELECT
        Presenze.Codice,
        Presenze.Giorno,
        Presenze.Ingresso,
        Presenze.Uscita
        FROM Presenze
        WHERE Presenze.Giorno <= '01/01/1999';

DELETE FROM Presenze WHERE Giorno <= '01/01/1999';
```

74.5.4 Viste

«

Le viste sono delle relazioni virtuali ottenute a partire da relazioni vere e proprie o da altre viste, purché non si formino ricorsioni. Il concetto non dovrebbe risultare strano. In effetti, il risultato delle interrogazioni è sempre in forma di relazione. La vista crea una sorta di interrogazione permanente che acquista la personalità di una relazione normale.

```
CREATE VIEW nome_vista [(attributo_1 [,...attributo_n)]]
    AS richiesta
```

Dopo la parola chiave '**AS**' deve essere indicato ciò che compone un'istruzione '**SELECT**'. L'esempio seguente, genera la vista dei movimenti di magazzino del solo articolo '**vite30**'.

```
CREATE VIEW Movimenti_Vite30
    AS SELECT Codice, Data, Carico, Scarico
        FROM Movimenti
        WHERE Codice = 'vite30'
```

L'eliminazione di una vista si ottiene con l'istruzione '**DROP VIEW**', come illustrato dallo schema sintattico seguente:

```
DROP VIEW nome_vista
```

Volendo eliminare la vista '**Movimenti_Vite30**', si può intervenire semplicemente come nell'esempio seguente:

```
DROP VIEW Movimenti_Vite30
```

74.5.5 Cursori

Quando il risultato di un'interrogazione SQL deve essere gestito all'interno di un programma, si pone un problema nel momento in cui ciò che si ottiene è composto da più di una sola tupla. Per poter scorrere un elenco ottenuto attraverso un'istruzione '**SELECT**', tupla per tupla, si deve usare un ***cursore***.

La dichiarazione e l'utilizzo di un cursore avviene all'interno di una transazione. Quando la transazione si chiude attraverso un'istruzione '**COMMIT**' o '**ROLLBACK**', si chiude anche il cursore.

74.5.5.1 Dichiarazione e apertura

«

L'SQL prevede due fasi prima dell'utilizzo di un cursore: la dichiarazione e la sua apertura:

```
DECLARE cursore [INSENSITIVE] [SCROLL] CURSOR FOR
    SELECT …
```

```
OPEN cursore
```

Nella dichiarazione, la parola chiave '**INSENSITIVE**' serve a stabilire che il risultato dell'interrogazione che si scandisce attraverso il cursore, non deve essere sensibile alle variazioni dei dati originali; la parola chiave '**SCROLL**' indica che è possibile estrarre più tuple simultaneamente attraverso il cursore.

```
DECLARE Mio_cursore CURSOR FOR
    SELECT
        Presenze.Giorno,
        Presenze.Ingresso,
        Presenze.Uscita,
        Indirizzi.Cognome,
        Indirizzi.Nome
        FROM Presenze, Indirizzi
        WHERE Presenze.Codice = Indirizzi.Codice;
```

L'esempio mostra la dichiarazione del cursore '**Mio_cursore**', abbinato alla selezione degli attributi composti dal collegamento di due relazioni, '**Presenze**' e '**Indirizzi**', dove le tuple devono avere lo stesso numero di codice. Per attivare questo cursore, lo si deve aprire come nell'esempio seguente:

```
OPEN Mio_cursore
```

74.5.5.2 Scansione

La scansione di un'interrogazione inserita in un cursore, avviene attraverso l'istruzione '**FETCH**'. Il suo scopo è quello di estrarre una tupla alla volta, in base a una posizione, relativa o assoluta.

```
FETCH [ [ NEXT | PRIOR | FIRST | LAST
              | { ABSOLUTE | RELATIVE } n ]
    FROM cursore ]
    INTO :variabile [,…]
```

Le parole chiave '**NEXT**', '**PRIOR**', '**FIRST**', '**LAST**', permettono rispettivamente di ottenere la tupla successiva, quella precedente, la prima e l'ultima. Le parole chiave '**ABSOLUTE**' e '**RELATIVE**' sono seguite da un numero, corrispondente alla scelta della tupla ***n***-esima, rispetto all'inizio del gruppo per il quale è stato definito il cursore ('**ABSOLUTE**'), oppure della tupla ***n***-esima rispetto all'ultima tupla estratta da un'istruzione '**FETCH**' precedente.

Le variabili indicate dopo la parola chiave '**INTO**', che in particolare sono precedute da due punti ('**:**'), ricevono ordinatamente il contenuto dei vari attributi della tupla estratta. Naturalmente, le variabili in questione devono appartenere a un linguaggio di programmazione che incorpora l'SQL, dal momento che l'SQL stesso non fornisce questa possibilità.

```
FETCH NEXT FROM Mio_cursore
```

L'esempio mostra l'uso tipico di questa istruzione, dove si legge la tupla successiva (se non ne sono state lette fino a questo punto, si

tratta della prima), dal cursore dichiarato e aperto precedentemente. L'esempio seguente è identico dal punto di vista funzionale.

```
FETCH RELATIVE 1 FROM Mio_cursore
```

I due esempi successivi sono equivalenti e servono a ottenere la tupla precedente.

```
FETCH PRIOR FROM Mio_cursore
```

```
FETCH RELATIVE -1 FROM Mio_cursore
```

74.5.5.3 Chiusura

«

Il cursore, al termine dell'utilizzo, deve essere chiuso:

```
CLOSE cursore
```

Seguendo gli esempi visti in precedenza, per chiudere il cursore '**`Mio_cursore`**' basta l'istruzione seguente:

```
CLOSE Mio_cursore
```

74.6 DCL

«

DCL, ovvero *Data control language*, è il linguaggio usato per il «controllo» delle basi di dati. In questa sezione viene trattato il linguaggio SQL per ciò che riguarda la gestione delle basi di dati, degli utenti, dei privilegi assegnati loro e il controllo delle transazioni.

74.6.1 Gestione delle utenze

«

La gestione degli accessi in una base di dati è molto importante e potenzialmente indipendente dall'eventuale gestione degli utenti del sistema operativo sottostante. Per quanto riguarda i sistemi Unix, il

DBMS può riutilizzare la definizione degli utenti del sistema operativo, farvi riferimento, oppure astrarsi completamente (spesso vale questa ultima ipotesi).

Un DBMS SQL richiede la presenza di almeno un amministratore complessivo, che come tale abbia sempre tutti i privilegi necessari a intervenire come vuole nel DBMS. Il nome simbolico predefinito per questo utente dal linguaggio SQL standard è '**_SYSTEM**'.

Il sistema di definizione e controllo delle utenze è esterno al linguaggio SQL standard; tuttavia, i DBMS principali utilizzano istruzioni abbastanza uniformi per questo scopo.

Per la creazione di un utente si dispone normalmente dell'istruzione '**CREATE USER**', con opzioni che dipendono dalle caratteristiche particolari del DBMS, nella gestione delle utenze. I due modelli sintattici successivi si riferiscono, rispettivamente, a Oracle e a PostgreSQL, ma omettono varie opzioni specifiche e presumono che l'utente debba essere identificato attraverso una parola d'ordine:

```
CREATE USER nome_utente IDENTIFIED BY 'parola_d'ordine'
```

```
CREATE USER nome_utente [WITH PASSWORD 'parola_d'ordine']
```

Nel caso di MySQL, invece di introdurre un'istruzione che non esiste nello standard, si estende quella con cui si concedono i privilegi (descritta in un'altra sezione):

```
GRANT privilegi
    ON risorsa [, ...]
    TO utente
    IDENTIFIED BY 'parola_d'ordine'
    [WITH GRANT OPTION]
```

In tal modo, attribuendo dei privilegi a un utente, se questo non esiste ancora, viene creato contestualmente. Si osservi comunque, che le versioni più recenti di MySQL dispongono di un'istruzione '**CREATE USER**' simile a quella di altri DBMS.

Per l'eliminazione di un utente si dispone normalmente dell'istruzione '**DROP USER**', con opzioni che di solito consentono l'eliminazione contestuale di tutto ciò che appartiene a tale utente:

```
DROP USER nome_utente
```

Per modificare la parola d'ordine di un utente, si dispone normalmente dell'istruzione '**ALTER USER**', con la quale si potrebbero cambiare anche altre opzioni legate ai privilegi generali di cui può disporre tale utente. I due modelli sintattici successivi si riferiscono, rispettivamente, a Oracle e a PostgreSQL, omettendo opzioni che non sono indispensabili:

```
ALTER USER nome_utente IDENTIFIED BY 'parola_d'ordine'
```

```
ALTER USER nome_utente [WITH PASSWORD 'parola_d'ordine']
```

Nel caso di MySQL si usa una forma differente:

```
SET PASSWORD FOR nome_utente = PASSWORD('parola_d'ordine')
```

74.6.2 Gestione delle basi di dati

La creazione e l'eliminazione delle basi di dati è una funzione non considerata dallo standard SQL, anche se è normale che un DBMS consenta la gestione di più basi di dati simultaneamente. Pertanto, i vari DBMS offrono delle istruzioni SQL abbastanza uniformi:

```
CREATE DATABASE nome_base_di_dati
```

Di solito, salvo indicazione diversa derivante da opzioni particolari aggiunte all'istruzione, l'utente che crea la base di dati ne diviene il proprietario, con ogni facoltà sulla stessa, anche quella di eliminarla.

È il caso di osservare che uno dei problemi tecnici da considerare nella creazione di una base di dati sta nel definire la codifica da usare per la memorizzazione delle informazioni testuali. Di solito, questo genere di cose viene definito tramite delle opzioni specifiche, che si aggiungono al modello sintetico e generalizzato mostrato qui.

L'eliminazione di una base di dati richiede generalmente un'istruzione altrettanto semplice:

```
DROP DATABASE nome_base_di_dati
```

La differenza più importante tra i vari DBMS consiste nel modo di comportarsi di fronte a questo comando, quando la base di dati non è vuota. Se esiste un contenuto, la cancellazione potrebbe essere ri-

fiutata, oppure potrebbe essere ammessa se si aggiungono opzioni specifiche che servono a confermarne l'eliminazione. Tuttavia, non si può contare su un controllo di questo genere e la cancellazione di una base di dati richiede sempre la dovuta prudenza.

74.6.3 Gestione dei privilegi standard

«

L'utente che crea una relazione, o un'altra risorsa, è il suo creatore. Su tale risorsa è l'unico utente che possa modificarne la struttura e che possa eliminarla. In pratica è l'unico che possa usare le istruzioni '**`DROP`**' e '**`ALTER`**'. Chi crea una relazione, o un'altra risorsa, può concedere o revocare i privilegi degli altri utenti su di essa.

I privilegi che si possono concedere o revocare su una risorsa sono di vario tipo, espressi attraverso una parola chiave particolare. È bene considerare i casi seguenti:

Privilegio	Descrizione
`SELECT`	rappresenta l'operazione di lettura del valore di un oggetto della risorsa, per esempio dei valori di una tupla da una relazione (in pratica si riferisce all'uso dell'istruzione '**`SELECT`**');
`INSERT`	rappresenta l'azione di inserire un nuovo oggetto nella risorsa, come l'inserimento di una tupla in una relazione;
`UPDATE`	rappresenta l'operazione di aggiornamento del valore di un oggetto della risorsa, per esempio la modifica del contenuto di una tupla di una relazione;
`DELETE`	rappresenta l'eliminazione di un oggetto dalla risorsa, come la cancellazione di una tupla da una relazione;

Privilegio	Descrizione
`ALL PRIVILEGES`	rappresenta simultaneamente tutti i privilegi possibili riferiti a un oggetto.

I privilegi su una relazione, o su un'altra risorsa, vengono concessi attraverso l'istruzione '**GRANT**':

```
GRANT privilegi
    ON risorsa [,…]
    TO utenti
    [WITH GRANT OPTION]
```

Nella maggior parte dei casi, le risorse da controllare coincidono con una relazione. L'esempio seguente permette all'utente '**Pippo**' di leggere il contenuto della relazione '**Movimenti**':

```
GRANT SELECT ON Movimenti TO Pippo
```

L'esempio seguente, concede tutti i privilegi sulla relazione '**Movimenti**' agli utenti '**Pippo**' e '**Arturo**':

```
GRANT ALL PRIVILEGES ON Movimenti TO Pippo, Arturo
```

L'opzione '**WITH GRANT OPTION**' permette agli utenti presi in considerazione di concedere a loro volta tali privilegi ad altri utenti. L'esempio seguente concede all'utente '**Pippo**' di accedere in lettura al contenuto della relazione '**Movimenti**' e gli permette di concedere lo stesso privilegio ad altri:

```
GRANT SELECT ON Movimenti TO Pippo WITH GRANT OPTION
```

I privilegi su una relazione, o un'altra risorsa, vengono revocati attraverso l'istruzione '**REVOKE**':

```
REVOKE privilegi
    ON risorsa [, ...]
    FROM utenti
```

L'esempio seguente toglie all'utente '**Pippo**' il permesso di accedere in lettura al contenuto della relazione '**Movimenti**':

```
REVOKE SELECT ON Movimenti FROM Pippo
```

L'esempio seguente toglie tutti i privilegi sulla relazione '**Movimenti**' agli utenti '**Pippo**' e '**Arturo**':

```
REVOKE ALL PRIVILEGES ON Movimenti FROM Pippo, Arturo
```

74.6.4 Controllo delle transazioni

«

Una transazione SQL, è una sequenza di istruzioni che rappresenta un corpo unico dal punto di vista della memorizzazione effettiva dei dati. In altre parole, secondo l'SQL, la registrazione delle modifiche apportate alla base di dati avviene in modo asincrono, raggruppando assieme l'effetto di gruppi di istruzioni determinati.

Una transazione inizia nel momento in cui l'interprete SQL incontra, generalmente, l'istruzione '**START TRANSACTION**', terminando con l'istruzione '**COMMIT**', oppure '**ROLLBACK**': nel primo caso si conferma la transazione che viene memorizzata regolarmente, mentre nel secondo si richiede di annullare le modifiche apportate dalla transazione.

```
START TRANSACTION
```

```
COMMIT [WORK]
```

```
ROLLBACK [WORK]
```

Stando così le cose, si intende la necessità di utilizzare regolarmente l'istruzione '**COMMIT**' per memorizzare i dati quando non esiste più la necessità di annullare le modifiche.

```
START TRANSACTION
…
COMMIT

INSERT INTO Indirizzi
    VALUES (
        01,
        'Pallino',
        'Pinco',
        'Via Biglie 1',
        '0222,222222'
    )

COMMIT
```

L'esempio mostra un uso intensivo dell'istruzione '**COMMIT**', dove dopo l'inserimento di una tupla nella relazione '**Indirizzi**', viene confermata immediatamente la transazione.

```
START TRANSACTION
…
COMMIT

INSERT INTO Indirizzi
    VALUES (
        01,
        'Pallino',
        'Pinco',
        'Via Biglie 1',
        '0222,222222'
    )

ROLLBACK
```

Questo esempio mostra un ripensamento (per qualche motivo). Dopo l'inserimento di una tupla nella relazione '**Indirizzi**', viene annullata la transazione, riportando la relazione allo stato precedente.

74.7 Riferimenti

«

- Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Riccardo Torlone, *Basi di dati, concetti, linguaggi e architetture*, McGraw-Hill
- James Hoffmann, *Introduction to Structured Query Language*, *http://www.highcroft.com/highcroft/sql_intro.pdf*
- *JCC's SQL Std. Page*, *http://www.jcc.com/sql.htm*
- *SQL Reference Page*, *http://www.contrib.andrew.cmu.edu/~shadow/sql.html*

- *SQL92 BNF*, http://www.contrib.andrew.cmu.edu/~shadow/sql/sql2bnf.aug92.txt
- *ISO/IEC 9075:1992, Database Language SQL*, http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt
- *BNF Grammar for ISO/IEC 9075:1999 - Database Language SQL (SQL-99)*, http://savage.net.au/SQL/sql-99.bnf , http://savage.net.au/SQL/sql-99.bnf.html
- *PostgreSQL*, http://www.postgresql.org/
- *MySQL*, http://www.mysql.com/
- *MaxDB*, http://www.mysql.com/sap/
- *Firebird*, http://firebird.sourceforge.net/

75.1 Struttura e preparazione 1950
75.1.1 Struttura dei dati nel file system 1951
75.1.2 Amministratore 1952
75.1.3 Creazione del sistema di basi di dati 1954
75.1.4 Avvio del servizio 1957
75.1.5 Configurazione del DBMS 1964
75.1.6 Accesso e autenticazione 1966
75.2 Gestione del DBMS 1972
75.2.1 Accesso a una base di dati 1973
75.2.2 Organizzazione degli utenti 1982
75.2.3 Creazione ed eliminazione delle basi di dati 1985
75.2.4 La base di dati amministrativa 1987
75.2.5 Manutenzione delle basi di dati 1989
75.2.6 Copie di sicurezza 1991
75.2.7 Importazione ed esportazione dei dati 1997
75.3 Il linguaggio 2000
75.3.1 Prima di iniziare 2001
75.3.2 Tipi di dati e rappresentazione 2002
75.3.3 Funzioni 2008
75.3.4 Esempi comuni 2010
75.3.5 Controllo delle transazioni 2023
75.3.6 Cursori 2024

75.3.7 Impostazione dell'ora locale 2026
75.4 Accesso attraverso PgAccess 2027
75.4.1 Accesso alla base di dati 2029
75.4.2 Gli «oggetti» secondo PgAccess 2031
75.4.3 Relazioni 2033
75.4.4 Interrogazioni e viste 2035
75.4.5 Stampe 2039
75.5 Accesso attraverso WWW-SQL 2041
75.5.1 Principio di funzionamento 2041
75.5.2 Preparazione delle basi di dati e accesso 2042
75.5.3 Linguaggio di WWW-SQL 2043
75.5.4 Istruzioni 2055
75.6 Riferimenti 2065

`pg_database` 1987 `pg_shadow` 1987 `pg_user` 1987 `psql` 1973 `$PGHOST` 1973 `$PGPORT` 1973

75.1 Struttura e preparazione

«

PostgreSQL[1] è un DBMS (*Data base management system*) relazionale, esteso agli oggetti. In questo capitolo si vuole introdurre al suo utilizzo e accennare alla sua struttura, senza affrontare le particolarità del linguaggio di interrogazione. Il nome lascia intendere che si tratti di un DBMS in grado di comprendere le istruzioni SQL.

75.1.1 Struttura dei dati nel file system

PostgreSQL, a parte i programmi binari, gli script e la documentazione, colloca i file di gestione delle basi di dati a partire da una certa directory, che nella documentazione originale viene definita '**PGDATA**'. Questo è il nome di una variabile di ambiente che può essere utilizzata per informare i vari programmi di PostgreSQL della sua collocazione; tuttavia, di solito questo meccanismo della variabile di ambiente non viene utilizzato, specificando tale directory in fase di compilazione dei sorgenti oppure avviando i programmi con opzioni appropriate.

Tutti i programmi che compongono il sistema di PostgreSQL, che hanno la necessità di sapere dove si trovano i dati, oltre al meccanismo della variabile di ambiente '**PGDATA**' permettono di indicare tale directory attraverso un'opzione della riga di comando. I programmi più importanti riconoscono l'opzione '**-D**'. Come si può intuire, l'utilizzo di questa opzione, o di un'altra equivalente per gli altri programmi, fa in modo che l'indicazione della variabile '**PGDATA**' non abbia effetto.

Questa directory è contenuta solitamente nella directory iniziale dell'utente di sistema per l'amministrazione di PostgreSQL, che dovrebbe essere '**postgres**'. La directory iniziale dell'utente '**postgres**' (ovvero '`~postgres/`') è normalmente '`/var/lib/postgres/`'; la directory usata normalmente per collocarvi le basi di dati è normalmente '`~postgres/data/`', ovvero '`/var/lib/postgres/data/`'. Di norma, tutto ciò che si trova a partire da '`~postgres/`' appartiene all'utente '**postgres**', anche se

i permessi per il gruppo e gli altri utenti variano a seconda della circostanza.

Inizialmente, la directory che costituisce l'inizio delle basi di dati (‘`~postgres/data/`’) dovrebbe contenere dei file di configurazione, una basi di dati amministrativa (trasparente) e una base di dati da usare come modello per la produzione di altre (‘**`template1`**’) o semplicemente per accedere al DBMS quando non se ne può indicare un'altra. Naturalmente, se per qualche ragione si utilizza l'utenza ‘**`postgres`**’ in modo normale, nella sua directory personale (‘`~postgres/`’) potrebbero apparire dei file che riguardano la personalizzazione di questo utente (‘`.profile`’, ‘`.bash_history`’, o altre cose simili, in funzione dei programmi che si utilizzano).

75.1.2 Amministratore

L'amministratore dei servizi offerti dal DBMS PostgreSQL potrebbe essere una persona diversa dall'amministratore del sistema operativo (l'utente ‘**`root`**’) e corrisponde di solito all'utente ‘**`postgres`**’. In condizioni normali, tale utente del DBMS viene riconosciuto implicitamente da PostgreSQL, purché acceda localmente utilizzando un'utenza del sistema operativo con lo stesso nome.

Quando la propria distribuzione GNU è già predisposta per PostgreSQL, l'utente ‘**`postgres`**’ dovrebbe essere stato previsto (non importa il numero UID che gli sia stato abbinato), ma quasi sicuramente la parola d'ordine per l'accesso al sistema operativo dovrebbe essere «impossibile», come nell'esempio seguente:

```
postgres:!:101:101:PostgreSQL Server:/var/lib/postgres:/bin/bash
```

Come si vede, in questo esempio il campo della parola d'ordine è occupato da un punto esclamativo che di fatto impedisce l'accesso

all'utente '**postgres**'.

A questo punto si pongono due alternative, a seconda che si voglia affidare la gestione del DBMS allo stesso utente '**root**' oppure che si voglia incaricare per questo un altro utente. Nel primo caso non occorrono cambiamenti: l'utente '**root**' può diventare '**postgres**' quando vuole con il comando '**su**'.

```
# su postgres [Invio]
```

Nel secondo caso, l'attribuzione di una parola d'ordine all'utente '**postgres**' permetterebbe a una persona diversa di amministrare il DBMS.

```
# passwd postgres [Invio]
```

Di solito, nella sua configurazione iniziale, l'utente '**postgres**' ha la facoltà di accedere localmente al DBMS, senza bisogno di altre forme di autenticazione, a parte il fatto di essere riconosciuto dal sistema operativo proprio con quello stesso nome. Ciò dipende principalmente dalla configurazione contenuta nel file '`pg_hba.conf`', che viene descritto in seguito, all'interno di questo capitolo. Negli esempi che si mostrano qui, si presume proprio che l'utente '**postgres**' del sistema operativo, in quanto tale, sia riconosciuto così anche dal DBMS; se così non fosse, a causa della configurazione, è probabile vedere apparire la richiesta di introdurre una parola d'ordine, riferita però al DBMS.

75.1.3 Creazione del sistema di basi di dati

La prima volta che si installa PostgreSQL, è molto probabile che venga predisposta automaticamente la directory '`~postgres/`'. Se così non fosse, o se per qualche motivo si dovesse intervenire manualmente, si può utilizzare '**initdb**', che però potrebbe risiedere al di fuori dei percorsi normali contenuti nella variabile '**$PATH**'; precisamente potrebbe trattarsi della directory '`/usr/lib/postgresql/bin/`'.

[*percorso*]`initdb` [*opzioni*] [`--pgdata=`*directory* | `-D` *directory*]

Lo schema sintattico mostra in modo molto semplice l'uso di '**initdb**'. Se si definisce correttamente la variabile di ambiente '**PGDATA**', si può fare anche a meno delle opzioni, diversamente diventa necessario dare questa informazione attraverso l'opzione '**-D**'.

Volendo fare tutto da zero, occorre predisporre la directory iniziale in modo che appartenga dell'utente fittizio '**postgres**':

```
# mkdir ~postgres[Invio]

# chown postgres: ~postgres[Invio]
```

Prima di avviare '**initdb**', è bene utilizzare l'identità dell'utente amministratore di PostgreSQL:

```
# su postgres[Invio]
```

Successivamente, si deve avviare '**initdb**' specificando la directory a partire dalla quale si devono articolare i file che costituisco-

no le basi di dati. Come già descritto, la directory in questione è normalmente '`~postgres/data/`':

```
postgres$ /usr/lib/postgresql/bin/initdb ↩
↪         --locale=it_IT.UTF-8 ↩
↪         --encoding=UNICODE ↩
↪         --pgdata=/var/lib/postgres/data [Invio]

  The files belonging to this database system will be owned
  by user "postgres".
  This user must also own the server process.

  The database cluster will be initialized with locale
  it_IT.UTF-8.

  creating directory /var/lib/postgres/data... ok
  creating directory /var/lib/postgres/data/base... ok
  creating directory /var/lib/postgres/data/global... ok
  creating directory /var/lib/postgres/data/pg_xlog... ok
  creating directory /var/lib/postgres/data/pg_clog... ok
  selecting default max_connections... 100
  selecting default shared_buffers... 1000
  creating configuration files... ok
  creating template1 database in
  /var/lib/postgres/data/base/1... ok
  initializing pg_shadow... ok
  enabling unlimited row size for system tables... ok
  initializing pg_depend... ok
  creating system views... ok
  loading pg_description... ok
  creating conversions... ok
  setting privileges on built-in objects... ok
  creating information schema... ok
  vacuuming database template1... ok
  copying template1 to template0... ok
```

```
Success. The database server should be started
automatically.
If not, you can start the database server using:

    /etc/init.d/postgresql start
```

Nell'esempio sono state usate anche due opzioni il cui significato dovrebbe risultare intuitivo.

Tabella 75.3. Alcune opzioni per l'uso di '**initdb**'.

Opzione	Descrizione
`--pgdata=`***directory_pgdata*** `-D` ***directory_pgdata***	Stabilisce la directory iniziale del sistema di basi di dati di PostgreSQL che si vuole creare. Di solito deve corrispondere a '`~postgres/data/`'.
`--locale=`***sigla_locale***	Stabilisce la configurazione locale. Se non viene utilizzata questa opzione si usa il contenuto delle variabili di ambiente '**LANG**' ed eventualmente '**LC_***'.
`--encoding=`***codifica*** `-E` ***codifica***	Stabilisce la codifica della base di dati usata come modello ('**template1**'), diventando di conseguenza la codifica predefinita per le nuove basi di dati. Tra le varie sigle che si possono usare vale la pena di ricordare '**UNICODE**', '**SQL_ASCII**'.

Teoricamente, '**initdb**' fa tutto quello che è necessario fare; in pratica potrebbe non essere così. La prima cosa da considerare sono i file di configurazione, che, seguendo l'esempio mostrato, vengono collocati nella directory '`~postgres/data/`'. Molto probabilmente la propria distribuzione GNU è organizzata per avere i file di configurazione in una directory '`/etc/postgresql/`', o si-

mile. Se le cose stanno così, bisogna provvedere a sostituire i file di configurazione nella directory '`~postgres/data/`' con dei collegamenti simbolici appropriati.

Le distribuzioni GNU possono avere la necessità di passare alcune informazioni, tramite variabili di ambiente, all'utente fittizio '**`postgres`**', cosa che si ottiene con un file '`~postgres/.profile`' appropriato. Se si vuole ricreare la directory '`~postgres/`' da zero, ma si nota la presenza di file di configurazione della shell, è necessario accertarsi del loro contenuto e provvedere di conseguenza nella ricostruzione della directory.

Un'ultima questione importante da sistemare è la directory '`~postgres/dumpall/`', che serve a contenere versioni vecchie degli eseguibili di PostgreSQL, con lo scopo di recuperare i dati dalle versioni vecchie delle basi di dati. Normalmente è sufficiente recuperare la directory già usata in precedenza.

75.1.4 Avvio del servizio

Il DBMS di PostgreSQL si basa su un sistema cliente-servente, in cui, il programma che vuole interagire con una base di dati determinata deve farlo attraverso delle richieste inviate a un servente. In questo modo, il servizio può essere esteso anche attraverso la rete.

L'organizzazione di PostgreSQL prevede la presenza di un demone sempre in ascolto (può trattarsi di un socket di dominio Unix o anche di una porta TCP, che di solito corrisponde al numero 5432). Quando questo riceve una richiesta valida per iniziare una connessione, attiva una copia del servente vero e proprio (*back-end*), a cui affida la connessione con il cliente. Il demone in ascolto per le ri-

chieste di nuove connessioni è '**postmaster**', mentre il servente è '**postgres**'.

Purtroppo, la scelta del nome «postmaster» è un po' infelice, dal momento che potrebbe far pensare all'amministratore del servizio di posta elettronica. Come al solito occorre un po' di attenzione al contesto in cui ci si trova.

Generalmente, il demone '**postmaster**' viene avviato attraverso la procedura di inizializzazione del sistema, in modo indipendente dal supervisore dei servizi di rete. In pratica, di solito si utilizza uno script collocato all'interno di '`/etc/init.d/`', o in un'altra collocazione simile, per l'avvio e l'interruzione del servizio.

Durante il funzionamento del sistema, quando alcuni clienti sono connessi, si può osservare una dipendenza del tipo rappresentato dallo schema seguente:

```
--postmaster-+-postgres
             |-postgres
             `-postgres
```

Il demone '**postmaster**' si occupa di restare in ascolto in attesa di una richiesta di connessione con un servente '**postgres**' (il programma terminale, o *back-end* in questo contesto). Quando riceve questo tipo di richiesta mette in connessione il cliente (programma frontale, o *front-end*) con una nuova copia del servente '**postgres**'.

`postmaster` [***opzioni***]

Per poter compiere il suo lavoro, il demone deve essere a conoscenza di alcune notizie essenziali, tra cui in particolare: la collocazione del programma '**postgres**' (se questo non è in uno dei percorsi della variabile '**PATH**') e la directory da cui si dirama il sistema di file che costituisce l'insieme delle varie basi di dati. Queste notizie possono essere predefinite, nella configurazione usata al momento della compilazione dei sorgenti, oppure possono essere indicate attraverso la riga di comando.

Il demone '**postmaster**' e i processi terminali da lui controllati, gestiscono dei file che compongono le varie basi di dati del sistema. Trattandosi di un sistema di gestione dei dati molto complesso, è bene evitare di inviare il segnale '**SIGKILL**' (9), perché con questo si provoca la conclusione immediata del processo destinatario e di tutti i suoi discendenti, senza permettere una conclusione corretta. Al contrario, gli altri segnali sono accettabili, come per esempio un '**SIGTERM**' che viene dato in modo predefinito quando si utilizza il comando '**kill**'.

Tabella 75.5. Alcune opzioni per l'avvio di '**postmaster**'.

Opzione	Descrizione
`-D` ***directory_dei_dati***	Permette di specificare la directory di inizio della struttura dei dati del DBMS.

Opzione	Descrizione
`-S`	Specifica che il programma deve funzionare in modo «silenzioso», senza emettere alcuna segnalazione, diventando un processo discendente direttamente da quello iniziale (Init), disassociandosi dalla shell e quindi dal terminale da cui è stato avviato. Questa opzione viene utilizzata particolarmente per avviare il programma all'interno della procedura di inizializzazione del sistema, quando non sono necessari dei controlli di funzionamento.
`-b` ***percorso_del_programma_terminale***	Se il programma terminale, ovvero '**`postgres`**', non si trova in uno dei percorsi contenuti nella variabile di ambiente '**`PATH`**', è necessario specificare la sua collocazione (il percorso assoluto) attraverso questa opzione.

Opzione	Descrizione
`-d` [***livello_di_diagnosi***]	Questa opzione permette di attivare la segnalazione di messaggi diagnostici (*debug*), da parte di '**`postmaster`**' e da parte dei programmi terminali, a più livelli di dettaglio: 1, segnala solo il traffico di connessione; 2, o superiore, attiva la segnalazione diagnostica anche nei programmi terminali, oltre ad aggiungere dettagli sul funzionamento di '**`postmaster`**'. Di norma, i messaggi diagnostici vengono emessi attraverso lo standard output da parte di '**`postmaster`**', anche quando si tratta di messaggi provenienti dai programmi terminali. Perché abbia significato l'uso di questa opzione, occorre avviare '**`postmaster`**' **senza l'opzione** '**`-S`**'.
`-i`	Abilita le connessioni TCP/IP. Senza l'indicazione di questa opzione, sono ammissibili solo le connessioni locali attraverso socket di dominio Unix (*Unix domain socket*).

Opzione	Descrizione
`-p` ***porta***	Se viene avviato in modo da accettare le connessioni attraverso la rete (l'opzione '**`-i`**'), specifica una porta di ascolto diversa da quella predefinita (5432).

Segue la descrizione di alcuni esempi.

- `# su postgres -c 'postmaster -S ↩`
  `↪ -D/var/lib/postgres/data'` [*Invio*]

  L'utente '**`root`**', avvia '**`postmaster`**' dopo essersi trasformato temporaneamente nell'utente '**`postgres`**' (attraverso '**`su`**'), facendo in modo che il programma si disassoci dalla shell e dal terminale, diventando un discendente da Init. Attraverso l'opzione '**`-D`**' si specifica la directory di inizio dei file della base di dati.

- `# su postgres -c 'postmaster -i -S ↩`
  `↪ -D/var/lib/postgres/data'` [*Invio*]

  Come nell'esempio precedente, specificando che si vuole consentire, in modo preliminare, l'accesso attraverso la rete.

  Per consentire in pratica l'accesso attraverso la rete, occorre anche intervenire all'interno del file di configurazione '`~postgres/pg_hda.conf`'.

- `# su postgres -c 'nohup postmaster ↩`
  `↪ -D/var/lib/postgres/data ↩`
  `↪ > /var/log/pglog 2>&1 &'` [*Invio*]

L'utente '**root**', avvia '**postmaster**' in modo simile al precedente, dove in particolare viene diretto lo standard output all'interno di un file, per motivi diagnostici. Si osservi l'utilizzo di '**nohup**' per evitare l'interruzione del funzionamento di '**postmaster**' all'uscita del programma '**su**'.

- `# su postgres -c 'nohup postmaster ↩`
  `↪ -D/var/lib/postgres -d 1 ↩`
  `↪ > /var/log/pglog 2>&1 &'` [*Invio*]

Come nell'esempio precedente, con l'attivazione del primo livello diagnostico nei messaggi emessi.

Riquadro 75.6. Controllo diagnostico.

Inizialmente, l'utilizzo di PostgreSQL si può dimostrare poco intuitivo, soprattutto per ciò che riguarda le segnalazioni di errore, spesso troppo poco esplicite. In caso di difficoltà, per permettere di avere una visione un po' più chiara di ciò che accade, sarebbe bene fare in modo che '**postmaster**' produca dei messaggi diagnostici, possibilmente diretti a un file o a una console virtuale inutilizzata.
Per avere una visione immediata di ciò che accade, l'esempio seguente avvia '**postmaster**' in modo manuale e, oltre a conservare le informazioni diagnostiche in un file, le visualizza continuamente attraverso una console virtuale inutilizzata, che in questo caso è l'ottava:

`# su postgres` [*Invio*]

`$ nohup postmaster -D/var/lib/postgres/data -d 1 ↩`
`↪> /var/log/pglog 2>&1 &` [*Invio*]

`$ exit` [*Invio*]

`# nohup tail -f /var/lib/postgres > /dev/tty8 &` [*Invio*]

75.1.5 Configurazione del DBMS

«

Come già accennato, è possibile influenzare il comportamento del servente PostgreSQL attraverso opzioni della riga di comando e variabili di ambiente. Oltre a questi metodi, è possibile intervenire nel file '`~postgres/data/postgresql.conf`', attraverso direttive che assomigliano all'assegnamento di variabili. Il loro significato dovrebbe risultare intuitivo. Viene mostrato un estratto di esempio di questo file:

```
# PostgreSQL configuration file
…
#
# TCP/IP access is allowed by default, but the default
# access given in pg_hba.conf will permit it only from
# localhost, not other machines.
#
tcpip_socket = true
…
#
#       Message display
#
log_connections = true
log_pid = true
log_timestamp = true
…
#
#       Syslog
#
syslog = 2      # range 0-2
…
#
#       Misc
#
```

```
dynamic_library_path = ↩
↪'/usr/share/postgresql:↩
↪/usr/lib/postgresql:/usr/lib/postgresql/lib'
…
#
# How (by default) to present dates to the frontend; the
# user can override this setting for his own session.
# The choices are:
#   Style      Date              Timestamptz
#   ---------------------------------------------------
#   ISO        1999-07-17        1999-07-17 07:09:18+01
#   SQL        17/07/1999        17/07/1999 07:09:19 BST
#   POSTGRES   17-07-1999        Sat 17 Jul 07:09:19 1999 BST
#   GERMAN     17.07.1999        17.07.1999 07:09:19 BST
#
# It is also possible to specify month-day or day-month
# ordering in date input and output.  Americans tend to use
# month-day; Europeans use day-month.  Specify European or
# US.  This is used for interpreting date input, even if the
# output format is ISO.  Separate the two parameters
# by a comma with no spaces
#
datestyle = 'ISO,European'
…
LC_MESSAGES = 'C'
LC_MONETARY = 'C'
LC_NUMERIC = 'C'
LC_TIME = 'C'
```

Si può osservare la direttiva '**`tcpip_socket = true`**', che abilita l'accesso al servente attraverso la rete, ma che richiede di specificare meglio le possibilità di accesso attraverso il file '`~postgres/data/pg_hba.conf`'.

Tabella 75.8. Elenco dei formati di data gestibili con PostgreSQL.

Stile	Descrizione	Esempio
`ISO`	ISO 8601	2012-12-31
`SQL`	Tipo tradizionale	12/31/2012
`POSTGRESQL`	Tipo specifico di PostgreSQL	12-31-2012
`GERMAN`		31.12.2012

Nel caso particolare della distribuzione GNU/Linux Debian, può essere controllato tutto a partire dai file che si trovano nella directory '`/etc/postgresql/`'. In particolare, si trova in questa directory il file '`pg_hba.conf`' e il file '`postgresql.conf`', già descritti in altre sezioni; inoltre, si trova un file aggiuntivo che viene interpretato dallo script della procedura di inizializzazione del sistema che si occupa di avviare e di arrestare il servizio. Si tratta dei file '`/etc/postgresql/postmaster.conf`', attraverso il quale si possono controllare delle piccole cose a cui non si può accedere con il file '`postgresql.conf`', che altrimenti richiederebbero di intervenire attraverso le opzioni della riga di comando del demone relativo.

75.1.6 Accesso e autenticazione

«

L'accesso alle basi di dati viene permesso attraverso un sistema di autenticazione. I sistemi di autenticazione consentiti possono essere diversi e dipendono dalla configurazione di PostgreSQL fatta all'atto della compilazione dei sorgenti.

Il file di configurazione ‘`pg_hba.conf`’ (*Host-based authentication*), che si trova nella directory ‘`~postgres/data/`’, serve per controllare il sistema di autenticazione una volta installato PostgreSQL.

L’autenticazione degli utenti può avvenire in modo incondizionato (‘**`trust`**’), dove ci si fida del nome fornito come utente del DBMS, senza richiedere altro.

L’autenticazione può essere semplicemente disabilitata, nel senso di impedire qualunque accesso incondizionatamente. Questo può servire per impedire l’accesso da parte di un certo gruppo di nodi.

L’accesso può essere controllato attraverso l’abbinamento di una parola d’ordine agli utenti di PostgreSQL.

Inoltre, l’autenticazione può avvenire attraverso un sistema Kerberos, oppure attraverso il protocollo IDENT (sezione 43.3). In questo ultimo caso, ci si fida di quanto riportato dal sistema remoto il quale conferma o meno che la connessione appartenga a quell’utente che si sta connettendo.

Il file ‘`~postgres/data/pg_hba.conf`’ (ma spesso questo è un collegamento simbolico che punta a ‘`/etc/postgresql/pg_hba.conf`’ o a un’altra posizione simile) permette di definire quali nodi possono accedere al servizio DBMS di PostgreSQL, eventualmente stabilendo anche un abbinamento specifico tra basi di dati, utenti e nodi di rete.

Le righe vuote e il testo preceduto dal simbolo ‘**`#`**’ vengono ignorati. I record (cioè le righe contenenti le direttive del file in questione) sono suddivisi in campi separati da spazi o caratteri di tabulazione. Il formato può essere semplificato nei due modelli sintattici seguenti,

tenendo conto che esistono comunque altri casi:

```
local base_di_dati utente_dbms autenticazione_utente [mappa]
```

```
host base_di_dati utente_dbms indirizzo_ip maschera_degli_indirizzi ↩
↪ autenticazione_utente [mappa]
```

Nel primo caso si intendono controllare gli accessi provenienti da programmi clienti avviati nello stesso sistema locale, utilizzando un socket di dominio Unix per il collegamento; nel secondo si fa riferimento ad accessi attraverso la rete (connessioni TCP).

- Il secondo campo del record serve a indicare il nome di una base di dati per la quale autorizzare l'accesso; in alternativa si può usare la parola chiave '**all**', in modo da specificare tutte le basi di dati in una sola volta.
- Il terzo campo del record serve a indicare il nome dell'utente del DBMS da autorizzare; in alternativa si può usare la parola chiave '**all**', in modo da rendere indifferente chi sia l'utente.
- I campi ***indirizzo_ip*** e ***maschera_degli_indirizzi*** rappresentano un gruppo di indirizzi di nodi che hanno diritto di accedere a quella base di dati determinata.
- Il campo ***autenticazione_utente*** rappresenta il tipo di autenticazione attraverso una parola chiave. Le più comuni sono elencate nella tabella 75.9.
- L'ultimo campo dipende dal penultimo. Nel caso di autenticazione '**ident**', si utilizza quasi sempre la parola chiave '**sameuser**'

per indicare a PostgreSQL che i nomi usati dagli utenti nei sistemi remoti da cui possono accedere, coincidono con quelli predisposti per la gestione del DBMS. Nel caso di autenticazione '**`password`**', l'ultimo campo potrebbe rappresentare il nome del file di testo contenente le parole d'ordine.

Tabella 75.9. Parole chiave che possono essere usate nel campo ***autenticazione_utente***.

Autenticazione	Descrizione
`trust`	L'autenticazione non ha luogo e si accetta il nome fornito dall'utente senza alcuna verifica.
`reject`	La connessione viene rifiutata in ogni caso.
`password`	Viene richiesta una parola d'ordine riferita all'utente del DBMS.
`md5` `crypt`	Viene richiesta una parola d'ordine riferita all'utente del DBMS, che però viene trasmessa in modo cifrato. Le due parole chiave si riferiscono a sistemi differenti; si osservi che, di solito, solo uno dei due sistemi può essere utilizzato, perché dipende dal modo in cui sono memorizzate le parole d'ordine. Pertanto, se uno dei due non funziona, si può tentare con l'altro (dopo aver verificato che comunque l'accesso con le parole d'ordine in chiaro funziona regolarmente).

Autenticazione	Descrizione
`ident sameuser` `ident` ***mappa***	L'autenticazione avviene attraverso il sistema operativo locale, oppure con il protocollo IDENT per gli accessi remoti (sezione 43.3). Si usa questa modalità di riconoscimento, prevalentemente per gli accessi locali, ma in tal caso si mette quasi sicuramente anche l'opzione '**`sameuser`**', per fare riferimento allo stesso utente del sistema operativo. Se non si utilizza la parola chiave '**`sameuser`**', al suo posto va messo il nome di una «mappa», da definire in un altro file.
`pam` [***servizio***]	L'autenticazione avviene attraverso il sistema PAM (*Pluggable authentication modules*) del sistema operativo. Se non viene indicato il servizio PAM, si intende '**`postgresql`**'.

Segue la descrizione di alcuni esempi.

- ```
  local   all     all                                     trust
  ```
  Concede a tutti gli utenti di accedere localmente (tramite un socket di dominio Unix), a qualunque base di dati, senza bisogno di alcun riconoscimento (si accetta il nome e basta).

- ```
  host    all     all     127.0.0.1   255.255.255.255  trust
  ```
  Concede a tutti gli utenti di accedere localmente, ma tramite un socket di dominio Internet (l'indirizzo 127.0.0.1 e normalmente quello di ogni nodo, dal punto di vista locale), senza bisogno di alcun riconoscimento.

- ```
  local   all     all                                     ident sameuser
  ```

  Concede a tutti gli utenti di accedere localmente (tramite un socket di dominio Unix), a qualunque base di dati, sulla base del riconoscimento fatto dal sistema operativo (si intende che ci si affida ai privilegi che ha ottenuto il programma usato per accedere).

- ```
  host  all  all  127.0.0.1  255.255.255.255  ident sameuser
  ```

  Concede a tutti gli utenti di accedere localmente, ma tramite un socket di dominio Internet, sulla base del riconoscimento ottenuto tramite l'uso del protocollo di rete IDENT. Questo metodo può essere usato in alternativa a quello dell'esempio precedente, se per qualche ragione il riconoscimento locale (senza rete), non dovesse funzionare.

- ```
  host   gazie   pippo  192.168.0.0   255.255.0.0   password
  ```

  Concede all'utente '**`pippo`**' di accedere alla base di dati '**`gazie`**', da un nodo qualunque tra quelli che hanno indirizzi del tipo 192.168.∗.∗, attraverso l'indicazione di una parola d'ordine, che viene trasmessa in chiaro.

L'esempio seguente rappresenta una configurazione che potrebbe essere considerata «ragionevole», per poter utilizzare l'utente '**`postgres`**', localmente, senza bisogno di fornire una parola d'ordine (come richiesto dagli esempi mostrati in questo capitolo), consentendo agli altri utenti di accedere da una rete locale qualunque (lo si determina in base al fatto che si fa riferimento a indirizzi IPv4 privati), ma in tal caso si richiede un riconoscimento basato su una parola d'ordine:
  ```

```
#
# TYPE DATABASE USER   IP-ADDRESS   IP-MASK      METHOD
#
local  all      postgres                        ident sameuser
#
local  all      all                             password
host   all      all    127.0.0.1    255.0.0.0    password
host   all      all    192.168.0.0  255.255.0.0  password
host   all      all    172.16.0.0   255.240.0.0  password
host   all      all    10.0.0.0     255.0.0.0    password
#
host   all      all    0.0.0.0      0.0.0.0      reject
```

75.2 Gestione del DBMS

«

La gestione di un DBMS richiede di occuparsi di utenze e di basi di dati. PostgreSQL mette a disposizione degli script per facilitare la loro creazione ed eliminazione, ma in generale è meglio avvalersi di istruzioni SQL, anche se non sono standard.

Per poter impartire comandi in forma di istruzioni SQL, occorre collegarsi al DBMS attraverso un programma appropriato (di solito '**psql**'); per poter maneggiare gli utenti e le basi di dati è necessario disporre dei privilegi necessari. Generalmente, le prime volte si compiono queste operazioni in qualità di amministratore, pertanto con l'utenza '**postgres**'.

Per accedere al DBMS, occorre indicare una basi di dati, anche se le funzioni in questione non interagiscono direttamente con questa. Di solito, dato che inizialmente non è disponibile altro, ci si collega alla basi di dati '**template1**'.

Teoricamente, PostgreSQL non distingue tra lettere maiuscole e minuscole quando si tratta di nominare le basi di dati, le relazioni (le tabelle o gli oggetti a seconda della definizione che si preferisce utilizzare) e gli elementi che compongono delle relazioni. Tuttavia, in certi casi si verificano degli errori inspiegabili dovuti alla scelta dei nomi che in generale conviene indicare sempre solo con lettere minuscole.

75.2.1 Accesso a una base di dati

L'accesso a una base di dati avviene attraverso un cliente, ovvero un programma frontale, o *front-end*, secondo la documentazione di PostgreSQL. Questo programma si avvale generalmente della libreria LibPQ. PostgreSQL fornisce un programma cliente standard, '**`psql`**', che si comporta come una sorta di shell tra l'utente e la base di dati stessa.

Il programma cliente tipico, dovrebbe riconoscere le variabili di ambiente '**`PGHOST`**' e '**`PGPORT`**'. La prima serve a stabilire l'indirizzo o il nome a dominio del servente, indicando implicitamente che la connessione avviene attraverso una connessione TCP e non con un socket di dominio Unix; la seconda specifica il numero della porta, ammesso che si voglia utilizzare un numero diverso da 5432. L'uso di queste variabili non è indispensabile, ma serve solo per non dover specificare queste informazioni attraverso opzioni della riga di comando.

Il programma '**`psql`**' permette un utilizzo interattivo attraverso una serie di comandi impartiti dall'utente su una riga di comando; op-

pure può essere avviato in modo da eseguire il contenuto di un file o di un solo comando fornito tra gli argomenti. Per quanto riguarda l'utilizzo interattivo, il modo più semplice per avviarlo è quello che si vede nell'esempio seguente, dove si indica semplicemente il nome della base di dati sulla quale intervenire.

```
$ psql mio_db[Invio]

  Welcome to the POSTGRESQL interactive sql monitor:
    Please read the file COPYRIGHT for copyright terms of
    POSTGRESQL

     type \? for help on slash commands
     type \q to quit
     type \g or terminate with semicolon to execute query
   You are currently connected to the database: mio_db

  mio_db=>_
```

Da questo momento si possono inserire le istruzioni SQL per la base di dati selezionata, in questo caso '**mio_db**', oppure si possono inserire dei comandi specifici di '**psql**'. Questi ultimi si notano perché sono composti da una barra obliqua inversa ('****'), seguita da un carattere.

Il comando interno di '**psql**' più importante è '**\h**' che permette di visualizzare una guida rapida alle istruzioni SQL che possono essere utilizzate.

```
=> \h[Invio]
```

```
type \h <cmd> where <cmd> is one of the following:
    abort                   abort transaction       alter table
    begin                   begin transaction       begin work
    cluster                 close                   commit
…
type \h * for a complete description of all commands
```

Nello stesso modo, il comando '**\?**' fornisce un riepilogo dei comandi interni di '**psql**'.

=> **\?** [*Invio*]

```
  \?            -- help
  \a            -- toggle field-alignment (currenty on)
  \C [<captn>] -- set html3 caption (currently '')
…
```

Tutto ciò che '**psql**' non riesce a interpretare come un suo comando interno viene trattato come un'istruzione SQL. Dal momento che queste istruzioni possono richiedere più righe, è necessario informare '**psql**' della conclusione di queste, per permettergli di analizzarle e inviarle al servente. Queste istruzioni possono essere terminate con un punto e virgola ('**;**'), oppure con il comando '**\g**'.

Si può osservare, utilizzando '**psql**', che l'invito mostrato cambia leggermente a seconda del contesto: inizialmente appare nella forma '**=>**', mentre quando è in corso l'inserimento di un'istruzione SQL non ancora terminata si trasforma in '**->**'. Il comando '**\g**' viene usato prevalentemente in questa situazione.

-> **\g** [*Invio*]

Le istruzioni SQL possono anche essere raccolte in un file di testo normale. In tal caso si può utilizzare il comando '**\i**' per fare

in modo che '**psql**' interpreti il suo contenuto, come nell'esempio seguente, dove il file in questione è '`mio_file.sql`'.

=> **\i mio_file.sql** [*Invio*]

Nel momento in cui si utilizza questa possibilità (quella di scrivere le istruzioni SQL in un file facendo in modo che poi questo venga letto e interpretato), diventa utile il poter annotare dei commenti. Questi sono iniziati da una sequenza di due trattini ('**--**'), come prescrive lo standard, e tutto quello che vi appare dopo viene ignorato.

La conclusione del funzionamento di '**psql**' si ottiene con il comando '**\q**'.

=> **\q** [*Invio*]

Per l'avvio di '**psql**' si può usare la sintassi seguente. L'opzione '**-f**' consente di indicare un file contenente istruzioni SQL da eseguire subito; in alternativa, un file di questo tipo può essere fornito attraverso lo standard input.

```
psql [opzioni] [base_di_dati]
```

```
psql -f file_di_istruzioni [altre_opzioni] [base_di_dati]
```

```
cat file_di_istruzioni | psql [opzioni] [base_di_dati]
```

Il programma '**psql**' può funzionare solo in abbinamento a una base di dati determinata. In questo senso, se non viene indicato il nome di una base di dati nella riga di comando, '**psql**' tenta di uti-

lizzarne una con lo stesso nome dell'utente. Per la precisione, si fa riferimento alla variabile di ambiente '**USER**'.

Questo dettaglio dovrebbe permettere di comprendere il significato della segnalazione di errore che si ottiene se si tenta di avviare '**psql**' senza indicare una base di dati, quando non ne esiste una con lo stesso nome dell'utente.

Tabella 75.19. Alcune opzioni per l'avvio di '**psql**'.

Opzione	Descrizione
`-c` ***istruzione_sql*** `--command` ***istruzione_sql***	Permette di fornire un'istruzione SQL già nella riga di comando, ottenendone il risultato attraverso lo standard output e facendo terminare subito dopo l'esecuzione di '**psql**'. Questa opzione viene usata particolarmente in abbinamento a '**-q**'.
`-d` ***base_di_dati*** `--dbname` ***base_di_dati***	Permette di indicare il nome della base di dati da utilizzare. Può essere utile quando per qualche motivo potrebbe essere ambigua l'indicazione del suo nome come ultimo argomento.
`-f` ***file_di_istruzioni*** `--file` ***file_di_istruzioni***	Permette di fornire a '**psql**' un file da interpretare contenente le istruzioni SQL (oltre agli eventuali comandi specifici di '**psql**'), senza avviare così una sessione di lavoro interattiva.

Opzione	Descrizione
`-h` ***nodo*** `--host` ***nodo***	Permette di specificare il nodo a cui connettersi per l'interrogazione del servente PostgreSQL.
`-H` `--html`	Fa in modo che l'emissione in forma tabellare avvenga utilizzando il formato HTML. In pratica, ciò è utile per costruire un risultato da leggere attraverso un navigatore ipertestuale.
`-o` ***file_output*** `--output` ***file_output***	Fa in modo che tutto l'output venga inviato nel file specificato dall'argomento.
`-p` ***porta*** `--port` ***porta***	Nel caso in cui '**`postmaster`**' sia in ascolto su una porta TCP diversa dal numero 5432 (corrispondente al valore predefinito), si può specificare con questa opzione il numero corretto da utilizzare.
`-q` `--quiet`	Fa sì che '**`psql`**' funzioni in modo «silenzioso», limitandosi all'emissione pura e semplice di quanto generato dalle istruzioni impartite. Questa opzione è utile quando si utilizza '**`psql`**' all'interno di script che devono occuparsi di rielaborare il risultato ottenuto.
`-t` `--tuple-only`	Disattiva l'emissione dei nomi degli attributi. Questa opzione viene utilizzata particolarmente in abbinamento con '**`-c`**' o '**`-q`**'.

Opzione	Descrizione
`-T` ***opzioni_tabelle_html*** `--table-attr` ***opzioni_tabelle_html***	Questa opzione viene utilizzata in abbinamento con '**`-H`**', per definire le opzioni HTML delle tabelle che si generano. In pratica, si tratta di ciò che può essere inserito all'interno del marcatore di apertura della tabella: '**`<TABLE ...>`**'.
`-U` ***utente*** `--username` ***utente***	Consente di specificare il nome dell'utente del DBMS.
`-W` `--password`	Forza '**`psql`**' a richiedere una parola d'ordine, in ogni caso.

Tabella 75.20. Alcuni comandi che '**`psql`**' riconosce durante il funzionamento interattivo.

Comando	Descrizione
`\h` [***comando***]	L'opzione '**`\h`**' usata da sola, elenca le istruzioni SQL che possono essere utilizzate. Se viene indicato il nome di una di queste, viene mostrata in breve la sintassi relativa.
`\?`	Elenca i comandi interni di '**`psql`**', cioè quelli che iniziano con una barra obliqua inversa ('**`\`**').

Comando	Descrizione
`\l`	Elenca tutte le basi di dati presenti nel servente. Ciò che si ottiene è una tabella contenente rispettivamente: i nomi delle basi di dati, i numeri di identificazione dei rispettivi amministratori (gli utenti che li hanno creati) e il nome della directory in cui sono collocati fisicamente.
`\connect` ***base_di_dati*** ↩ ↪ [***nome_utente***]	Chiude la connessione con la base di dati in uso precedentemente e tenta di accedere a quella indicata. Se il sistema di autenticazione lo consente, si può specificare anche il nome dell'utente con cui si intende operare sulla nuova base di dati. Generalmente, ciò dovrebbe essere impedito. Se si utilizza un'autenticazione basata sul file '`pg_hba.conf`', l'autenticazione di tipo '**`trust`**' consente questo cambiamento di identificazione, altrimenti, il tipo '**`ident`**' lo impedisce.
`\d` [***relazione***]	L'opzione '**`\d`**' usata da sola, elenca le relazioni contenute nella base di dati, altrimenti, se viene indicato il nome di una di queste relazioni, si ottiene l'elenco degli attributi. Se si utilizza il comando '**`\d *`**', si ottiene l'elenco di tutte le relazioni con le informazioni su tutti gli attributi rispettivi.
`\i` ***file***	Con questa opzione si fa in modo che '**`psql`**' esegua di seguito tutte le istruzioni contenute nel file indicato come argomento.

Comando	Descrizione
`\q`	Termina il funzionamento di '**psql**'.

Segue la descrizione di alcuni esempi.

- `$` **psql mio_db** [*Invio*]

  Cerca di connettersi con la base di dati '**mio_db**' nel nodo locale, riferendosi alla stessa utenza riconosciuta dal sistema operativo, utilizzando il meccanismo del socket di dominio Unix.

- `$` **psql -d mio_db** [*Invio*]

  Esattamente come nell'esempio precedente, con l'uso dell'opzione '**-d**' che serve a evitare ambiguità sul fatto che '**mio_db**' sia il nome della base di dati.

- `$` **psql -U tizio -d mio_db** [*Invio*]

  Come nell'esempio precedente, ma specificando che si intende accedere in qualità di utente '**tizio**'.

- `$` **psql -U tizio -W -d mio_db** [*Invio*]

  Come nell'esempio precedente, ma forzando in ogni caso la richiesta di inserimento di una parola d'ordine.

- `$` **psql -U tizio -W -h dinkel.brot.dg -d mio_db** [*Invio*]

  Come nell'esempio precedente, ma questa volta l'accesso viene fatto a una base di dati con lo stesso nome presso il nodo *dinkel.brot.dg*.

- `$` **psql -U tizio -W -f istruzioni.sql -d mio_db** [*Invio*]

Cerca di connettersi con la base di dati '**mio_db**' nel nodo locale, utilizzando il meccanismo del socket di dominio Unix, quindi esegue le istruzioni contenute nel file '`istruzioni.sql`'.

- `$` **`psql -U tizio -W -d mio_db < istruzioni.sql`**[*Invio*]

  Come nell'esempio precedente, ricevendo il contenuto del file '`istruzioni.sql`' dallo standard input.

75.2.1.1 Variabile di ambiente «PAGER»

Il programma '**psql**' è sensibile alla presenza o meno della variabile di ambiente '**PAGER**'. Se questa esiste e non è vuota, '**psql**' utilizza il programma indicato al suo interno per controllare l'emissione dell'output generato. Per esempio, se contiene '**less**', come si vede nell'esempio seguente che fa riferimento a una shell POSIX o compatibile con quella di Bourne, si fa in modo che l'output troppo lungo venga controllato da Less:

```
PAGER=less
export PAGER
```

Per eliminare l'impostazione di questa variabile, in modo da ritornare allo stato predefinito, basta annullare il contenuto della variabile nel modo seguente:

```
PAGER=
export PAGER
```

75.2.2 Organizzazione degli utenti

La creazione di un utente per il DBMS si ottiene con l'istruzione '**CREATE USER**', che nel modello seguente appare in modo semplificato:

```
CREATE USER nome_utente
    [WITH [PASSWORD 'parola_d'ordine' ]
        [CREATEDB|NOCREATEDB|CREATEUSER|NOCREATEUSER] ]
```

Si comprende intuitivamente il significato delle parole chiave delle opzioni finali, con le quali è possibile concedere o negare i privilegi di creare o eliminare delle basi di dati e di creare o eliminare degli utenti. Si osservi che la parola d'ordine va indicata esattamente tra apici singoli. Se si omettono le opzioni finali, i privilegi relativi vengono negati, come se fossero state specificate implicitamente le parole chiave '**NOCREATEDB**' e '**NOCREATEUSER**'.

L'uso della parola chiave '**CREATEUSER**' nella creazione o nella modifica di un'utenza, concede a questa la facoltà di creare o eliminare delle utenze, senza limitazioni, oltre che di creare ed eliminare delle basi di dati. In altri termini, dà all'utente il ruolo di amministrazione del DBMS, con tutti i poteri necessari.

L'esempio seguente mostra i passaggi per la creazione, presso il DBMS locale, dell'utente '**tizio**' (con una parola d'ordine di esempio) a cui viene concesso di creare delle basi di dati, ma non di gestire delle utenze:

```
# su postgres [Invio]

postgres$ psql template1 [Invio]

template1=# CREATE USER tizio WITH PASSWORD 'segreta' ↩
↪           CREATEDB NOCREATEUSER; [Invio]
```

```
  CREATE USER

template1=# \q[Invio]

postgres$ exit[Invio]
```

Così come è stato creata, le caratteristiche di un'utenza possono essere modificate con l'istruzione '**ALTER USER**', per esempio per modificare la parola d'ordine:

```
ALTER USER nome_utente
    [WITH [PASSWORD 'parola_d'ordine' ]
          [CREATEDB|NOCREATEDB]
          [CREATEUSER|NOCREATEUSER] ]
```

Logicamente, se si tratta di modificare la parola d'ordine, può essere lo stesso utente che esegue questa istruzione; altrimenti, per cambiare i privilegi, è necessario che intervenga un utente che ha maggiori facoltà. Nell'esempio seguente, l'utente '**tizio**' creato in quello precedente, modifica la sua parola d'ordine; si osservi che la scelta della base di dati '**template1**' è puramente casuale:

```
$ psql --username tizio template1[Invio]

Password: digitazione_all'oscuro [Invio]

template1=> ALTER USER tizio ↩
↪                       WITH PASSWORD 'segretissima'; [Invio]

  ALTER USER

template1=> \q[Invio]
```

L'eliminazione di un'utenza avviene con un'istruzione molto semplice, senza opzioni particolari:

```
DROP USER nome_utente
```

L'esempio seguente elimina l'utenza '**tizio**' e l'operazione viene svolta dall'utente '**postgres**':

```
# su postgres[Invio]

postgres$ psql template1[Invio]

template1=# DROP USER tizio;[Invio]

  DROP USER

template1=# \q[Invio]

postgres$ exit[Invio]
```

75.2.3 Creazione ed eliminazione delle basi di dati

La creazione di una base di dati è consentita agli amministratori e agli utenti che hanno ottenuto questo privilegio. La base di dati può essere creata per sé, oppure per farla gestire da un altro utente.

```
CREATE DATABASE nome_base_di_dati
    [[WITH] [OWNER [=] utente_proprietario]
            [ENCODING [=] 'codifica']]
```

Il modello sintattico mostrato omette alcune opzioni, di utilizzo meno frequente. In particolare, sarebbe possibile specificare il modello di riferimento per la creazione della base di dati, ma in modo prede-

finito viene utilizzata la base di dati '**template1**' per crearne una di nuova.

Nell'esempio seguente, l'utente '**tizio**' crea la base di dati '**mia_db**', specificando espressamente la codifica; inizialmente accede facendo riferimento alla base di dati '**template1**':

```
$ psql --username tizio template1[Invio]

Password: digitazione_all'oscuro [Invio]

template1=> CREATE DATABASE mia_db ENCODING 'UNICODE';[Invio]

  CREATE DATABASE

template1=> \q[Invio]
```

L'eliminazione di una base di dati si ottiene con l'uso di un'istruzione molto semplice:

```
DROP DATABASE nome_base_di_dati
```

Questa istruzione può essere usata da un amministratore, oppure dall'utente che ne è proprietario. Nell'esempio seguente, l'utente '**tizio**' elimina la sua base di dati '**mia_db**'; per farlo, accede facendo riferimento a un'altra (la solita '**template1**'):

```
$ psql --username tizio template1[Invio]

Password: digitazione_all'oscuro [Invio]

template1=> DROP DATABASE mia_db;[Invio]

  DROP DATABASE

template1=> \q[Invio]
```

75.2.4 La base di dati amministrativa

PostgreSQL memorizza le informazioni sugli utenti e sulle basi di dati all'interno di una sorta di base di dati amministrativa, senza nome. Alle relazioni di questa base di dati trasparente, si accede da qualunque posizione; in pratica, le relazioni sono accessibili quando si apre la base di dati '**template1**', o qualunque altra, ma ovviamente, solo l'amministratore del DBMS ha la facoltà di modificarle direttamente.

Come conseguenza del fatto che le relazioni della base di dati amministrativa sono accessibili da qualunque posizione, si comprende che i nomi di queste relazioni non si possono utilizzare per la costruzione di nuove.

La documentazione originale di PostgreSQL individua queste relazioni, definendole «cataloghi». In questo documento, si preferisce indicarle come relazioni o tabelle della base di dati amministrativa.

Le relazioni più importanti della base di dati amministrativa sono '**pg_user**', '**pg_shadow**' e '**pg_database**'. Vale la pena di osservare il loro contenuto.

```
postgres:~$ psql -d template1 [Invio]
```

La relazione '**pg_user**' è in realtà una vista del catalogo '**pg_shadow**', che contiene le informazioni sugli utenti di PostgreSQL. La figura 75.28 mostra un esempio di come potrebbe essere composta. La consultazione della relazione si ottiene con l'istruzione SQL seguente:

```
template1=> SELECT * FROM pg_user; [Invio]
```

Figura 75.28. Esempio del contenuto di '**pg_user**'.

```
 usename  |usesysid|usecreatedb|usesuper|usecatupd| passwd |valuntil|useconfig
----------+--------+-----------+--------+---------+--------+--------+---------
postgres  |       1|t          |t       |t        |********|        |
pgnanouser|     100|t          |f       |f        |********|        |
tizio     |    1001|t          |t       |t        |********|        |
```

Si può osservare che l'utente '**postgres**' ha tutti gli attributi booleani attivi ('**usecreatedb**', '**usesuper**', '**usecatupd**') e questo per permettergli di compiere tutte le operazioni all'interno delle basi di dati. In particolare, l'attributo '**usecreatedb**' permette all'utente di creare una base di dati e '**usesuper**' permette di aggiungere utenti. In effetti, osservando l'esempio della figura, l'utente '**tizio**' ha praticamente gli stessi privilegi dell'amministratore '**postgres**'.

La relazione '**pg_shadow**' è il contenitore delle informazioni sugli utenti, a cui si accede normalmente tramite la vista '**pg_user**'. Il suo scopo è quello di conservare in un file più sicuro, in quanto non accessibile agli utenti comuni, i dati delle parole d'ordine degli utenti che intendono usare le forme di autenticazione basate su queste. L'esempio della figura 75.29 mostra gli stessi utenti a cui non viene abbinata alcuna parola d'ordine (probabilmente perché accedono localmente e vengono identificati dal sistema operativo). La consultazione della relazione si ottiene con l'istruzione SQL seguente:

```
template1=> SELECT * FROM pg_shadow; [Invio]
```

Figura 75.29. Esempio del contenuto di '**pg_shadow**'.

```
 usename   | usesysid | usecreatedb | usesuper | usecatupd | passwd | valuntil | useconfig
-----------+----------+-------------+----------+-----------+--------+----------+-----------
 postgres  |        1 | t           | t        | t         |        |          |
 pgnanouser|      100 | t           | f        | f         |        |          |
 tizio     |     1001 | t           | t        | t         |        |          |
```

La relazione ‘**pg_database**’ contiene le informazioni sulle basi di dati esistenti. La figura 75.30 mostra un esempio di come potrebbe essere composta. La consultazione della relazione si ottiene con l’istruzione SQL:

```
template1=> SELECT * FROM pg_database; [Invio]
```

Figura 75.30. Esempio del contenuto di ‘**pg_database**’, diviso in due parti per motivi tipografici.

```
  datname  | datdba | encoding | datistemplate | datallowconn | datlastsysoid |
-----------+--------+----------+---------------+--------------+---------------+--
 nanodb    |    100 |        6 | f             | t            |         17140 |
 template1 |      1 |        6 | t             | t            |         17140 |
 template0 |      1 |        6 | t             | f            |         17140 |

      | datvacuumxid | datfrozenxid | datpath | datconfig |          datacl
     -+--------------+--------------+---------+-----------+--------------------------
      |         8159 |   3221233632 |         |           |
      |         8271 |   3221233744 |         |           | {postgres=C*T*/postgres}
      |          464 |          464 |         |           | {postgres=C*T*/postgres}
```

Il primo attributo rappresenta il nome della base di dati, il secondo riporta il numero di identificazione dell’utente che rappresenta il suo DBA, cioè colui che l’ha creata o che comunque deve amministrarla. Per esempio, si può osservare che la base di dati ‘**nanodb**’ è stata creata dall’utente identificato dal numero 100, che da quanto riportato in ‘**pg_user**’ è ‘**pgnanouser**’.

75.2.5 Manutenzione delle basi di dati

Un problema comune dei DBMS è quello della riorganizzazione periodica dei dati, in modo da semplificare e accelerare le elaborazioni successive. Nei sistemi più semplici si parla a volte di «ricostruzione indici», o di qualcosa del genere. Nel caso di PostgreSQL, si utilizza un comando specifico che è estraneo all’SQL standard: ‘**VACUUM**’.

```
VACUUM [altre_opzioni] [VERBOSE] [nome_relazione]
```

```
VACUUM [altre_opzioni] [VERBOSE] ANALYZE ↩
↪      [nome_relazione [(attributo_1 [, ... attributo_n])]]
```

L'operazione di pulizia si riferisce alla base di dati aperta in quel momento. L'opzione '**VERBOSE**' permette di ottenere i dettagli sull'esecuzione dell'operazione; '**ANALYZE**' serve invece per indicare specificatamente una relazione, o addirittura solo alcuni attributi (le colonne delle tabelle) una relazione e avere informazioni su questi. Eventualmente, sono disponibili altre opzioni per ottenere una riorganizzazione dei dati più importante.

Anche se non si tratta di un comando SQL standard, per PostgreSQL è importante che venga eseguita periodicamente una ripulitura con il comando '**VACUUM**', eventualmente attraverso uno script simile a quello seguente, da avviare per mezzo del sistema Cron:

```
#!/bin/sh
su postgres -c "psql $1 -c 'VACUUM'"
```

In pratica, richiamando questo script con i privilegi dell'utente '**root**', indicando come argomento il nome della base di dati (viene inserito al posto di '**$1**' dalla shell), si ottiene di avviare il comando '**VACUUM**' attraverso '**psql**'.

Per riuscire a fare il lavoro in serie per tutte le basi di dati, si potrebbe scrivere uno script più complesso, come quello seguente. In questo caso, lo script deve essere avviato con i privilegi dell'utente '**postgres**'.

```
#!/bin/sh
#
BASI_DATI=`psql template1 -t -c "SELECT datname from pg_database"`
#
echo "Procedimento di ripulitura e sistemazione delle basi di dati"
echo "di PostgreSQL."
echo "Se l'operazione dovesse essere interrotta accidentalmente,"
echo "potrebbe essere necessaria l'eliminazione del file pg_vlock"
echo "contenuto nella directory della base di dati relativa."
#
for BASE_DATI in $BASI_DATI
do
    printf "$BASE_DATI: "
    psql $BASE_DATI -c "VACUUM"
done
```

In breve, si utilizza la prima volta '**psql**' in modo da aprire la base di dati '**template1**' (quella usata come modello, che si ha la certezza di trovare sempre), accedendo alla relazione '**pg_database**', che fa parte della base di dati amministrativa, per leggere l'attributo contenente i nomi delle basi di dati. In particolare, l'opzione '**-t**' serve a evitare di inserire il nome dell'attributo stesso. L'elenco che si ottiene viene inserito nella variabile di ambiente '**BASI_DATI**', che in seguito viene scandita da un ciclo '**for**', all'interno del quale si utilizza '**psql**' per ripulire ogni singola base di dati.

75.2.6 Copie di sicurezza

Prima di poter pensare a copiare o a spostare una base di dati occorre avere chiaro in mente che si tratta di file «binari» (nel senso che non si tratta di file di testo), contenenti informazioni collegate l'una all'altra in qualche modo più o meno oscuro. Queste informazioni possono a volte essere espresse anche in forma numerica; in tal caso dipende dall'architettura in cui sono state create. Ciò implica due

cose fondamentali: la copia deve essere fatta in modo che non si perdano dei pezzi per la strada; lo spostamento dei dati in forma binaria, in un'altra architettura, non è ammissibile.

La copia di sicurezza binaria, di tutto ciò che serve a PostgreSQL per la gestione delle sue basi di dati, si ottiene semplicemente archiviando quanto contenuto a partire da '`~postgres/`', così come si può comprendere intuitivamente. Ciò che conta è che il ripristino dei dati avvenga nello stesso contesto (architettura, sistema operativo, librerie, versione di PostgreSQL e configurazione).

Per una copia di sicurezza più «sicura», è necessario archiviare i dati in modo indipendente da tutto. Si ottiene questo generando un file di testo, contenente istruzioni SQL con le quali ricostruire poi una sola base di dati o anche tutte assieme. Per questo vengono in aiuto due programmi di PostgreSQL: '**`pg_dump`**' e '**`pg_dumpall`**'.

Non sempre il procedimento di trasferimento dei dati in forma di comandi SQL può essere portato a termine con successo. Può succedere che delle relazioni troppo complesse o con dati troppo grandi, non siano tradotte correttamente nella fase di archiviazione. Questo problema deve essere preso in considerazione già nel momento della progettazione di una base di dati, avendo cura di verificare, sperimentandolo, che il procedimento di scarico e recupero dei dati possa funzionare.

Lo scarico di una sola base di dati si ottiene attraverso il programma '**`pg_dump`**', che, eventualmente, potrebbe risiedere al di fuori dei percorsi normali contenuti nella variabile '**`$PATH`**' e potrebbe trovarsi nella directory '`/usr/lib/postgresql/bin/`':

```
pg_dump [opzioni] base_di_dati
```

Se non si indicano delle opzioni e ci si limita a specificare la base di dati su cui intervenire, si ottiene il risultato attraverso lo standard output, composto in pratica dai comandi necessari a '**psql**' per ricostruire le relazioni che compongono la base di dati (la base di dati stessa deve essere ricreata manualmente). Tanto per chiarire subito il senso della cosa, se si utilizza '**pg_dump**' nel modo seguente, si ottiene il file di testo '`mio_db.dump`':

```
$ pg_dump mio_db > mio_db.dump[Invio]
```

Questo file va verificato, ricercando la presenza eventuale di segnalazioni di errore che vengono generate in presenza di dati che non possono essere riprodotti fedelmente; eventualmente, il file può anche essere modificato se si conosce la sintassi dei comandi che vengono inseriti in questo script. Per fare in modo che le relazioni della base di dati vengano ricreate e caricate, si può utilizzare '**psql**' nel modo seguente:

```
$ psql -e mio_db < mio_db.dump[Invio]
```

Tabella 75.33. Alcune opzioni che possono essere usate con '**pg_dump**'.

Autenticazione	Descrizione
`-d` `--inserts`	In condizioni normali, '**pg_dump**' salva i dati delle relazioni (le tabelle secondo l'SQL) in una forma compatibile con il comando '**COPY**', che però non è compatibile con lo standard SQL. Con l'opzione '**-d**', utilizza il comando '**INSERT**' tradizionale.

Autenticazione	Descrizione
`-D` `--column-inserts`	Come con l'opzione '**`-d`**', con l'aggiunta dell'indicazione degli attributi (le colonne secondo l'SQL) in cui vanno inseriti i dati. In pratica, questa opzione permette di generare uno script più preciso e dettagliato.
`-f` ***file*** `--file=`***file***	Permette di definire un file diverso dallo standard output, che si vuole generare con il risultato dell'elaborazione di '**`pg_dump`**'.
`-h` ***nodo*** `--host=`***nodo***	Permette di specificare il nodo a cui connettersi per l'interrogazione del servente PostgreSQL. In pratica, se l'accesso è consentito, è possibile scaricare una base di dati gestita presso un nodo remoto.
`-p` ***porta*** `--port=`***porta***	Nel caso in cui '**`postmaster`**' sia in ascolto su una porta TCP diversa dal numero 5432 (corrispondente al valore predefinito), si può specificare con questa opzione il numero corretto da utilizzare.
`-s` `--schema-only`	Scarica soltanto la struttura delle relazioni, senza occuparsi del loro contenuto. In pratica, serve per poter riprodurre le relazioni vuote.
`-t` ***nome_relazione*** `--table=`***nome_relazione***	Utilizzando questa opzione, indicando il nome di una relazione, si ottiene lo scarico solo di quella.
`-U` ***nome***	Specifica con quale nominativo utente identificarsi per eseguire l'operazione.

Autenticazione	Descrizione
`-W`	Forza la richiesta di inserire una parola d'ordine, che comunque dovrebbe essere chiesta automaticamente se il DBMS la richiede.

Per copiare o trasferire tutte le basi di dati del sistema di PostgreSQL, si può utilizzare '**pg_dumpall**', che, eventualmente, potrebbe risiedere al di fuori dei percorsi normali contenuti nella variabile '**$PATH**' e potrebbe trovarsi nella directory '`/usr/lib/postgresql/bin/`':

[*percorso*]`pg_dumpall` [*opzioni*]

Il programma '**pg_dumpall**' provvede a scaricare tutte le basi di dati, assieme alle informazioni necessarie per ricreare il catalogo '**pg_shadow**' (la vista '**pg_user**' si ottiene di conseguenza). Come si può intuire, si deve utilizzare '**pg_dumpall**' con i privilegi dell'amministratore del DBMS (di solito l'utente '**postgres**').

`postgres$` **pg_dumpall > basi_dati.dump**[*Invio*]

L'esempio mostra il modo più semplice di utilizzare '**pg_dumpall**' per scaricare tutte le basi di dati in un file unico. In questo caso, si ottiene il file di testo '`basi_dati.dump`'. Questo file va verificato alla ricerca di segnalazioni di errore che potrebbero essere generate in presenza di dati che non possono essere riprodotti fedelmente; eventualmente, può essere modificato se si conosce la sintassi dei comandi che vengono inseriti in questo script.

Il recupero dell'insieme completo delle basi di dati avviene normal-

mente in un ambiente PostgreSQL, in cui il sistema delle basi di dati sia stato predisposto, ma non sia stata creata alcuna base di dati (a parte quelle standard, come '**template1**'). Come si può intuire, il comando necessario per ricaricare le basi di dati, assieme alle informazioni sugli utenti (la relazione '**pg_shadow**'), è quello seguente:

```
postgres$ psql -e template1 < basi_dati.dump [Invio]
```

La situazione tipica in cui è necessario utilizzare '**pg_dumpall**' per scaricare tutto il sistema delle basi di dati, è quella del momento in cui ci si accinge ad aggiornare la versione di PostgreSQL. In breve, in quella occasione, si devono eseguire i passaggi seguenti:

1. con la versione vecchia di PostgreSQL, si deve utilizzare '**pg_dumpall**' in modo da scaricare tutto il sistema delle basi di dati in un solo file di testo;
2. si aggiorna PostgreSQL;
3. si elimina il contenuto della directory '`~postgres/data/`', ovvero quella che altrimenti viene definita '**PGDATA**' (prima conviene forse fare una copia di sicurezza del suo contenuto, tale e quale, in forma binaria);
4. si ricrea il sistema delle basi di dati, vuoto, attraverso '**initdb**';
5. si ricaricano le basi di dati precedenti, assieme alle informazioni sugli utenti, attraverso '**psql**', utilizzando il file generato in precedenza attraverso '**pg_dumpall**'.

Quello che manca, eventualmente, è la configurazione di PostgreSQL, in particolare per ciò che riguarda i sistemi di accesso e au-

tenticazione (il file '`~postgres/data/pg_hda.conf`'), che deve essere ripristinata manualmente.

75.2.7 Importazione ed esportazione dei dati

Al posto di utilizzare gli script già pronti per la copia e il recupero dei dati, è possibile avvalersi di comandi SQL. PostgreSQL fornisce un'istruzione speciale per permettere l'importazione e l'esportazione dei dati da e verso un file indipendente dalla piattaforma. Si tratta dell'istruzione '**`COPY`**', la cui sintassi semplificata è quella seguente:

```
COPY relazione TO { 'file' | STDIN }
    [ [WITH]
        [BINARY]
        [DELIMITER [AS] 'delimitatore' ] ]
```

```
COPY relazione FROM { 'file' | STDIN }
    [ [WITH]
        [BINARY]
        [DELIMITER [AS] 'delimitatore' ] ]
```

Nella prima delle due forme, si esportano i dati verso un file o verso lo standard input; nella seconda si importano da un file o dallo standard output.

Se si usa l'opzione '**`BINARY`**' si ottiene un file «binario» indipendente dalla piattaforma; diversamente si ottiene un file di testo tradizionale. Nel caso del file di testo, ogni riga corrisponde a una tupla della relazione; gli attributi sono separati da un carattere di delimitazione, che in mancanza della definizione tramite l'opzione '**`DELIMITER`**

AS’ è un carattere di tabulazione. In ogni caso, anche se si specifica tale opzione, può trattarsi solo di un carattere. In pratica, sempre nell’ipotesi di creazione di un file di testo, ogni riga è organizzata secondo lo schema seguente:

attributo_1 *x* ***attributo_2*** *x* ··· *x* ***attributo_n***

Nello schema, ***x*** rappresenta il carattere di delimitazione, che, come si può vedere, non viene inserito all’inizio e alla fine.

Quando l’istruzione ‘**COPY**’ viene usata per importare dati dallo standard input, in formato testo, è necessario che dopo l’ultima riga che contiene attributi da inserire nella relazione, sia presente una sequenza di escape speciale: una barra obliqua inversa seguita da un punto (‘**\.**’). Il file ottenuto quando si esporta verso lo standard output contiene questo simbolo di conclusione.

Il file di testo in questione può contenere anche altre sequenze di escape, che si trovano descritte nella tabella 75.34.

Tabella 75.34. Sequenze di escape nei file di testo generati e utilizzati da ‘**COPY**’.

Escape	Descrizione
\\	Una barra obliqua inversa.
\.	Simbolo di conclusione del file.
\N	‘**NULL**’.
delimitatore	Protegge il simbolo che viene già utilizzato come delimitatore.

Escape	Descrizione
`\`<*LF*>	Tratta <*LF*> in modo letterale.
`\b`	<*BS*>.
`\f`	<*FF*>.
`\n`	<*LF*>.
`\r`	<*CR*>.
`\t`	<*HT*> (tabulazione orizzontale).
`\v`	<*VT*> (tabulazione verticale).
`\`***ooo***	Codice per un byte espresso in ottale.

È importante fare mente locale al fatto che l'istruzione viene eseguita dal servente. Ciò significa che i file, quando non si tratta di standard input o di standard output, sono creati o cercati secondo il file system che questo servente si trova ad avere sotto di sé.

Segue la descrizione di alcuni esempi.

- ```
  COPY Indirizzi TO STDOUT;
  ```

  L'esempio mostra l'istruzione necessaria a emettere attraverso lo standard output del programma cliente ('**psql**') la trasformazione in testo del contenuto della relazione '**Indirizzi**'.
  ```

- `COPY Indirizzi TO STDOUT BINARY;`

  Come nell'esempio precedente, generando però un formato binario, indipendente dalla piattaforma.

- `COPY Indirizzi TO '/tmp/prova' WITH DELIMITER AS '|';`

  In questo caso, si genera il file di testo '`/tmp/prova`' nel file system dell'elaboratore servente, inoltre gli attributi sono separati attraverso una barra verticale ('`|`').

- `COPY Indirizzi FROM STDIN;`

  In questo caso, si aggiungono tuple alla relazione '**`Indirizzi`**', utilizzando quanto proviene dallo standard input, che si attende essere un file di testo (alla fine deve apparire la sequenza di escape '`\.`').

- `COPY Indirizzi FROM STDIN BINARY;`

  Come nell'esempio precedente, attendendo i dati in formato binario.

- `COPY Indirizzi FROM '/tmp/prova' WITH DELIMITER AS '|';`

  Si aggiungono tuple alla relazione '**`Indirizzi`**', utilizzando quanto proviene dal file '`/tmp/prova`', che si trova nel file system dell'elaboratore servente. Il file deve essere in formato testo e gli attributi si intendono separati da una barra verticale ('`|`').

75.3 Il linguaggio

«

PostgreSQL è un ORDBMS, ovvero un *Object-relational DBMS*, cioè un DBMS relazionale a oggetti. La sua documentazione utilizza terminologie differenti, a seconda delle preferenze dei rispettivi autori. In generale si possono distinguere tre modalità, riferite

a tre punti di vista: la programmazione a oggetti, la teoria generale sui DBMS e il linguaggio SQL. Le equivalenze dei termini sono riassunte dallo schema seguente:

classi	istanze	attributi	tipi di dati contenibili negli attributi
relazioni	tuple	attributi	domini
tabelle	righe	colonne	tipi di dati contenibili nelle colonne

In questo capitolo si intende usare la terminologia tradizionale dei DBMS, dove i dati sono organizzati in relazioni, tuple e attributi, affiancando eventualmente i termini del linguaggio SQL tradizionale (tabelle, righe e colonne). Inoltre, la sintassi delle istruzioni (interrogazioni) SQL che vengono mostrate è limitata alle funzionalità più semplici, sempre compatibilmente con le possibilità di PostgreSQL. Per una visione più estesa delle funzionalità SQL di PostgreSQL conviene consultare la sua documentazione.

75.3.1 Prima di iniziare

Per fare pratica con il linguaggio SQL, il modo migliore è quello di utilizzare il programma '**psql**' con il quale si possono eseguire interrogazioni interattive con il servente. Quello che conta è tenere a mente che per poterlo utilizzare occorre avere già creato una base di dati (vuota), in cui vanno poi inserite delle nuove relazioni, con le quali si possono eseguire altre operazioni.

Attraverso le istruzioni SQL si fa riferimento sempre a un'unica base di dati: quella a cui ci si collega quando si avvia '**psql**'.

Utilizzando '**psql**', le istruzioni devono essere terminate con il punto e virgola ('**;**'), oppure dal comando interno '**\g**' (*go*).

75.3.2 Tipi di dati e rappresentazione

«

I tipi di dati gestibili sono un punto delicato della compatibilità tra un DBMS e lo standard SQL. Vale la pena di riepilogare i tipi più comuni, compatibili con lo standard SQL, che possono essere trovati nella tabella 75.42.

Tabella 75.42. Elenco dei tipi di dati standard utilizzabili con PostgreSQL.

Tipo	Standard	Descrizione
`CHAR` `CHARACTER`	SQL92	Un carattere singolo.
`CHAR(`***n***`)` `CHARACTER(`***n***`)`	SQL92	Una stringa di lunghezza fissa, di ***n*** caratteri, completata da spazi.
`VARCHAR(`***n***`)` `CHARACTER VARYING(`***n***`)` `CHAR VARYING(`***n***`)`	SQL92	Una stringa di lunghezza variabile con un massimo di ***n*** caratteri.
`INTEGER`	SQL92	Intero (al massimo nove cifre numeriche).
`SMALLINT`	SQL92	Intero più piccolo di '**`INTEGER`**' (al massimo quattro cifre numeriche).
`FLOAT`	SQL92	Numero a virgola mobile.
`FLOAT(`***n***`)`	SQL92	Numero a virgola mobile lungo ***n*** bit.

Tipo	Standard	Descrizione
`REAL`	SQL92	Numero a virgola mobile (teoricamente più preciso di '**`FLOAT`**').
`DOUBLE PRECISION`	SQL92	Numero a virgola mobile (più o meno equivalente a '**`REAL`**').
`NUMERIC` `NUMERIC` (***precisione*** [, ***scala***]) `DECIMAL` `DECIMAL` (***precisione*** [, ***scala***]) `DEC` `DEC(`***precisione*** [, ***scala***]`)`	SQL92	Numero composto da un massimo di tante cifre numeriche quante indicate dalla precisione, cioè il primo argomento tra parentesi. Se viene specificata anche la scala, si intende riservare quella parte di cifre per quanto appare dopo la virgola.
`DATE`	SQL92	Data, di solito nella forma '***mm/gg/aaaa***'.
`TIME`	SQL92	Orario, nella forma '***hh:mm:ss***', oppure solo '***hh:mm***'.
`TIMESTAMP`	SQL92	Informazione completa data-orario.
`INTERVAL`	SQL92	Intervallo di tempo.
`BIT(`***n***`)`	SQL92	Stringa binaria di dimensione fissa.
`BIT VARYING(`***n***`)`	SQL92	Stringa binaria di dimensione variabile.

Tipo	Standard	Descrizione
`BOOLEAN`	SQL99	Valore logico booleano.

Oltre ai tipi di dati gestibili, è necessario conoscere il modo di rappresentarli in forma costante. In particolare, è bene osservare che PostgreSQL ammette solo l'uso degli apici singoli come delimitatori; pertanto, per rappresentare un apice in una stringa delimitata in questo modo, lo si può raddoppiare, oppure si può usare la sequenza di escape '`\'`'. La tabella 75.43 mostra alcuni esempi.

Tabella 75.43. Esempi di rappresentazione dei valori costanti. Si osservi che in alcuni casi, conviene dichiarare il tipo di valore, seguito da una costante stringa che lo rappresenta, come in questi esempi a proposito di valori data-orario.

Tipo di valore in generale	Esempi di rappresentazione in forma di costante letterale
`CHAR` `CHARACTER` `CHAR(`***n***`)` `CHARACTER(`***n***`)` `VARCHAR(`***n***`)` `CHARACTER VARYING(`***n***`)` `CHAR VARYING(`***n***`)`	`'a'` `'ciao'` `'Ciao'` `'123/der:876'`

Tipo di valore in generale	Esempi di rappresentazione in forma di costante letterale
INTEGER SMALLINT	1 123 -987
FLOAT FLOAT(*n*) REAL DOUBLE PRECISION NUMERIC NUMERIC(*precisione* [, *scala*]) DECIMAL DECIMAL(*precisione* [, *scala*]) DEC DEC(*precisione* [, *scala*])	123.45 -45.3 123.45e+10 123.45e-10

Tipo di valore in generale	Esempi di rappresentazione in forma di costante letterale
DATE	DATE '31.12.2012' DATE '12/31/2012' DATE '2012-12-31'
TIME	TIME '15:55:27' TIME '15:59'
TIMESTAMP	TIMESTAMP '2012-12-31 15:55:27' TIMESTAMP '2012-12-31 15:55:27+1'
INTERVAL	INTERVAL '15:55:27' INTERVAL '15 HOUR 59 MINUTE' INTERVAL '- 15 HOUR'
BIT BIT VARYING(*n*)	B'1' B'101' X'2F'

Tipo di valore in generale	Esempi di rappresentazione in forma di costante letterale
BOOLEAN	1 'y' 'yes' 't' 'true' 0 'n' 'no' 'f' 'false'

In particolare, le costanti stringa possono contenere delle sequenze di escape, rappresentate da una barra obliqua inversa seguita da un simbolo. La tabella 75.44 mostra le sequenze di escape tipiche e inserisce anche il caso del raddoppio degli apici singoli.

Tabella 75.44. Sequenze di escape utilizzabili all'interno delle stringhe di caratteri costanti.

Escape	Significato
\n	*<LF>*

Escape	Significato
`\r`	*<CR>*
`\b`	*<BS>*
`\'`	'
`''`	'
`\"`	"
`\\`	\
`\%`	*%*
`\_`	_

75.3.3 Funzioni

PostgreSQL, come altri DBMS SQL, offre una serie di funzioni che fanno parte dello standard SQL, assieme ad altre non standard che però sono ampiamente diffuse e di grande utilità. Le tabelle 75.45 e 75.46 ne riportano alcune.

Tabella 75.45. Funzioni SQL riconosciute da PostgreSQL.

Funzione	Descrizione
`POSITION(`***stringa_1*** `IN` ***stringa_2***`)`	Posizione di ***stringa_1*** in ***stringa_2***.
`SUBSTRING(`***stringa*** `[FROM` ***n***`] [FOR` ***m***`])`	Sottostringa da ***n*** per ***m*** caratteri.

Funzione	Descrizione
`TRIM([LEADING|TRAILING|BOTH] ↩ ↪ ['x'] FROM [stringa])`	Ripulisce all'inizio e alla fine del testo.

Tabella 75.46. Alcune funzioni riconosciute dal linguaggio di PostgreSQL.

Funzione	Descrizione
`UPPER(`***stringa***`)`	Converte la stringa in caratteri maiuscoli.
`LOWER(`***stringa***`)`	Converte la stringa in caratteri minuscoli.
`INITCAP(`***stringa***`)`	Converte la stringa in modo che le parole inizino con la maiuscola.
`SUBSTR(`***stringa***`,`***n***`,`***m***`)`	Estrae la stringa che inizia dalla posizione ***n***, lunga ***m*** caratteri.
`LTRIM(`***stringa***`, 'x')`	Ripulisce la stringa a sinistra (*Left trim*).
`RTRIM(`***stringa***`, 'x')`	Ripulisce la stringa a destra (*Right trim*).

Segue la descrizione di alcuni esempi.

- ```
  SELECT POSITION( 'o' IN 'Topo' )
  ```
  Restituisce il valore due.

- ```
  SELECT POSITION( 'ino' IN Cognome ) FROM Indirizzi
  ```
  Restituisce un elenco delle posizioni in cui si trova la stringa '**ino**' all'interno dell'attributo '**Cognome**', per tutte le tuple della relazione '**Indirizzi**'.

- ```
  SELECT SUBSTRING( 'Daniele' FROM 3 FOR 2 )
  ```
  ```

Restituisce la stringa '**ni**'.

- ```
  SELECT TRIM( LEADING '*' FROM '*****Ciao****' )
  ```

  Restituisce la stringa '**Ciao****'.

- ```
  SELECT TRIM( TRAILING '*' FROM '*****Ciao****' )
  ```

  Restituisce la stringa '*****Ciao**'.

- ```
  SELECT TRIM( BOTH '*' FROM '*****Ciao****' )
  ```

  Restituisce la stringa '**Ciao**'.

- ```
  SELECT TRIM( BOTH ' ' FROM '    Ciao    ' )
  ```

  Restituisce la stringa '**Ciao**'.

- ```
  SELECT TRIM( '    Ciao    ' )
  ```

  Esattamente come nell'esempio precedente, dal momento che lo spazio normale è il carattere predefinito e che la parola chiave '**BOTH**' è anche predefinita.

- ```
  SELECT LTRIM( '*****Ciao****', '*' )
  ```

  Restituisce la stringa '**Ciao****'.

- ```
  SELECT RTRIM( '*****Ciao****', '*' )
  ```

  Restituisce la stringa '*****Ciao**'.

75.3.4 Esempi comuni

«

Nelle sezioni seguenti vengono mostrati alcuni esempi comuni di utilizzo del linguaggio SQL, limitato alle possibilità di PostgreSQL. La sintassi non viene descritta, salvo quando la differenza tra quella standard e quella di PostgreSQL è importante.

Negli esempi si fa riferimento frequentemente a una relazione di indirizzi, il cui contenuto è visibile nella figura 75.57.
  ```

Figura 75.57. La relazione '**`Indirizzi (Codice, Cognome, Nome, Indirizzo, Telefono)`**' usata in molti esempi del capitolo.

Indirizzi				
Codice	**Cognome**	**Nome**	**Indirizzo**	**Telefono**
1	Pallino	Pinco	Via Biglie 1	0222,222222
2	Tizi	Tizio	Via Tazi 5	0555,555555
3	Cai	Caio	Via Caini 1	0888,888888
4	Semproni	Sempronio	Via Sempi 7	0999,999999

75.3.4.1 Creazione di una relazione

La relazione di esempio mostrata nella figura 75.57, potrebbe essere creata nel modo seguente:

```
CREATE TABLE Indirizzi (
        Codice          integer,
        Cognome         char(40),
        Nome            char(40),
        Indirizzo       varchar(60),
        Telefono        varchar(40)
    );
```

Quando si inseriscono i valori per una tupla, può capitare che venga omesso l'inserimento di alcuni attributi. In questi casi, il campo corrispondente riceve il valore '**`NULL`**', cioè un valore indefinito, oppure il valore predefinito attraverso quanto specificato con l'espressione che segue la parola chiave '**`DEFAULT`**'.

In alcuni casi non è possibile definire un valore predefinito e nem-

meno è accettabile che un dato resti indefinito. In tali situazioni si può aggiungere l'opzione '**NOT NULL**', dopo la definizione del tipo.

75.3.4.2 Modifica della relazione

La modifica di una relazione implica l'intervento sulle caratteristiche degli attributi, oppure la loro aggiunta ed eliminazione. Seguono due esempi, con cui si aggiunge un attributo e poi lo si elimina:

```
ALTER TABLE Indirizzi ADD COLUMN Comune char(30);
```

```
ALTER TABLE Indirizzi DROP COLUMN Comune;
```

L'esempio seguente modifica il tipo di un attributo già esistente:

```
ALTER TABLE Indirizzi ALTER COLUMN Codice TYPE REAL;
```

Naturalmente, la conversione del tipo di un attributo può avere significato solo se i valori contenuti nelle tuple esistenti, in corrispondenza di quell'attributo, sono convertibili.

75.3.4.3 Inserimento dati in una relazione

L'esempio seguente mostra l'inserimento dell'indirizzo dell'impiegato «Pinco Pallino».

```
INSERT INTO Indirizzi
    VALUES (
        01,
        'Pallino',
        'Pinco',
        'Via Biglie 1',
        '0222,222222'
    );
```

In questo caso, si presuppone che i valori inseriti seguano la sequenza degli attributi, così come è stata creata la relazione in origine.

Se si vuole indicare un comando più leggibile, occorre aggiungere l'indicazione della sequenza degli attributi da compilare, come nell'esempio seguente:

```
INSERT INTO Indirizzi (
        Codice,
        Cognome,
        Nome,
        Indirizzo,
        Telefono
    )
    VALUES (
        01,
        'Pallino',
        'Pinco',
        'Via Biglie 1',
        '0222,222222'
    );
```

In questo stesso modo, si può evitare di compilare il contenuto di un attributo particolare, indicando espressamente solo gli attributi che si vogliono fornire; in tal caso gli altri attributi ricevono il valore predefinito o '**NULL**' in mancanza d'altro. Nell'esempio seguente viene indicato solo il codice e il nominativo:

```
INSERT INTO Indirizzi (
        Codice,
        Cognome,
        Nome,
    )
    VALUES (
        01,
        'Pallino'
        'Pinco',
    );
```

75.3.4.4 Eliminazione di una relazione

Una relazione può essere eliminata completamente attraverso l'istruzione '**DROP**'. L'esempio seguente elimina la relazione degli indirizzi degli esempi già mostrati:

```
DROP TABLE Indirizzi;
```

75.3.4.5 Interrogazioni semplici

L'esempio seguente emette tutto il contenuto della relazione degli indirizzi già vista negli esempi precedenti:

```
SELECT * FROM Indirizzi;
```

Seguendo l'esempio fatto in precedenza si dovrebbe ottenere l'elenco riportato sotto, equivalente a tutto il contenuto della relazione.

```
codice  cognome   nome       indirizzo     telefono

     1  Pallino   Pinco      Via Biglie 1  0222,222222
     2  Tizi      Tizio      Via Tazi 5    0555,555555
     3  Cai       Caio       Via Caini 1   0888,888888
     4  Semproni  Sempronio  Via Sempi 7   0999,999999
```

Per ottenere un elenco ordinato in base al cognome e al nome (in caso di ambiguità), lo stesso comando si completa nel modo seguente:

```
SELECT * FROM Indirizzi ORDER BY Cognome, Nome;
```

```
codice  cognome   nome       indirizzo     telefono

     3  Cai       Caio       Via Caini 1   0888,888888
     1  Pallino   Pinco      Via Biglie 1  0222,222222
     4  Semproni  Sempronio  Via Sempi 7   0999,999999
     2  Tizi      Tizio      Via Tazi 5    0555,555555
```

La selezione degli attributi permette di ottenere un risultato che contenga solo quelli desiderati, permettendo anche di cambiarne l'intestazione. L'esempio seguente permette di mostrare solo i nominativi e il telefono, cambiando un po' le intestazioni:

```
SELECT Cognome as cognomi, Nome as nomi,
       Telefono as numeri_telefonici
    FROM Indirizzi;
```

Quello che si ottiene è simile all'elenco seguente:

```
cognomi   nomi       numeri_telefonici

Pallino   Pinco      0222,222222
Tizi      Tizio      0555,555555
Cai       Caio       0888,888888
Semproni  Sempronio  0999,999999
```

La selezione delle tuple può essere fatta attraverso la condizione che segue la parola chiave '**WHERE**'. Nell'esempio seguente vengono selezionate le tuple in cui l'iniziale dei cognomi è compresa tra '**N**' e '**T**'.

```
SELECT * FROM Indirizzi
         WHERE Cognome >= 'N' AND Cognome <= 'T';
```

Dall'elenco che si ottiene, si osserva che '**Caio**' è stato escluso:

```
codice  cognome   nome       indirizzo     telefono

     1  Pallino   Pinco      Via Biglie 1  0222,222222
     2  Tizi      Tizio      Via Tazi 5    0555,555555
     4  Semproni  Sempronio  Via Sempi 7   0999,999999
```

Per evitare ambiguità possono essere indicati i nomi degli attributi prefissati dal nome della relazione a cui appartengono, separando le due parti con l'operatore punto ('.'). Nell'esempio seguente si selezionano solo il cognome, il nome e il numero telefonico, specificando il nome della relazione a cui appartengono gli attributi:

```
SELECT Indirizzi.Cognome, Indirizzi.Nome,
       Indirizzi.Telefono
    FROM Indirizzi;
```

Ecco il risultato:

```
cognome   nome       telefono

Pallino   Pinco      0222,222222
Tizi      Tizio      0555,555555
Cai       Caio       0888,888888
Semproni  Sempronio  0999,999999
```

75.3.4.6 Interrogazioni simultanee di più relazioni

«

Se dopo la parola chiave '**FROM**' si indicano più relazioni (ciò vale anche se si indica più volte la stessa relazione), si intende fare riferimento a una relazione generata dal «prodotto» di queste. Si immagini di abbinare alla relazione '**Indirizzi**' la relazione '**Presenze**' contenente i dati visibili nella figura 75.76.

Figura 75.76. La relazione '**Presenze (Codice, Giorno, Ingresso, Uscita)**'.

Presenze			
Codice	**Giorno**	**Ingresso**	**Uscita**
1	01/01/2012	07:30	13:30
2	01/01/2012	07:35	13:37
3	01/01/2012	07:45	14:00
4	01/01/2012	08:30	16:30
1	01/02/2012	07:35	13:38
2	01/02/2012	08:35	14:37
4	01/02/2012	07:30	13:30

Come si può intendere, il primo attributo, '**Codice**', serve a identificare la persona per la quale è stata fatta l'annotazione dell'ingresso e dell'uscita. Tale codice viene interpretato in base al contenuto della relazione '**Indirizzi**'. Si immagini di volere ottenere un elenco contenente tutti gli ingressi e le uscite, indicando chiaramente il cognome e il nome della persona a cui si riferiscono.

```
SELECT
    Presenze.Giorno,
    Presenze.Ingresso,
    Presenze.Uscita,
    Indirizzi.Cognome,
    Indirizzi.Nome
    FROM Presenze, Indirizzi
    WHERE Presenze.Codice = Indirizzi.Codice;
```

Ecco quello che si dovrebbe ottenere:

```
giorno      ingresso  uscita    cognome   nome

01-01-2012  07:30:00  13:30:00  Pallino   Pinco
01-01-2012  07:35:00  13:37:00  Tizi      Tizio
01-01-2012  07:45:00  14:00:00  Cai       Caio
01-01-2012  08:30:00  16:30:00  Semproni  Sempronio
01-02-2012  07:35:00  13:38:00  Pallino   Pinco
01-02-2012  08:35:00  14:37:00  Tizio     Tizi
01-02-2012  07:40:00  13:30:00  Semproni  Sempronio
```

75.3.4.7 Alias

«

Una stessa relazione può essere presa in considerazione come se si trattasse di due o più relazioni differenti. Per distinguere tra questi punti di vista diversi, si devono usare degli alias, che sono in pratica dei nomi alternativi. Gli alias si possono usare anche solo per questioni di leggibilità. L'esempio seguente è la semplice ripetizione di quello mostrato nella sezione precedente, con l'aggiunta però della definizione degli alias '**Pre**' e '**Nom**'.

```
SELECT
    Pre.Giorno,
    Pre.Ingresso,
    Pre.Uscita,
    Nom.Cognome,
    Nom.Nome
    FROM Presenze AS Pre, Indirizzi AS Nom
    WHERE Pre.Codice = Nom.Codice;
```

75.3.4.8 Viste

«

Attraverso una vista, è possibile definire una relazione virtuale:

```
CREATE VIEW Presenze_dettagliate AS
SELECT
    Presenze.Giorno,
    Presenze.Ingresso,
    Presenze.Uscita,
    Indirizzi.Cognome,
    Indirizzi.Nome
    FROM Presenze, Indirizzi
    WHERE Presenze.Codice = Indirizzi.Codice;
```

L'esempio mostra la creazione della vista '**Presenze_dettagliate**', ottenuta dalle relazioni '**Presenze**' e '**Indirizzi**'. In pratica, questa vista permette di interrogare direttamente la relazione virtuale '**Presenze_dettagliate**', invece di utilizzare ogni volta un comando '**SELECT**' molto complesso, per ottenere lo stesso risultato.

75.3.4.9 Aggiornamento delle tuple

«

La modifica di tuple già esistenti avviene attraverso l'istruzione '**UPDATE**', la cui efficacia viene controllata dalla condizione posta dopo la parola chiave '**WHERE**'. Se tale condizione manca, l'effetto delle modifiche si riflette su tutte le tuple della relazione.

L'esempio seguente, aggiunge un attributo alla relazione degli indirizzi, per contenere il nome del comune di residenza degli impiegati; successivamente viene inserito il nome del comune '**Sferopoli**' in base al prefisso telefonico.

```
ALTER TABLE Indirizzi ADD COLUMN Comune char(30);
```

```
UPDATE Indirizzi
    SET Comune='Sferopoli'
    WHERE Telefono >= '022' AND Telefono < '023';
```

In pratica, viene aggiornata solo la tupla dell'impiegato '**Pinco Pallino**'.

75.3.4.10 Cancellazione delle tuple

«

L'esempio seguente elimina dalla relazione delle presenze le tuple riferite alle registrazioni del giorno 01/01/2012 e le eventuali antecedenti.

```
DELETE FROM Presenze WHERE Giorno <= '01/01/2012';
```

75.3.4.11 Creazione di una nuova relazione a partire da altre

«

L'esempio seguente crea la relazione '**mia_prova**' dalla fusione della relazioni degli indirizzi e delle presenze, come già mostrato in un esempio precedente:

```
SELECT
    Presenze.Giorno,
    Presenze.Ingresso,
    Presenze.Uscita,
    Indirizzi.Cognome,
    Indirizzi.Nome
    INTO TABLE mia_prova
    FROM Presenze, Indirizzi
    WHERE Presenze.Codice = Indirizzi.Codice;
```

75.3.4.12 Inserimento in una relazione esistente

L'esempio seguente aggiunge alla relazione dello storico delle presenze le registrazioni vecchie che poi vengono cancellate.

```
INSERT INTO PresenzeStorico (
        PresenzeStorico.Codice,
        PresenzeStorico.Giorno,
        PresenzeStorico.Ingresso,
        PresenzeStorico.Uscita
    )
    SELECT
        Presenze.Codice,
        Presenze.Giorno,
        Presenze.Ingresso,
        Presenze.Uscita
        FROM Presenze
        WHERE Presenze.Giorno <= '2012/01/01';

DELETE FROM Presenze WHERE Giorno <= '2012/01/01';
```

75.3.4.13 Controllare gli accessi a una relazione

Quando si creano delle relazioni in una base di dati, tutti gli altri utenti che sono stati registrati nel sistema del DBMS, potrebbero accedervi e fare le modifiche che vogliono. Per controllare questi accessi, l'utente proprietario delle relazioni (di solito è colui che le ha create), può usare le istruzioni '**GRANT**' e '**REVOKE**'. La prima permette a un gruppo di utenti di eseguire operazioni determinate, la seconda toglie dei privilegi.

```
GRANT {ALL | SELECT | INSERT | UPDATE | DELETE
              | RULE} [,…]
    ON relazione [,…]
    TO {PUBLIC | GROUP gruppo | utente}
```

```
REVOKE {ALL | SELECT | INSERT | UPDATE | DELETE
              | RULE} [,…]
    ON relazione [,…]
    FROM {PUBLIC | GROUP gruppo | utente}
```

La sintassi delle due istruzioni è simile, basta fare attenzione a cambiare la parola chiave '**TO**' con '**FROM**'. I gruppi e gli utenti sono nomi che fanno riferimento a quanto registrato all'interno del DBMS.

L'esempio seguente toglie a tutti gli utenti ('**PUBLIC**') tutti i privilegi sulle relazioni delle presenze e degli indirizzi; successivamente vengono ripristinati tutti i privilegi solo per l'utente '**tizio**':

```
REVOKE ALL
    ON Presenze, Indirizzi
    FROM PUBLIC;

GRANT ALL
    ON Presenze, Indirizzi
    TO tizio;
```

75.3.5 Controllo delle transazioni

La gestione delle transazioni richiede che queste siano introdotte dall'istruzione '**START TRANSACTION**':

```
START TRANSACTION
```

L'esempio seguente mostra il caso in cui si voglia isolare l'inserimento di una tupla nella relazione '**Indirizzi**' all'interno di una transazione, che alla fine viene confermata regolarmente con l'istruzione '**COMMIT**':

```
START TRANSACTION;

INSERT INTO Indirizzi
    VALUES (
        05,
        'De Pippo',
        'Pippo',
        'Via Pappo, 5',
        '0333,3333333'
    );

COMMIT;
```

Nell'esempio seguente, si rinuncia all'inserimento della tupla con l'istruzione '**ROLLBACK**' finale:

```
START TRANSACTION;

INSERT INTO Indirizzi
    VALUES (
        05,
        'De Pippo',
        'Pippo',
        'Via Pappo, 5',
        '0333,3333333'
    );

ROLLBACK;
```

75.3.6 Cursori

«

La gestione dei cursori da parte di PostgreSQL è abbastanza compatibile con lo standard, a parte il fatto che avviene fuori dal contesto previsto, che viene consentito un accesso in sola lettura e che non è possibile assegnare i dati a delle variabili.

La gestione dei cursori riguarda generalmente gli accessi a un DBMS tramite codice SQL incorporato in un programma (che usa un altro linguaggio), mentre PostgreSQL estende il loro utilizzo anche se il «programma» in questione è costituito esclusivamente da codice SQL.

La dichiarazione di un cursore si ottiene nel modo solito, con la differenza che questa deve avvenire esplicitamente in una transazione. In particolare, con PostgreSQL, il cursore viene aperto automaticamente nel momento della dichiarazione, per cui l'istruzione '**`OPEN`**' non è disponibile.

```
START TRANSACTION;

DECLARE Mio_cursore INSENSITIVE CURSOR FOR
    SELECT * FROM Indirizzi ORDER BY Cognome, Nome;

-- L'apertura del cursore non esiste in PostgreSQL
-- OPEN Mio_cursore;
…
```

L'esempio mostra la dichiarazione dell'inizio di una transazione, assieme alla dichiarazione del cursore '**Mio_cursore**', per selezionare tutta la relazione '**Indirizzi**' in modo ordinato per '**Cognome**'. Si osservi che per PostgreSQL la selezione che si ingloba nella gestione di un cursore non può aggiornarsi automaticamente se i dati originali cambiano, per cui è come se fosse sempre definita la parola chiave '**INSENSITIVE**'.

```
…
FETCH NEXT FROM Mio_cursore;
…
COMMIT;
```

L'esempio mostra l'uso tipico dell'istruzione '**FETCH**', in cui si preleva la tupla successiva rispetto alla posizione corrente del cursore e più avanti si conclude la transazione con un '**COMMIT**'. L'esempio seguente è identico, con la differenza che si indica espressamente il passo.

```
…
FETCH RELATIVE 1 FROM Mio_cursore;
…
COMMIT;
```

Un cursore dovrebbe essere chiuso attraverso una richiesta esplicita,

con l'istruzione '**CLOSE**', ma la chiusura della transazione chiude implicitamente il cursore, se questo dovesse essere rimasto aperto. L'esempio seguente riepiloga quanto visto sopra, completato dell'istruzione '**CLOSE**'.

```
START TRANSACTION;

DECLARE Mio_cursore INSENSITIVE CURSOR FOR
    SELECT * FROM Indirizzi ORDER BY Cognome, Nome;

-- L'apertura del cursore non esiste in PostgreSQL
-- OPEN Mio_cursore;

FETCH NEXT FROM Mio_cursore;

CLOSE Mio_cursore;

COMMIT;
```

75.3.7 Impostazione dell'ora locale

«

Il linguaggio SQL dispone dell'istruzione '**SET TIME ZONE**' per definire l'ora locale e di conseguenza lo scostamento dal tempo universale. PostgreSQL dispone della stessa istruzione che funziona in modo molto simile allo standard; per la precisione, la definizione dell'ora locale avviene attraverso le definizioni riconosciute dal sistema operativo (nel caso di GNU/Linux si tratta delle definizioni che si articolano a partire dalla directory '`/usr/share/zoneinfo/`').

```
SET TIME ZONE { 'definizione_ora_locale' | LOCAL }
```

Per esempio, per definire che si vuole fare riferimento all'ora locale italiana, si potrebbe usare il comando seguente:

```
SET TIME ZONE 'Europe/Rome';
```

Questa impostazione riguarda la visione del programma cliente, mentre il programma servente può essere stato preconfigurato attraverso le variabili di ambiente '**LC_***' oppure la variabile '**LANG**', che in questo caso hanno effetto sullo stile di rappresentazione delle informazioni data-orario. Anche il programma cliente può essere preconfigurato attraverso la variabile di ambiente '**PGTZ**', assegnandole gli stessi valori che si possono utilizzare per l'istruzione '**SET TIME ZONE**'.

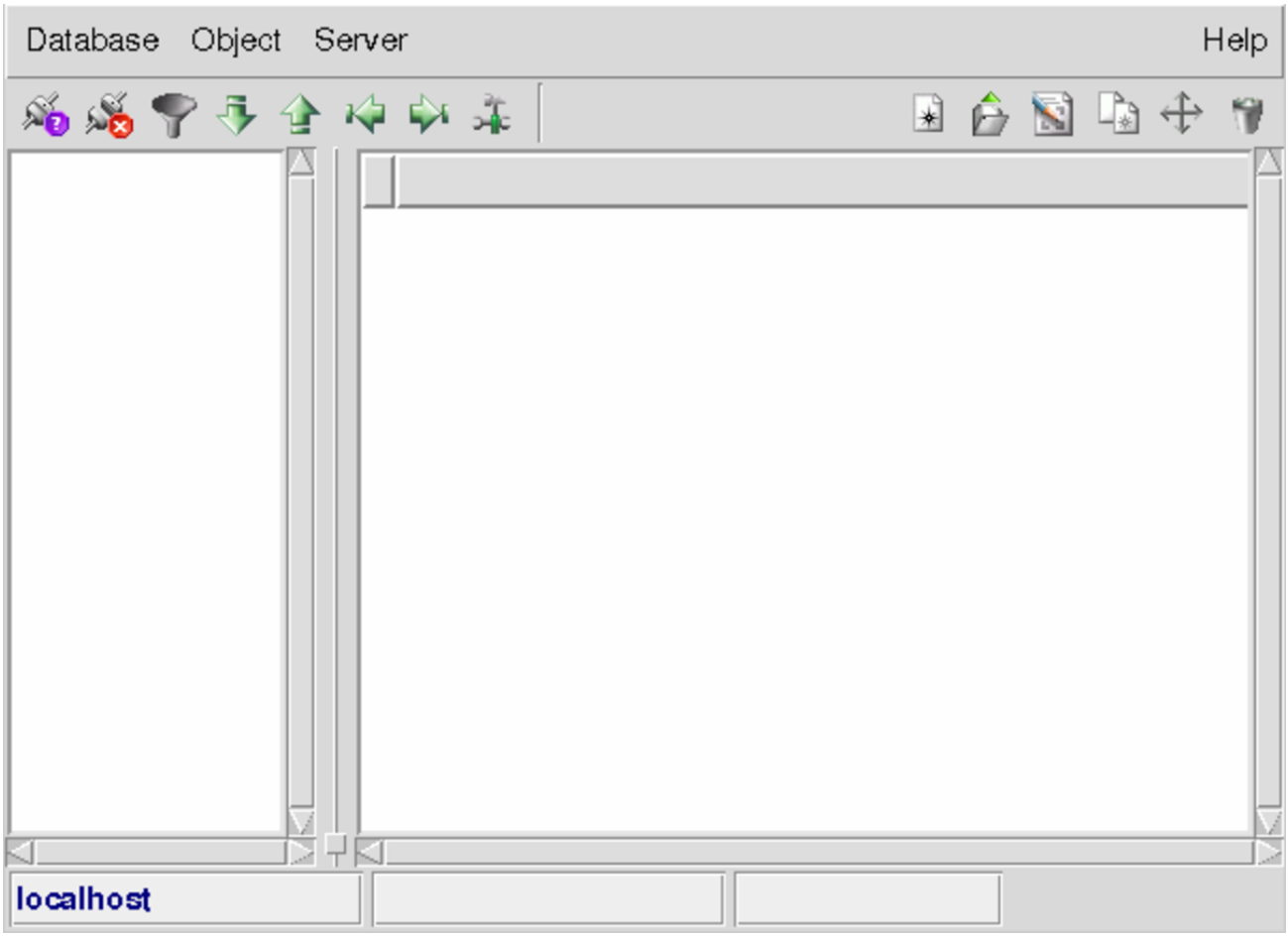
75.4 Accesso attraverso PgAccess

PgAccess [2] (ovvero PostgreSQL Access) è un componente di una libreria Tcl/Tk: LibPgTcl. A volte viene distribuito come un pacchetto autonomo, che comunque dipende dalla libreria indicata, oppure viene incluso nello stesso pacchetto della libreria. PgAccess è un programma frontale (che utilizza l'interfaccia grafica) per accedere alle funzionalità di PostgreSQL.

Prima di poter utilizzare qualunque programma frontale per PostgreSQL, occorre ricordare di configurare correttamente PostgreSQL stesso, in modo che questo consenta gli accessi previsti.

PgAccess è costituito in pratica dall'eseguibile '**pgaccess**', che si utilizza senza argomenti e si presenta inizialmente come si vede nella figura 75.94.

Figura 75.94. Finestra iniziale di PgAccess, quando viene avviato per la prima volta dall'utente.



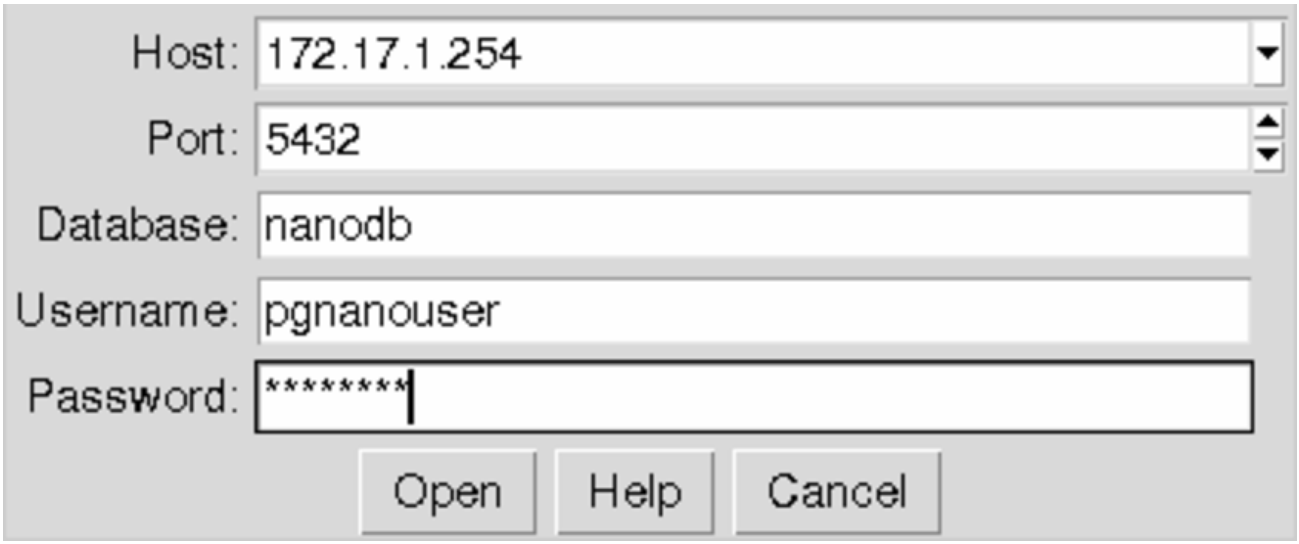
Mentre lo si usa, PgAccess memorizza alcune informazioni nella directory '`~/.pgaccess/`' e questo fatto facilita successivamente le operazioni di accesso alla base di dati da parte dell'utente.

Purtroppo, l'uso di programmi come questo, che mediano la comunicazione con un DBMS attraverso delle finestre grafiche, può risultare più complicato della scrittura manuale del codice SQL necessario. In questo capitolo, le figure appartengono a versioni diverse del programma, perché alcune funzionalità essenziali della versione aggiornata, si sono rivelate inaffidabili.

75.4.1 Accesso alla base di dati

PostgreSQL è un DBMS in grado di gestire diverse basi di dati simultaneamente; pertanto, con PgAccess è necessario stabilire per prima cosa quale sia la base di dati. Dal menù *Database*, si seleziona la funzione *Open*, ottenendo la mascherina che si vede nella figura 75.95. Da lì si possono indicare tutte le informazioni necessarie alla connessione con la base di dati desiderata; in particolare, per quanto riguarda le informazioni sull'autenticazione, queste sono richieste solo in base al modo in cui sono stati regolati i permessi di accesso da parte di PostgreSQL.

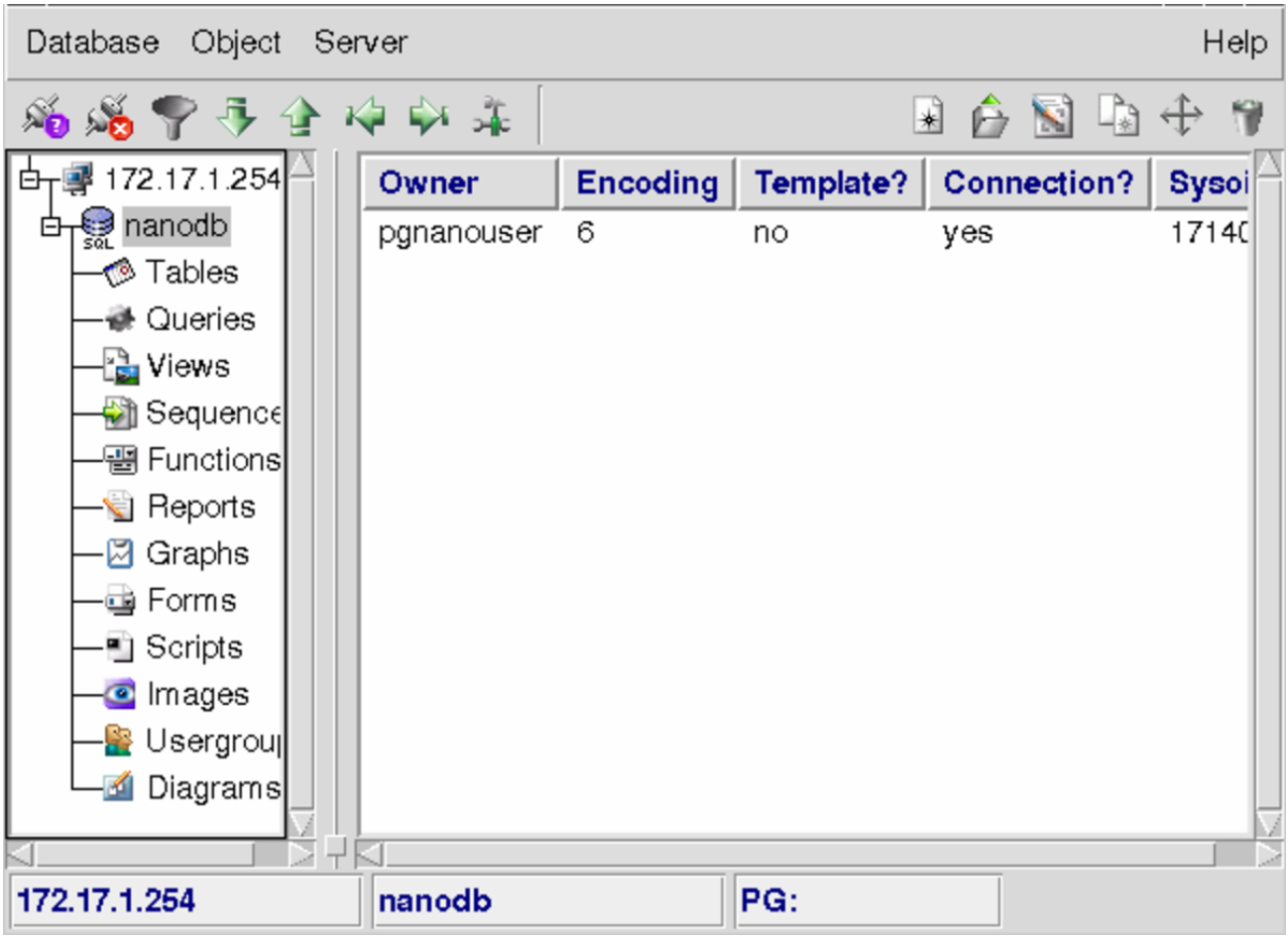
Figura 75.95. Connessione alla base di dati '**nanodb**', presso il nodo 172.17.1.254, come utente '**pgnanouser**'.



Attraverso PgAccess non è possibile creare una base di dati. Per questo occorre usare le funzioni di PostgreSQL, descritto nella sezione 75.2.

La base di dati aperta, assieme all'indicazione del nodo presso il quale si trova il DBMS con cui si interagisce, appare in basso, nella finestra principale di PgAccess.

Figura 75.96. Quando è attiva una connessione con una base di dati, lo si vede dalle informazioni che appaiono in basso nella finestra principale di PgAccess.



È importante ricordare che PgAccess tiene nota dell'ultima base di dati aperta attraverso i file di configurazione contenuti in '`~/.pgaccess/`'; in questo modo la connessione viene ritentata automaticamente all'avvio del programma la volta successiva che lo si utilizza.

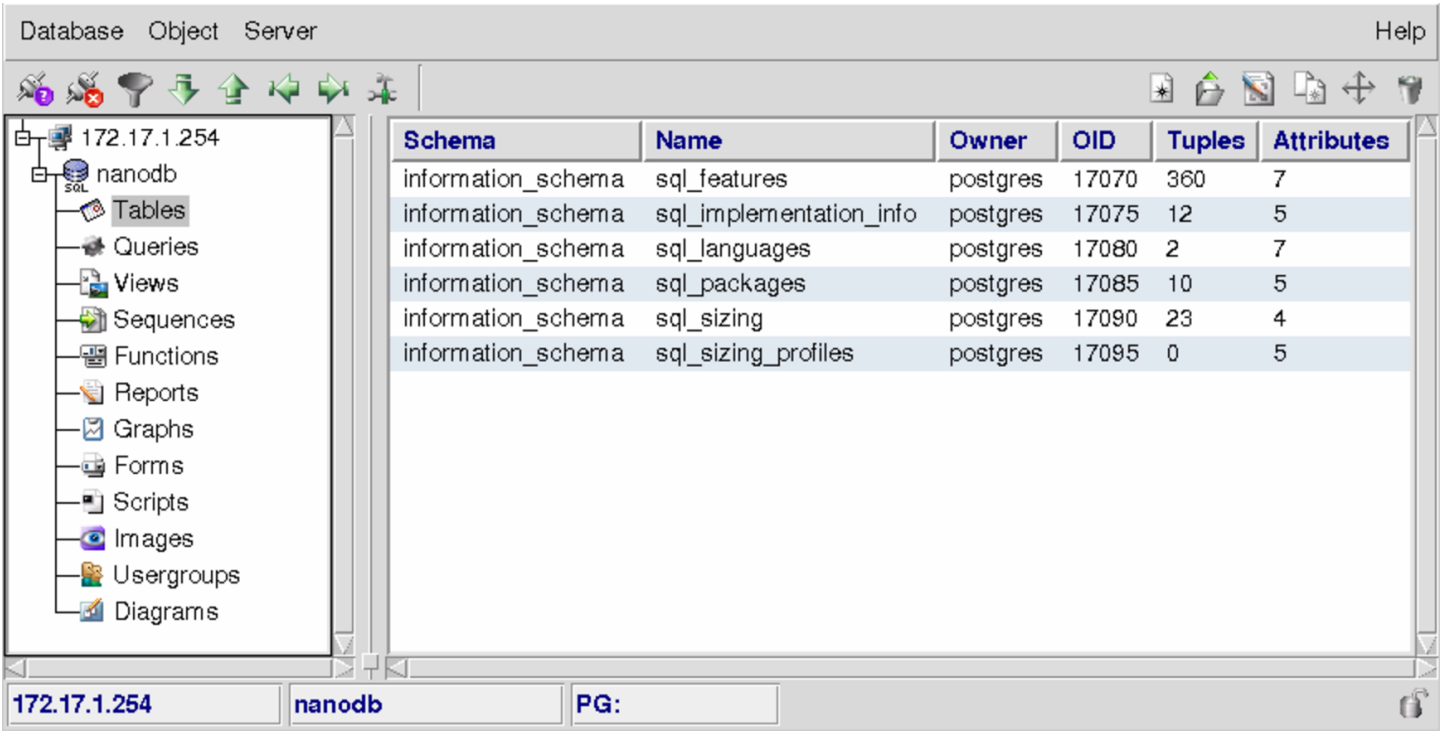
75.4.2 Gli «oggetti» secondo PgAccess

Dal punto di vista di PgAccess, una base di dati contiene degli «oggetti» (secondo la stessa filosofia di PostgreSQL). Questi possono

essere delle relazioni, il risultato di interrogazioni SQL, delle viste, delle stampe o altro ancora.

Per intervenire su ognuno di questi oggetti basta selezionare la voce relativa che si trova sulla parte sinistra (nella figura 75.97 si vede selezionata la gestione delle «tabelle», ovvero delle relazioni).

Figura 75.97. L'aspetto di PgAccess quando viene evidenziata a sinistra la voce *Tables*.



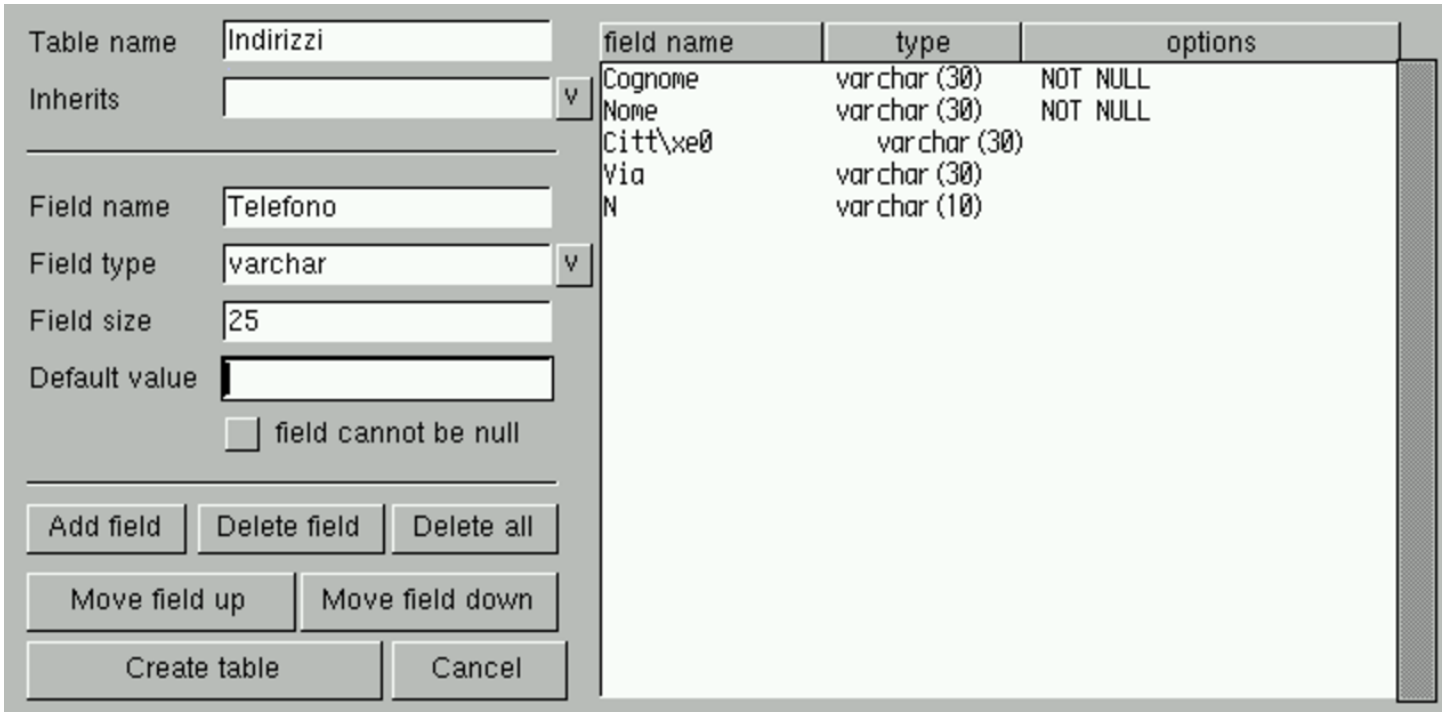
Premendo il tasto destro del mouse quando il puntatore si trova nel riquadro centrale, si ottiene un menù a scomparsa, con il quale è possibile modificare gli oggetti a cui fa riferimento la voce selezionata a sinistra; in alternativa, le stesse voci sono disponibili dal menù *Object*. In particolare, la voce *New* serve a creare un oggetto nuovo, *Open* serve ad accedervi e *Design* serve a modificarne la struttura (ammesso che ciò sia consentito in base al tipo di oggetto). Eventualmente è possibile anche modificare il nome dell'oggetto e visualizzarne la struttura.

PgAccess gestisce una serie aggiuntiva di oggetti rispetto a quanto fa PostgreSQL. Per realizzarli, PgAccess gestisce delle relazioni proprie, che non vengono mostrate all'utente, distinguibili per il fatto di avere un nome che inizia per '**pga_**'. In generale, queste relazioni hanno tutti i permessi di accesso per tutti gli utenti di PostgreSQL.

75.4.3 Relazioni

La figura 75.98 mostra l'esempio della creazione di una relazione molto semplice, per contenere una serie di indirizzi. Alla creazione della relazione, dopo avere selezionato la voce relativa a questo tipo di oggetto, si accede selezionando la voce *New* del menù *Object*.

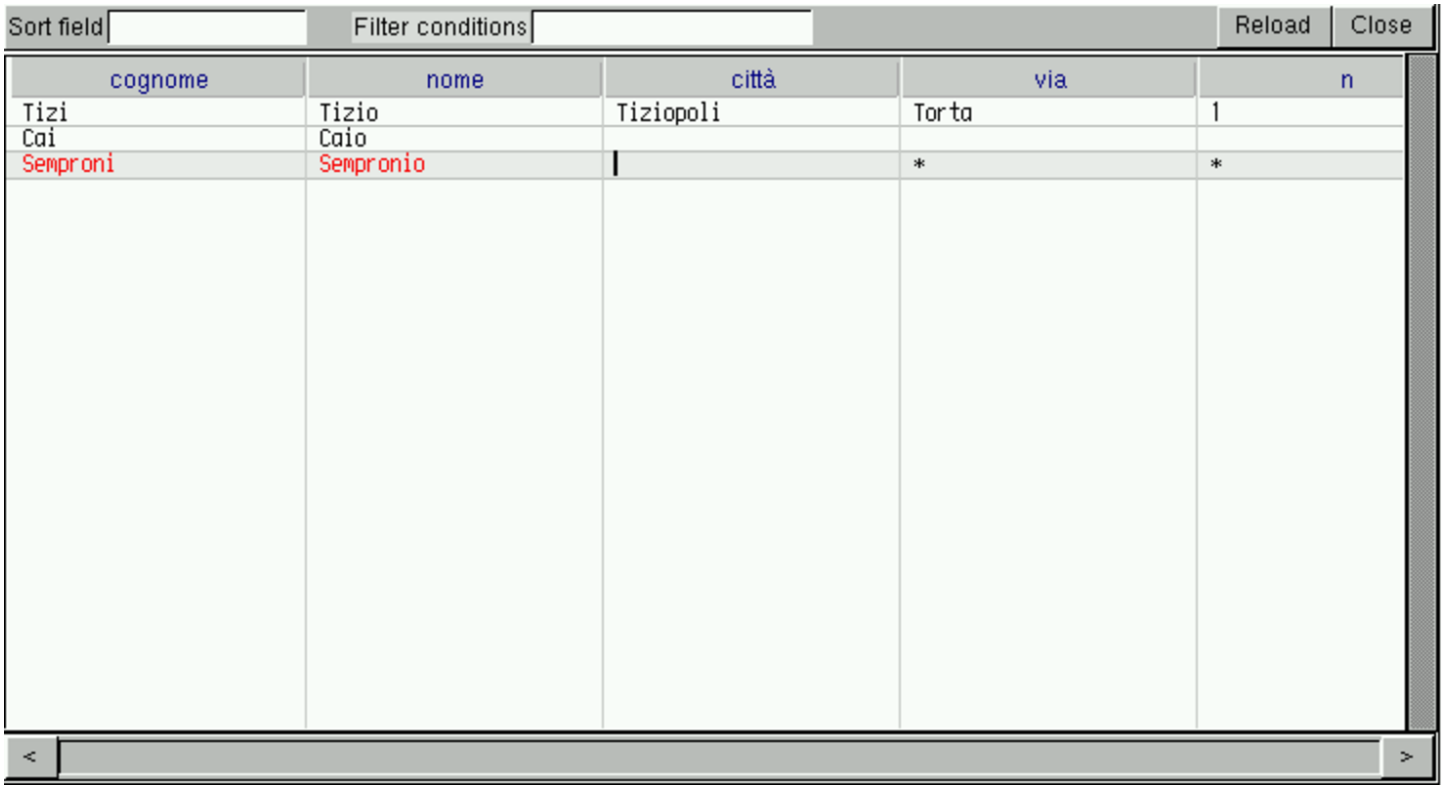
Figura 75.98. Finestra per la creazione di una relazione.



Una volta creata la relazione (si ottiene questo confermando con il pulsante grafico CREATE TABLE), il suo nome appare nella parte

centrale della finestra principale del programma; per accedere al suo contenuto basta selezionare la voce *Open* dal menù Object, ottenendo così una tabella di scorrimento con la quale si possono aggiungere e modificare righe preesistenti. La figura 75.99 mostra l'inserimento di alcuni nomi. Si osservi in particolare il fatto che, eventualmente, si può richiedere espressamente l'aggiunta di una riga nuova premendo il terzo tasto del mouse.

Figura 75.99. Finestra per lo scorrimento del contenuto di una relazione.



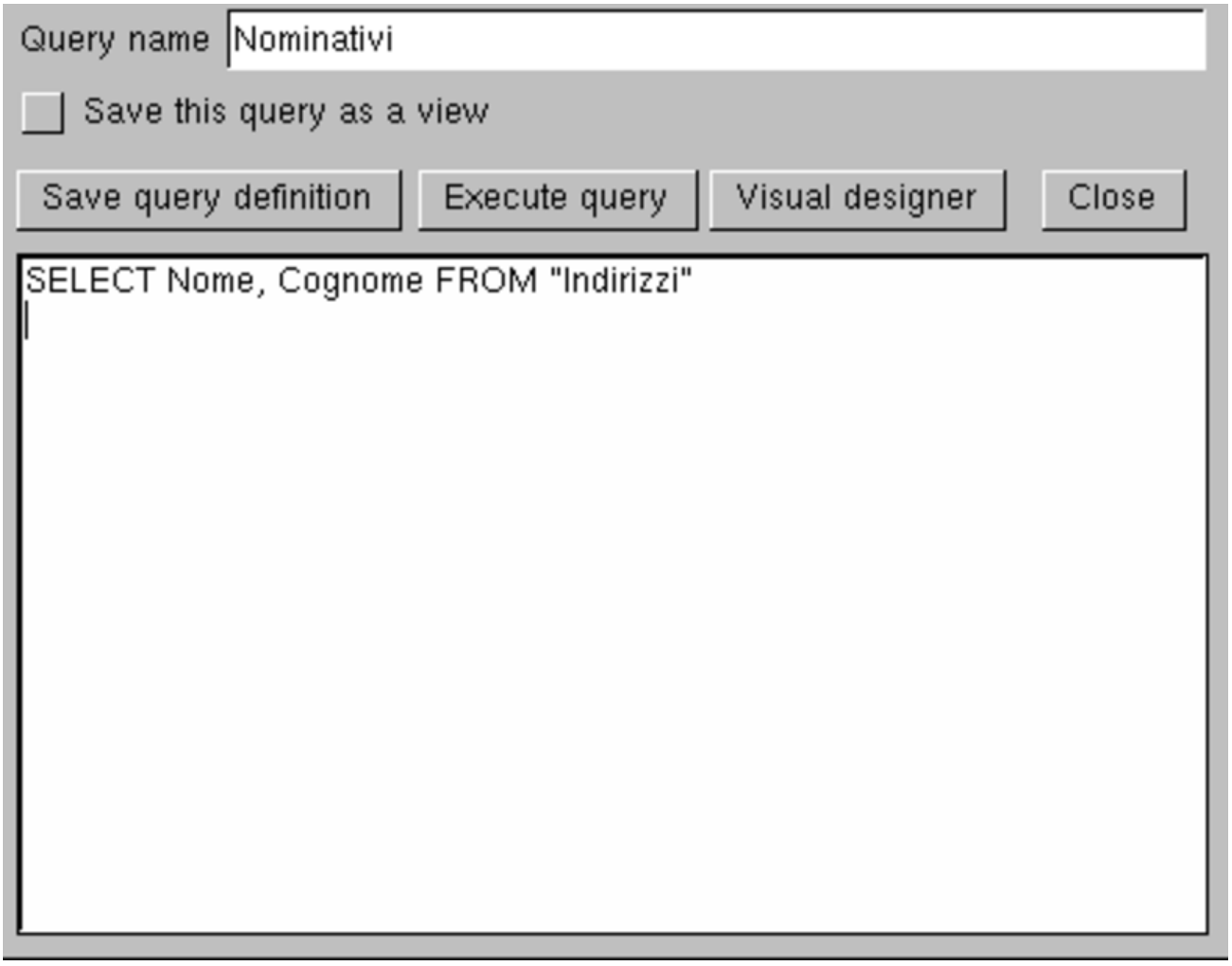
Vale la pena di osservare che la maschera di scorrimento e inserimento dati nella relazione, permette di leggere le tuple in ordine, in base a un certo attributo, filtrando eventualmente le tuple in base a una condizione. Si stabilisce questo mettendo il nome di un attributo nella casella '**`Sort field`**' e mettendo l'espressione della condizione di filtro nella casella '**`Filter conditions`**': se poi si seleziona il pulsante grafico RELOAD (che comunque appare come un'icona),

si riottiene il contenuto ordinato e filtrato in base alle preferenze indicate.

75.4.4 Interrogazioni e viste

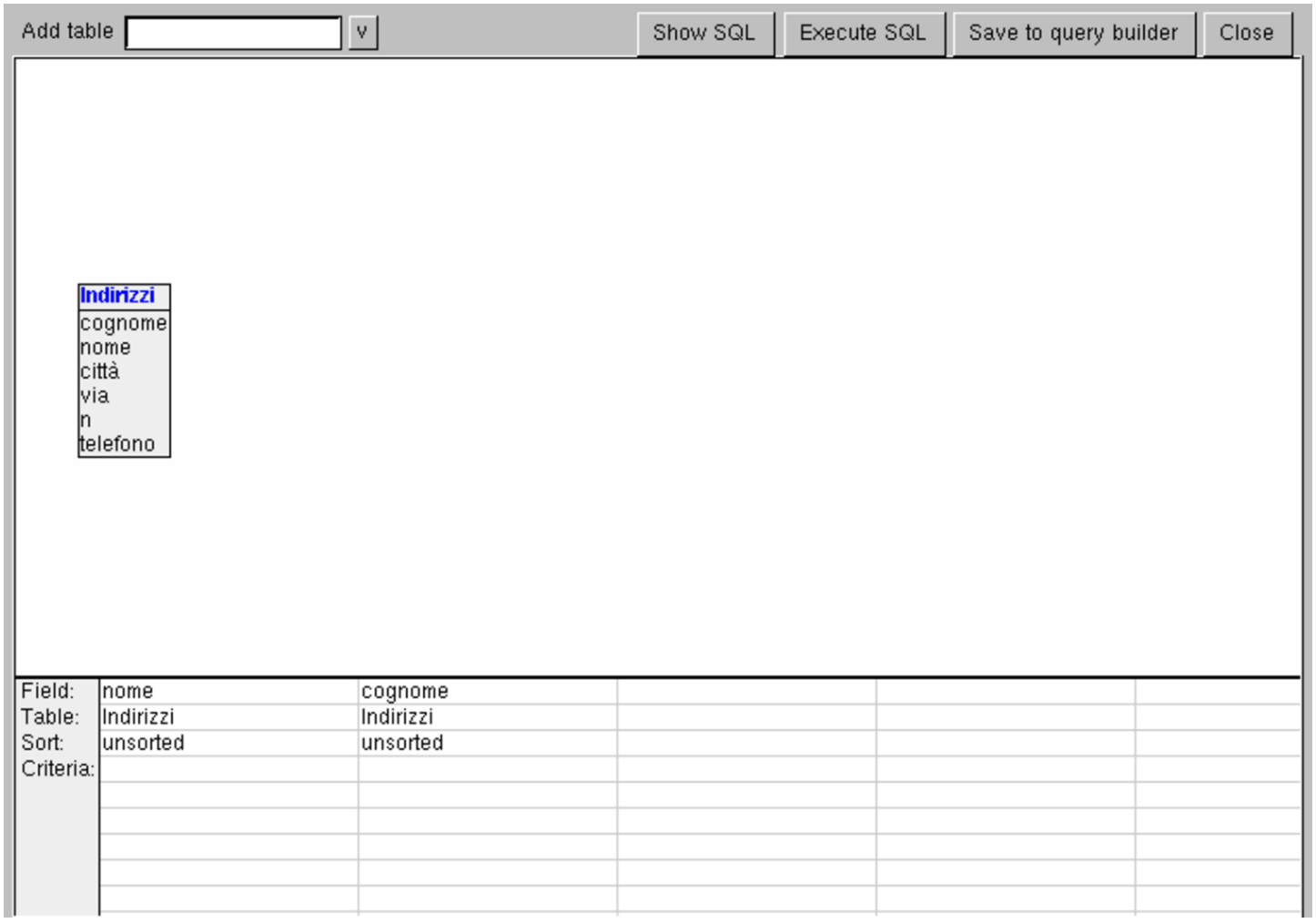
È possibile realizzare facilmente dei modelli di interrogazione e delle viste, attraverso la selezione delle voci *Queries* e *Views* (nella parte sinistra della finestra, sotto alla voce *Tables*). Nel primo caso si tratta di interrogazioni SQL che vengono memorizzate da PgAccess e richiamate a piacere, mentre nel secondo si tratta di viste vere e proprie. A livello operativo, con PgAccess le due cose sono praticamente identiche, per cui si passa generalmente per la creazione di un'interrogazione SQL che poi, eventualmente, si salva come vista. La figura 75.100 mostra la definizione dell'interrogazione '**`Nominativi`**', abbinata al comando '**`SELECT Cognome, Nome FROM "Indirizzi"`**', scritto manualmente dall'utilizzatore.

Figura 75.100. Finestra per la creazione di un'interrogazione.



Nella figura si può osservare che è disponibile una casella di selezione attraverso la quale si può richiedere di salvare come vista. In particolare, con il pulsante grafico SAVE QUERY DEFINITION si salva il modello dell'interrogazione, con il nome fissato in alto; ma volendo, con il pulsante grafico VISUAL DESIGNER, si accede a una maschera per la definizione grafica dell'interrogazione, come si vede nella figura 75.101.

Figura 75.101. Finestra per la creazione visuale di un'interrogazione.



In alto appare una casella in cui si deve indicare il nome di una relazione da cui si vogliono prelevare i campi; una volta fatto, appare un riepilogo di questi campi, in un riquadro. Questi nomi possono essere trascinati con il puntatore del mouse, in basso, dove vengono elencati i campi da includere nell'interrogazione; se si sbaglia, gli elementi che si vogliono togliere possono essere cancellati premendo il tasto [*Canc*] ([*Del*] nelle tastiere inglesi). Nella figura mostrata, sono già stati trascinati e depositati i campi del nome e del cognome.

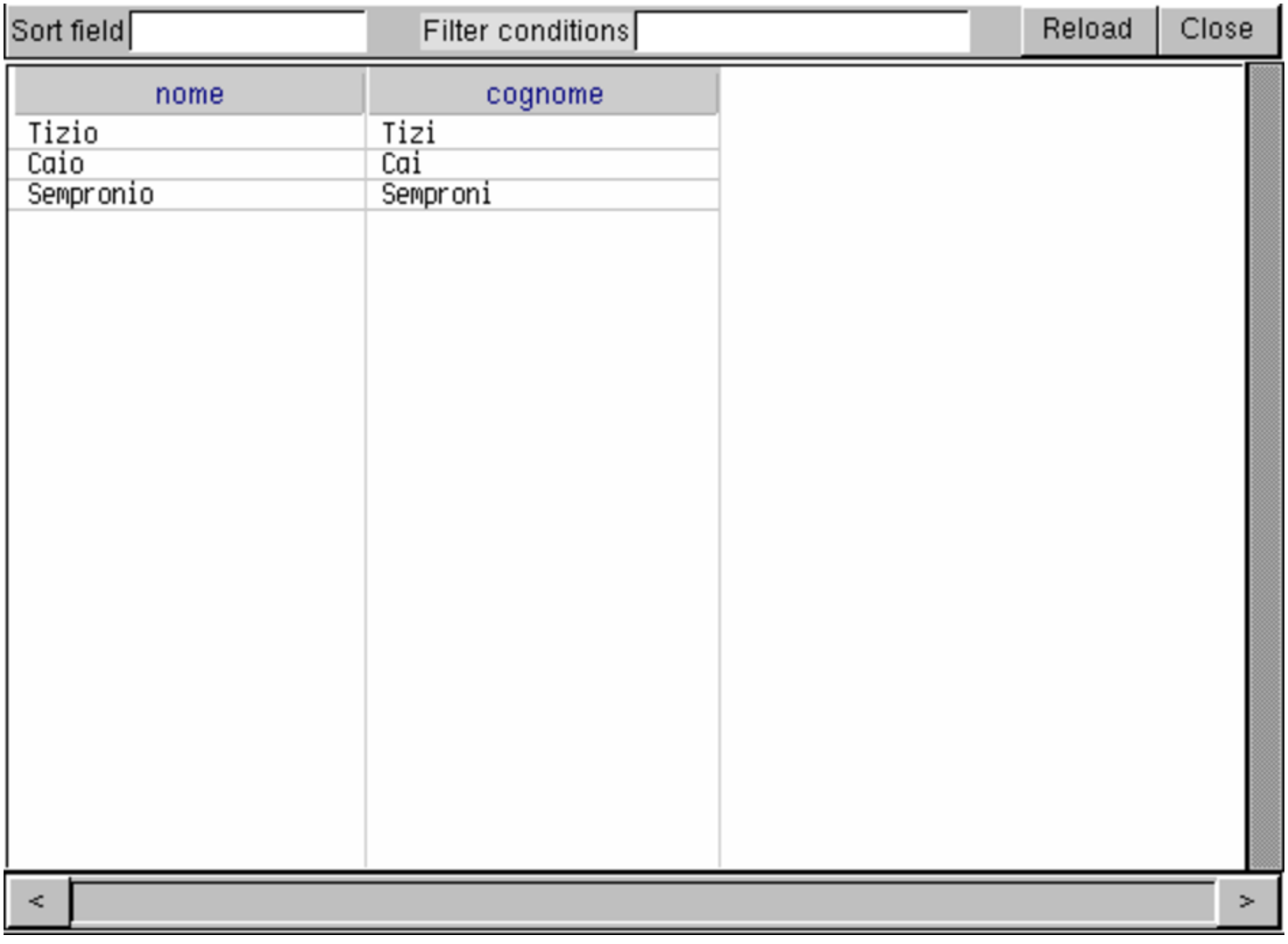
Al termine, se si è soddisfatti del risultato, si può confermare con

il pulsante grafico SAVE TO QUERY BUILDER, ritrovando poi nella finestra precedente l'interrogazione corrispondente alle scelte fatte, che può essere ritoccata a mano se lo si desidera. Nel caso dell'esempio mostrato, l'interrogazione SQL che si ottiene è:

```
select t0.nome, t0.cognome from "Indirizzi" t0
```

L'apertura di un'interrogazione o di una vista, genera lo scorrimento del risultato dell'interrogazione, oppure della vista, come si vede nella figura 75.103 che fa sempre riferimento agli esempi precedenti.

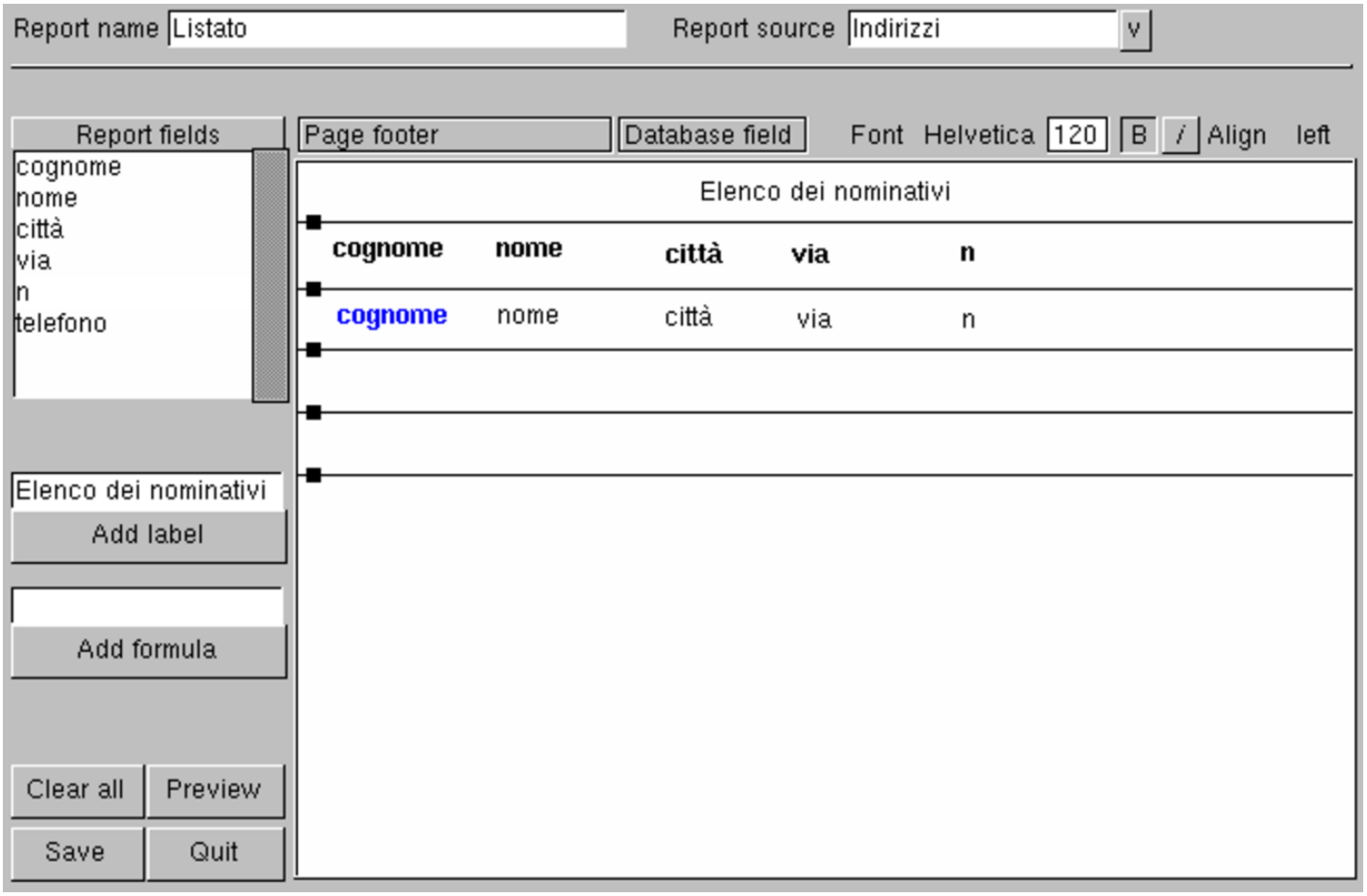
Figura 75.103. Scorrimento di una vista.



75.4.5 Stampe

Con PgAccess è possibile definire anche delle stampe, nel senso di rapporti stampati contenenti il risultato di un'interrogazione SQL. La figura 75.104 mostra la finestra che si utilizza per questo scopo, dove è già iniziata la compilazione dello schema di stampa.

Figura 75.104. Creazione di un tabulato.



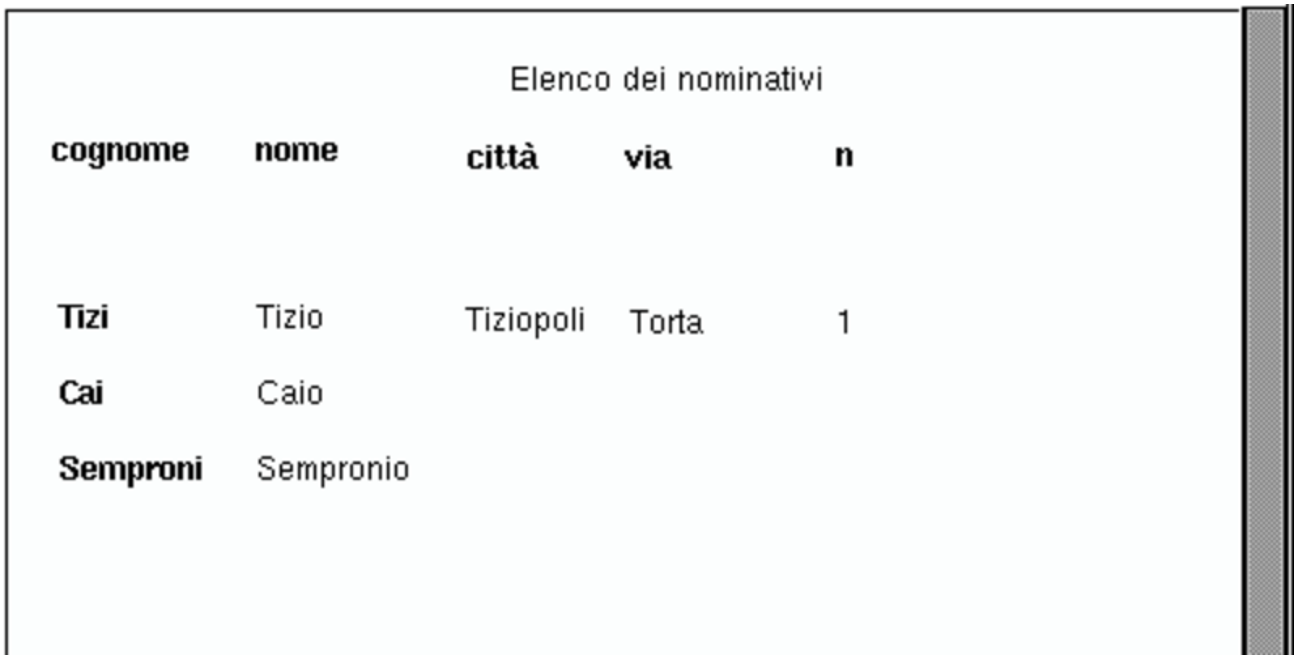
Una volta selezionata la relazione da cui prelevare i campi, dopo aver indicato il nome del tabulato che si vuole generare, basta fare un clic con il tasto sinistro del mouse mentre si punta sul nome del campo che si vuole inserire sullo schema di destra (che rappresenta il modello della stampa). Una volta che sono apparsi i nomi nello spazio a destra, questi possono essere trascinati dove si vuole, even-

tualmente possono anche essere cancellati usando il tasto [*Canc*]. Nell'esempio della figura, si vede anche che è stato inserito un titolo.

Spostando il puntatore del mouse sullo spazio che rappresenta lo schema di stampa, si vede cambiare la sua descrizione in alto. Nella figura mostrata viene indicato '**`Page footer`**', perché in quel momento il puntatore del mouse era nella penultima riga di quello schema.

Per verificare il risultato, è disponibile anche un'anteprima, che si ottiene selezionando il pulsante grafico PREVIEW. Seguendo gli esempi precedenti, la figura 75.105 mostra questa anteprima. Da lì si può passare alla stampa, che però potrebbe limitarsi a generare un file PostScript.

Figura 75.105. Anteprima di stampa.



75.5 Accesso attraverso WWW-SQL

WWW-SQL [3] è un programma CGI in grado di creare pagine HTML a partire dalle informazioni ottenute da una base di dati PostgreSQL o MySQL. In questo capitolo si vuole vedere in particolare l'interazione rispetto alle basi di dati di PostgreSQL. In ogni caso, per poter leggere questo capitolo, occorre sapere cosa sia un programma CGI e come interagisce con un servente HTTP, come spiegato nel capitolo 40.

È molto probabile che la propria distribuzione GNU abbia organizzato due pacchetti distinti, in base all'uso che se ne intende fare, per l'abbinamento con PostgreSQL, oppure con MySQL. In questo modo, il nome del programma CGI a cui si deve fare riferimento può cambiare leggermente, anche da una distribuzione all'altra. Qui si fa riferimento al nome '**`www-pgsql`**' per quello che riguarda l'uso con PostgreSQL.

75.5.1 Principio di funzionamento

Ammesso che il pacchetto organizzato dalla propria distribuzione sia stato realizzato nel modo corretto, l'eseguibile '**`www-pgsql`**' dovrebbe trovarsi nella directory più adatta per i programmi CGI, ovvero quella a cui si accede normalmente con l'URI *http://localhost/cgi-bin/*. In tal caso, per accedere a questo programma, basta avviare il proprio navigatore preferito e puntare sull'indirizzo *http://localhost/cgi-bin/www-pgsql*. Ma non basta, dal momento che il programma in questione ha bisogno di interpretare un file HTML speciale dal quale restituisce poi un risultato. Per capire come funziona la cosa, prima ancora di avere affrontato lo studio del linguaggio specifico di WWW-SQL, si può provare con un file HTML normale:

si supponga di avere a disposizione il file ‘`http://localhost/index.html`’; per fare in modo che WWW-SQL lo analizzi, basta indicare l’URI *http://localhost/cgi-bin/www-pgsql/index.html*. Il risultato è identico all’originale, ma per arrivare a questo si passa attraverso l’elaborazione del programma CGI, dimostrando così il suo funzionamento.

Volendo, se il proprio programma servente HTTP è Apache, è possibile rendere la cosa più elegante attraverso una configurazione opportuna del file ‘`srm.conf`’. Per esempio si potrebbe fare in modo che i file che terminano con l’estensione ‘`.pgsql`’ vengano elaborati automaticamente attraverso il programma CGI in questione:

```
Action www-pgsql /cgi-bin/www-pgsql
AddHandler www-pgsql pgsql
```

Tuttavia, occorre considerare che alcune installazioni di Apache sono state predisposte in modo da impedire l’utilizzazione dell’istruzione ‘**`Action`**’. Se dopo le modifiche di questo file, il servizio di Apache non si riavvia, ciò potrebbe essere un sintomo di questo problema.

75.5.2 Preparazione delle basi di dati e accesso

«

Perché il programma CGI possa accedere alle basi di dati di PostgreSQL, occorre ricordare di predisporre gli utenti e i permessi necessari all’interno della gestione delle basi di dati stesse. Potrebbe essere conveniente prevedere la possibilità di accesso per l’utente di sistema usato dal processo elaborativo del servente HTTP, quando esegue i programmi CGI, in modo da semplificare l’istruzione necessaria alla connessione. Supponendo che si tratti dell’utente ‘**`www-cgi`**’,

volendo procedere in questo modo, occorre aggiungere tale utente, con lo stesso nome, nel sistema di PostgreSQL:

```
postgres$ createuser www-cgi [Invio]
```

Quindi occorre intervenire nelle basi di dati regolando i permessi attraverso i comandi '**GRANT**' e '**REVOKE**', tenendo conto che a questo proposito si può consultare quanto già spiegato nella sezione 75.1. Per fare un esempio, volendo concedere l'accesso in lettura alla relazione '**Indirizzi**', della base di dati '**anagrafe**', all'utente '**www-cgi**', si potrebbe agire come si vede di seguito:

```
postgres$ psql anagrafe [Invio]

anagrafe=> GRANT SELECT ON Indirizzi TO www-cgi; [Invio]
```

75.5.3 Linguaggio di WWW-SQL

WWW-SQL interpreta un file HTML alla ricerca di istruzioni secondo il formato schematizzato di seguito:

```
<! SQL comando [argomento...] >
```

Come si vede, queste istruzioni assomigliano a dei commenti per l'HTML, ma anche se non lo sono realmente, di solito i navigatori ignorano dei marcatori di questo tipo. Tuttavia, questa si può considerare solo come una misura di sicurezza, dal momento che questi file non dovrebbero essere raggiunti direttamente, ma solo attraverso l'intermediazione di WWW-SQL.

Le istruzioni di WWW-SQL rappresentano un linguaggio di programmazione, semplice, ma efficace per lo scopo che ci si prefigge. Si osservi che il «comando» è una parola chiave che rappresen-

ta il tipo di azione che si intende svolgere; inoltre, gli argomenti possono essere presenti o meno, in funzione del comando. Gli argomenti di un comando possono essere racchiusi tra apici doppi (‘"…"’): all’interno di queste stringhe si possono indicare delle variabili da espandere e si possono usare anche delle sequenze di escape per rappresentare simboli speciali che altrimenti avrebbero un altro significato.

Le parole chiave che costituiscono le istruzioni di WWW-SQL possono essere scritte indipendentemente utilizzando lettere maiuscole o minuscole. Inoltre, lo spazio dopo il delimitatore iniziale ‘**<!**’ e lo spazio prima del delimitatore finale ‘**>**’ sono facoltativi.

Per iniziare subito con un esempio che faccia capire la logica di funzionamento di WWW-SQL, si osservi il «programma» seguente, rappresentato dal file ‘`variabili.pgsql`’:

```
<HTML>
<HEAD>
    <TITLE>Esempio sul funzionamento delle
           variabili con WWW-SQL</TITLE>
</HEAD>
<BODY>
<H1>Esempio sul funzionamento delle variabili
    con WWW-SQL</H1>

<P><! SQL PRINT "var = $var" ></P>

<FORM ACTION="variabili.pgsql" METHOD="GET">
    <P><INPUT NAME="var">
    <INPUT TYPE="submit">
</FORM>
```

```
</BODY>
```

L'unica istruzione per WWW-SQL è '**<!SQL PRINT**...**>**', con la quale si vuole ottenere la visualizzazione di una stringa tra apici doppi. Si osservi che '**$var**' è il riferimento alla variabile '**var**', che viene espanso, come parte della valutazione della stringa.

Come si può intuire leggendo l'esempio, i campi definiti attraverso i modelli (gli elementi '**FORM**'), si traducono in variabili per WWW-SQL.

Per verificare il funzionamento di questo programma, supponendo di avere collocato il file '`variabili.pgsql`' nella directory iniziale dei documenti HTML offerti dal servente HTTP, basta puntare il navigatore sull'indirizzo *http://localhost/cgi-bin/www-pgsql/variabili.pgsql* (sempre ammettendo che l'indirizzo *http://localhost/cgi-bin/www-pgsql* corrisponda all'avvio del programma CGI che costituisce in pratica WWW-SQL).

Quello che si ottiene dovrebbe essere un modulo HTML molto semplice, dove si può inserire un testo. Inviando il modulo compilato, dovrebbe essere restituito lo stesso modulo, con la stringa iniziale aggiornata, dove viene mostrato che è stato recepito il dato inserito (nella figura 75.108 si vede che è stata inviata la stringa «Saluti».

Figura 75.108. Risultato dell'interpretazione del file '`variabili.pgsql`' attraverso WWW-SQL.

I sorgenti di WWW-SQL possono essere compilati in modo differente. In particolare, si può distinguere tra due tipi di scansione: il tipo vecchio non permette l'uso di istruzioni che prevedono un'iterazione. In pratica, in quel caso, non funzionano i cicli iterativi e gli altri comandi correlati.

75.5.3.1 Espressioni

«

Si distinguono due tipi di espressioni che si possono valutare all'interno delle istruzioni di WWW-SQL: quelle che si applicano ai valori numerici e quelle che si applicano alle stringhe. Le tabelle 75.109 e 75.110 elencano gli operatori che possono essere utilizzati a questo proposito. Si osservi in particolare l'operatore '`:`', che permette di fare un confronto tra una stringa e un'espressione regolare.

Tabella 75.109. Elenco degli operatori utilizzabili con operandi numerici.

Operatore e operandi	Descrizione
+op	Non ha alcun effetto.
−op	Inverte il segno dell'operando.
op1 + *op2*	Somma i due operandi.
op1 − *op2*	Sottrae dal primo il secondo operando.
op1 * *op2*	Moltiplica i due operandi.
op1 / *op2*	Divide il primo operando per il secondo.
op1 % *op2*	Modulo: il resto della divisione tra il primo e il secondo operando.
op1 ^ *op2*	Eleva il primo operando alla potenza del secondo.
op1 == *op2*	*Vero* se gli operandi sono uguali.
op1 = *op2*	*Vero* se gli operandi sono uguali (sinonimo di '**==**').
op1 != *op2*	*Vero* se gli operandi sono differenti.
op1 > *op2*	*Vero* se il primo operando è maggiore del secondo.
op1 < *op2*	*Vero* se il primo operando è minore del secondo.
op1 >= *op2*	*Vero* se il primo operando è maggiore o uguale al secondo.

Operatore e operandi	Descrizione
op1 `<=` ***op2***	*Vero* se il primo operando è minore o uguale al secondo.
`!` ***op***	Negazione logica.
op1 `&&` ***op2***	AND logico.
op1 `&` ***op2***	AND logico (sinonimo di '**`&&`**').
op1 `||` ***op2***	OR logico.
op1 `|` ***op2***	OR logico (sinonimo di '**`||`**').

Tabella 75.110. Elenco degli operatori utilizzabili con operandi di tipo stringa.

Operatore e operandi	Descrizione
op1 `==` ***op2***	*Vero* se gli operandi sono uguali.
op1 `!=` ***op2***	*Vero* se gli operandi sono differenti.
op1 `>` ***op2***	*Vero* se il primo operando è lessicograficamente successivo al secondo.
op1 `<` ***op2***	*Vero* se il primo operando è lessicograficamente precedente al secondo.
op1 `>=` ***op2***	*Vero* se il primo operando non è lessicograficamente precedente al secondo.
op1 `<=` ***op2***	*Vero* se il primo operando non è lessicograficamente successivo al secondo.
str `:` ***regexp***	*Vero* se l'espressione regolare corrisponde alla stringa.

All'interno delle stringhe è prevista l'espansione di variabili e sono anche riconosciute alcune sequenze di escape (tabella 75.111). Le variabili in questione vanno intese come parte del linguaggio di WWW-SQL; alcune di queste sono la ripetizione di variabili di ambiente corrispondenti, altre sono variabili interne del programma (come elencato nella tabella 75.112), altre ancora possono essere definite all'interno del «programma» stesso, o meglio ancora, attraverso dei moduli, come è stato mostrato nell'esempio iniziale. Le variabili vengono riconosciute in quanto scritte secondo lo schema seguente:

prefisso nome_della_variabile

Il prefisso è un simbolo a scelta tra: '**$**', '**@**', '**?**', '**#**'. In pratica, '**$var**', '**@var**', '**?var**', e '**#var**', sono riferimenti identici alla stessa variabile '**var**'. Per questo motivo, se si vogliono usare i simboli corrispondenti a questi prefissi in modo letterale, occorre usare una sequenza di escape.

Tabella 75.111. Sequenze di escape utilizzabili all'interno delle stringhe.

Escape	Significato
\\	\
\"	"
\n	<*LF*>
\t	<*HT*> (tabulazione)

Escape	Significato
`\$`	$
`\@`	@
`\#`	#
`\?`	?
`\~`	~

Tabella 75.112. Variabili interne di WWW-SQL.

Variabile	Descrizione
`AFFECTED_ROWS`	Numero di righe coinvolte dall'ultima interrogazione.
`NUM_FIELDS`	Numero di campi restituiti dall'ultima interrogazione.
`NUM_ROWS`	Numero di righe restituiti dall'ultima interrogazione.
`WWW_SQL_VERSION`	Versione di WWW-SQL.
`GATEWAY_INTERFACE`	Versione dell'interfaccia CGI.
`HOSTTYPE`	Tipo di macchina del servente HTTP.
`HTTPHOST`	Nome del nodo servente.
`HTTP_REFERER`	Pagina da cui proviene il cliente.
`HTTP_USER_AGENT`	Nome del programma di navigazione (cliente).

Variabile	Descrizione
`OSTYPE`	Nome del sistema operativo del servente.
`PATH_INFO`	Percorso relativo dello script attuale.
`PATH_TRANSLATED`	Percorso assoluto del file corrispondente allo script attuale.
`REMOTE_ADDR`	Indirizzo del nodo remoto.
`REMOTE_HOST`	Nome del nodo remoto.
`SERVER_ADMIN`	Indirizzo di posta elettronica dell'amministratore.
`SERVER_NAME`	Nome del servente.
`SERVER_PORT`	Numero della porta utilizzata per la connessione con il servente.
`SERVER_PROTOCOL`	Nome e versione del protocollo (HTTP).
`SERVER_SOFTWARE`	Nome del software usato come servente HTTP.
`SCRIPT_FILENAME`	Percorso del programma CGI (l'eseguibile di WWW-SQL).
`SCRIPT_NAME`	Percorso relativo del programma CGI (l'eseguibile di WWW-SQL).
`REQUEST_URI`	Indirizzo richiesto.

Per prendere confidenza con le variabili interne di WWW-SQL, si può realizzare lo script seguente ('`interne.pgsql`'), che con l'istruzione '**`<!SQL DUMPVARS>`**' le elenca tutte. La figura 75.114 mostra il risultato che si potrebbe ottenere.

```
<HTML>
<HEAD>
    <title>Visualizzazione delle variabili interne</title>
</HEAD>
<BODY>
<H1>Visualizzazione delle variabili interne</H1>

<! SQL DUMPVARS >

</BODY>
```

Figura 75.114. Esempio del contenuto delle variabili interne attraverso l'istruzione '**<!SQL DUMPVARS>**'.

```
          Visualizzazione delle variabili interne

WWW_SQL_VERSION = 0.5.5
SERVER_SOFTWARE = Apache/1.3.3 (Unix) Debian/GNU
SERVER_PROTOCOL = HTTP/1.0
SERVER_PORT = 80
SERVER_NAME = dinkel.brot.dg
SERVER_ADMIN = webmaster@dinkel.brot.dg
SCRIPT_FILENAME = /usr/lib/cgi-bin/www-pgsql
SCRIPT_NAME = /cgi-bin/www-pgsql
REQUEST_URI = /cgi-bin/www-pgsql/interne.pgsql
REMOTE_ADDR = 127.0.0.1
QUERY_STRING =
PATH_TRANSLATED = /var/www/interne.pgsql
PATH_INFO = /interne.pgsql
HTTP_USER_AGENT = Lynx/2.8.1rel.2 libwww-FM/2.14
HTTP_HOST = localhost
GATEWAY_INTERFACE = CGI/1.1
DOCUMENT_ROOT = /var/www
```

75.5.3.2 Strutture di controllo

Attraverso le istruzioni di WWW-SQL, si possono realizzare le strutture di controllo che sono comuni nei linguaggi di programmazione. È prevista la struttura condizionale e il ciclo iterativo.

```
<! SQL IF espressione >
    ...
[<! SQL ELSIF espressione >]
    ...
[<! SQL ELSE >]
    ...
<! SQL ENDIF >
```

La struttura condizionale che si vede nello schema, permette di delimitare uno spazio da filtrare in base all'esito delle espressioni condizionali coinvolte. Si osservi l'esempio seguente:

```
<! SQL IF $NUM_ROWS == 10 >
    <P>Il numero delle righe è uguale a 10.</P>
<! SQL ELSE >
    <P>Il numero delle righe non corrisponde a
    quanto previsto.</P>
<! SQL ENDIF >
```

In questo modo si condiziona la visualizzazione di una frase in base al fatto che la variabile '**NUM_ROWS**' contenga o meno il valore 10.

È importante osservare che l'espressione usata come condizione di controllo potrebbe restituire un risultato numerico e non logico. In tal caso, lo zero corrisponde a *Falso*, mentre qualunque altro valore corrisponde a *Vero*.

```
<! SQL WHILE espressione >
    ...
<! SQL DONE >
```

La struttura iterativa che si vede nello schema, permette di delimitare uno spazio da interpretare ripetitivamente, finché l'espressione condizionale introduttiva continua a restituire il valore *Vero* (o un valore numerico diverso da zero).

```
<! SQL SET contatore 10 >
<! SQL WHILE $contatore > 0 >
    <P>Il contatore ha raggiunto il livello
       <! SQL PRINT "$contatore" >.</P>
    <! SQL SETEXPR contatore $contatore - 1 >
<! SQL DONE >
```

L'esempio mostra l'inizializzazione di una variabile, denominata '**contatore**', al valore iniziale 10; quindi inizia un ciclo iterativo che si arresta quando tale variabile raggiunge lo zero. A ogni ciclo, viene visualizzato il contenuto della variabile, che subito dopo viene ridotto di un'unità.

Se l'istruzione '**<!SQL WHILE**...**>**' non viene riconosciuta, significa che non è disponibile la scansione iterativa.

Nell'ambito di un'iterazione, possono essere usate delle istruzioni per interrompere il ciclo in corso o per interrompere tutta l'iterazione:

```
<! SQL CONTINUE >
```

```
<! SQL BREAK >
```

La prima della due istruzioni interrompe il ciclo attuale, facendo riprendere immediatamente l'iterazione, mentre il secondo interrompe l'iterazione del tutto.

Esiste anche un altro tipo di iterazione, il cui scopo è la scansione delle righe ottenute dall'interrogazione di una base di dati:

```
<! SQL PRINT_LOOP riferimento_all'interrogazione >
    ...
<! SQL DONE >
```

Anche all'interno di questa struttura si possono usare le istruzioni '**<!SQL CONTINUE>**' e '**<!SQL BREAK>**'.

75.5.4 Istruzioni

Le istruzioni «normali» di WWW-SQL, ovvero quelle che non servono a descrivere delle strutture di controllo, sono descritte in questa sezione e in quelle seguenti. In particolare si può notare che WWW-SQL offre delle istruzioni per la lettura semplificata dell'esito di un'interrogazione SQL e altre per la lettura dettagliata, fino ad arrivare a distinguere tupla per tupla e attributo per attributo.

È importante chiarire che, anche se un'«interrogazione» serve principalmente per leggere dati da una relazione di una base di dati, nello stesso modo, attraverso WWW-SQL si potrebbero fare delle modifiche ai dati.

Segue un elenco di istruzioni di tipo vario, mentre nelle sezioni seguenti vengono raccolte altre istruzioni più specifiche.

- Emissione di una stringa con espansione di variabili:

```
<! SQL PRINT stringa >
```

  L'istruzione '**<!SQL PRINT** ...**>**' permette di emettere una stringa. Dal momento che un file HTML non ha bisogno di accorgimenti particolari per mostrare una stringa costante, è evidente che il senso di questa istruzione sta nella possibilità di indicare delle variabili da espandere, come nell'esempio seguente:

```
<P>Il contatore ha raggiunto il livello
   <! SQL PRINT "$contatore" >.</P>
```

- Risultato di un'espressione:

```
<! SQL EVAL espressione >
```

  L'istruzione '**<!SQL EVAL** ...**>**' è simile a '**<!SQL PRINT** ...**>**', con la differenza che l'argomento non è più una stringa, ma un'espressione differente, il cui risultato viene emesso alla fine.

- Impostazione di una variabile:

```
<! SQL SET nome_variabile valore_da_assegnare >
```

L'istruzione '**<!SQL SET ...>**' permette di definire e inizializzare una variabile. L'esempio seguente definisce la variabile '**contatore**', inizializzandola a zero:

```
<! SQL SET contatore 0 >
```

- Impostazione di una variabile attraverso un'espressione:

```
<! SQL SETEXPR nome_variabile espressione >
```

L'istruzione '**<!SQL SETEXPR ...>**' permette di definire e inizializzare una variabile; in particolare, il valore che si assegna può essere il risultato della valutazione di un'espressione. L'esempio seguente definisce la variabile '**contatore**', inizializzandola con il risultato dell'espressione '**$contatore - 1**'. In pratica viene decrementato il contenuto della variabile '**contatore**':

```
<! SQL SETEXPR contatore $contatore - 1 >
```

- Definizione di un valore predefinito per il contenuto di una variabile:

```
<! SQL SETDEFAULT nome_variabile valore_da_assegnare >
```

L'istruzione '**<!SQL SETDEFAULT ...>**' permette di stabilire un valore predefinito per una variabile; a differenza di '**<!SQL SET ...>**' la variabile non viene modificata se esiste già e ha un valore. L'esempio seguente definisce la variabile '**contatore**', solo se necessario, inizializzandola con il valore 10:

```
<! SQL SETDEFAULT contatore 10 >
```

- Elenco variabili:

```
<! SQL DUMPVARS >
```

L'istruzione '**<!SQL DUMPVARS>**' emette l'elenco delle variabili esistenti, assieme al valore che contengono. Può essere usato per scopo diagnostico, quando si cerca di capire cosa succede realmente.

75.5.4.1 Apertura e chiusura di una connessione, e accesso a una base di dati

«

L'interrogazione di una base di dati deve essere preceduta dalla connessione a un servente DBMS e dalla selezione di una base di dati; inoltre, al termine delle interrogazioni, si passa normalmente alla chiusura di una connessione, in pratica secondo lo schema seguente:

```
<! SQL CONNECT … >
<! SQL DATABASE nome_della_base_di_dati >
…
…
<! SQL CLOSE >
```

In breve: '**<!SQL CONNECT** ...**>**' serve a iniziare una connessione con un servente per l'accesso a una base di dati; '**<!SQL DATABASE** ...**>**' serve a indicare la base di dati specifica presso il servente; '**<!SQL CLOSE>**' chiude la connessione.

- Accesso a un servente DBMS:

```
<! SQL CONNECT [nodo [utente [parola_d'ordine] ] ] >
```

L'istruzione '**<!SQL CONNECT ...>**' permette di iniziare una connessione con un DBMS. Dipende dal DBMS stesso se è possibile accedere senza alcun sistema di autenticazione. In generale, se non si indica il nodo a cui accedere, si intende *localhost*; inoltre, se non si indica l'utente, si fa riferimento al numero UID con il quale funziona il programma servente del servizio HTTP (che a sua volta avvia il programma CGI). L'esempio che segue richiede di connettersi al servente DBMS PostgreSQL che opera nello stesso elaboratore locale, utilizzando l'identità dell'utente '**pgnanouser**' e senza specificare alcuna parola d'ordine:

```
<! SQL CONNECT localhost pgnanouser >
```

- Selezione di una base di dati specifica:

```
<! SQL DATABASE nome_base_di_dati >
```

L'istruzione '**<!SQL DATABASE ...>**' permette di aprire una base di dati specifica; per la precisione, utilizzando PostgreSQL, l'accesso al servente avviene solo dopo che è stata specificata la base di dati.

- Chiusura di una connessione:

```
<! SQL CLOSE >
```

La chiusura di una connessione (e quindi anche di una base di dati aperta), si ottiene con l'istruzione '**<!SQL CLOSE>**'.

Prima di passare alla descrizione delle istruzioni che permettono l'interrogazione del contenuto di una base di dati, viene mostrato un esempio che si limita a elencare la relazione '**Indirizzi**' della base di dati '**anagrafe**':

```
<HTML>
<HEAD>
    <TITLE>Esempio di interrogazione</TITLE>
</HEAD>
<BODY>
<H1>Esempio di interrogazione</H1>
<! SQL CONNECT localhost nobody >
<! SQL DATABASE anagrafe >
<! SQL QUERY "SELECT * FROM Indirizzi" RICHIESTA_1 >
<! SQL QTABLE RICHIESTA_1 >
<! SQL FREE RICHIESTA_1 >
<! SQL CLOSE >
</BODY>
```

75.5.4.2 Istruzioni di interrogazione normali

«

L'interrogazione di una base di dati avviene attraverso la definizione di un riferimento, che si apre e si chiude come se fosse un flusso di file nei linguaggi di programmazione comuni. Per aprire questo riferimento si inizia con l'invio di un'interrogazione SQL; successivamente è possibile leggere l'esito dell'interrogazione attraverso il riferimento che è stato aperto; infine si passa alla chiusura del riferimento:

```
<! SQL QUERY stringa_di_interrogazione_sql riferimento >
…
…
<! SQL FREE riferimento >
```

- Apertura di un'interrogazione:

```
<! SQL QUERY stringa_di_interrogazione_sql riferimento >
```

L'istruzione '**<!SQL QUERY ...>**' definisce una stringa di interrogazione da inviare al servente DBMS. A questa interrogazione viene abbinato un riferimento costituito da un nome, che in seguito deve essere usato per leggere l'esito dell'interrogazione. Nell'esempio che appare nella sezione precedente, si vedeva l'istruzione seguente con la quale si selezionano tutte le tuple della relazione '**Indirizzi**', abbinando questo risultato al nome '**RICHIESTA_1**':

```
<! SQL QUERY "SELECT * FROM Indirizzi" RICHIESTA_1 >
```

- Tabella rapida:

```
<! SQL QTABLE riferimento [borders] >
```

L'istruzione '**<!SQL QTABLE ...>**' consente di rappresentare rapidamente il risultato di un'interrogazione attraverso una tabella HTML. In particolare, utilizzando la parola chiave '**borders**', la tabella che si genera ha i bordi delle caselle visibili. L'esempio seguente mostra in che modo visualizzare rapidamente il risultato dell'interrogazione abbinata al nome '**RICHIESTA_1**':

```
<! SQL QTABLE RICHIESTA_1 >
```

- Elenco rapido:

```
<! SQL QLONGFORM riferimento >
```

L'istruzione '**<!SQL QLONGFORM ...>**' si utilizza in modo simile a '**<!SQL QTABLE ...>**', per rappresentare il risultato di un'in-

terrogazione attraverso un elenco dettagliato, senza una tabella HTML.

- Chiusura del riferimento all'interrogazione:

```
<! SQL FREE riferimento >
```

Come è stato mostrato all'inizio, l'istruzione '**<!SQL FREE ...>**' serve a chiudere il riferimento a un'interrogazione.

- Realizzazione di un elenco di voci da selezionare:

```
<! SQL QSELECT riferimento variabile_modulo_html >
```

Con l'istruzione '**<!SQL QSELECT ...>**' si ottiene un elenco di voci di un modulo di selezione. In generale, la cosa corrisponde a:

```
<SELECT NAME="variabile_modulo_html">
<! SQL PRINT_ROWS riferimento ↩
↪"<OPTION name=\"@riferimento.0\">riferimento.1">
</SELECT>
```

L'istruzione '**<!SQL PRINT_ROWS ...>**' è descritta nella prossima sezione.

75.5.4.3 Istruzioni per la selezione dettagliata di tuple e attributi

«

È possibile selezionare in maniera più precisa le tuple e gli attributi da ciò che si ottiene da un'interrogazione SQL. Attraverso l'istruzione '**<!SQL FETCH *riferimento*>**' si preleva la tupla attuale dall'interrogazione a cui si fa riferimento. Questo prelievo permette

di fare riferimento agli attributi della tupla attraverso una notazione particolare:

```
@riferimento . n
```

In pratica, è come se fosse l'espansione di una variabile, con la differenza che si indica il nome di un riferimento a un'interrogazione aperta, aggiungendo un'estensione numerica, separata da un punto, dove lo zero corrisponde al primo attributo e $n-1$ corrisponde all'attributo n-esimo.

- Spostamento della tupla attuale all'interno del risultato di un'interrogazione:

  ```
  <! SQL SEEK riferimento n_riga >
  ```

  L'istruzione '**<!SQL SEEK ...>**' permette di spostare la tupla attuale all'interno di un'interrogazione. Per indicare il numero della tupla da raggiungere, occorre tenere presente che lo zero corrisponde alla prima. L'esempio seguente fa in modo che la tupla attuale diventi la seconda del riferimento '**RICHIESTA_1**':

  ```
  <! SQL SEEK RICHIESTA_1 1 >
  ```

- Prelievo della tupla attuale di un certo riferimento:

  ```
  <! SQL FETCH riferimento >
  ```

  L'istruzione '**<!SQL FETCH ...>**' permette di rendere disponibile il contenuto della tupla attuale di un certo riferimento. L'esempio seguente preleva il contenuto della tupla attuale del riferimento '**RICHIESTA_1**'; quindi mostra il primo e il secondo attributo di

questa tupla, che si presume corrispondano al cognome e al nome di una persona:

```
<! SQL FETCH RICHIESTA_1 >
<P>Cognome: <! SQL PRINT "@RICHIESTA_1.0" ></P>
<P>Nome: <! SQL PRINT "@RICHIESTA_1.1" ></P>
```

- Emissione di una stringa per ogni tupla:

```
<! SQL PRINT_ROWS riferimento stringa >
```

L'istruzione '**<!SQL PRINT_ROWS ...>**' è una sorta di istruzione '**<!SQL PRINT ...>**' ripetuta per tutte le tuple di un'interrogazione, a partire da quella corrente. L'esempio seguente mostra la visualizzazione dei primi due attributi di tutte le tuple di un'interrogazione, a cui si fa riferimento con il nome '**Q**':

```
<! SQL SEEK Q 0 >
<! SQL PRINT_ROWS Q "<P>Cognome: @Q.0</P>\n<P>Nome: @Q.1</P>\n" >
```

L'esempio seguente mostra la realizzazione di un modulo per la selezione di un articolo, attraverso l'invio del codice corrispondente. A questo proposito, si suppone che il primo attributo del risultato dell'interrogazione a cui si fa riferimento con il nome '**ELENCO**', corrisponda al codice dell'articolo, mentre il secondo corrisponda a una sua descrizione:

```
<FORM ACTION="ordine.pgsql">
<P><SELECT NAME="codice">
<! SQL PRINT_ROWS ELENCO "<OPTION name=\"@ELENCO.0\">@ELENCO.1" >
</SELECT>
<INPUT TYPE="submit">
</FORM>
```

Dal momento che si fa riferimento alle prime due colonne, la stessa cosa avrebbe potuto essere realizzata con l'istruzione '**<!SQL QSELECT ...>**', nel modo seguente:

```
<FORM ACTION="ordine.pgsql">
<! SQL QSELECT ELENCO codice >
<INPUT TYPE="submit">
</FORM>
```

75.6 Riferimenti

«

- *PostgreSQL*, *http://www.postgresql.org/*
- The PostgreSQL Global Development Group, *PostgreSQL Documentation*, *http://www.postgresql.org/docs/manuals/*
- *PgAccess*, *http://sourceforge.net/projects/pgaccess/*
- James Henstridge, *WWW-SQL*, *http://www.jamesh.id.au/software/www-sql/www-sql.html*

[1] **PostgreSQL** software libero con licenza speciale

[2] **PgAccess** software libero con licenza speciale

[3] **WWW-SQL** GNU GPL

76.1 Struttura e preparazione 2067
76.1.1 Configurazione del servente 2069
76.1.2 Avvio e arresto del servente 2074
76.1.3 Utenze 2077
76.1.4 Accesso comune al servente 2081
76.1.5 Base di dati amministrativa 2086
76.2 Gestione del DBMS 2090
76.2.1 Controllo delle utenze e degli accessi 2090
76.2.2 Amministrazioni varie attraverso il sistema operativo 2098
76.2.3 Ripristino della parola d'ordine dell'amministratore 2099
76.2.4 Archiviazione e recupero delle basi di dati 2100
76.3 Riferimenti 2102

76.1 Struttura e preparazione

MySQL [1] è un DBMS (*Data base management system*) relazionale. Il nome contiene la sigla «SQL», a indicare che si tratta di un DBMS in grado di comprendere le istruzioni SQL.

L'installazione del servente MySQL può essere molto laboriosa, se non si dispone di un pacchetto già pronto per la propria distribuzione

GNU. La descrizione di come installare MySQL viene omessa, perché questa appare nella documentazione di MySQL in modo molto dettagliato. Qui si fa riferimento a una situazione relativamente «comune», a seguito dell'installazione automatica di un pacchetto pronto.

In generale, il servente MySQL è costituito dal demone '**mysqld**', avviato attraverso uno script della procedura di inizializzazione del sistema, che potrebbe corrispondere a '`/etc/init.d/mysql`'. Assieme a questo demone ci sono altri programmi di servizio che servono principalmente per l'amministrazione e la manutenzione delle basi di dati, il più importante dei quali è '**mysqladmin**'.

L'installazione del servente MySQL richiede, nell'ambito del sistema operativo, la preparazione di un utente e un gruppo speciali, entrambi con il nome '**mysql**'. Questa utenza dovrebbe essere già disponibile oppure potrebbe essere creata automaticamente dal sistema di installazione dei pacchetti della propria distribuzione. La directory personale associata a questo utente speciale dovrebbe rappresentare il contenitore dei file che compongono le basi di dati gestite da MySQL: '`~mysql/`'.

Il servente MySQL è sottoposto al controllo di un file di configurazione, che in condizioni normali potrebbe corrispondere a '`/etc/mysql/my.cnf`', oppure solo '`/etc/my.cnf`'. Tuttavia, questo file è suddiviso in sezioni e contiene anche la configurazione relativa ai programmi clienti, mentre una configurazione specifica del solo servente può essere collocata nel file '`~mysql/my.cnf`'.

L'amministratore del DBMS con MySQL si chiama convenzional-

mente ‘**root**’ e si prevede che in un sistema Unix coincida con l’utente ‘**root**’, ma ciò non è strettamente necessario. L’accesso da parte di questo utente, come degli altri, è sottoposto alla presentazione di una parola d’ordine che viene stabilita con l’ausilio di ‘**mysqladmin**’, o attraverso istruzioni SQL appropriate; pertanto, anche se ci può essere una corrispondenza tra utenze di MySQL e utenze del sistema operativo, le parole d’ordine usate non sono collegate tra loro.

MySQL ha una gestione particolare delle proprie utenze, distinguendole in base alla provenienza degli accessi. A seconda del contesto può capitare di utilizzare notazioni del tipo ‘**tizio@dinkel.brot.dg**’, dove si intende fare riferimento all’utente denominato ‘**tizio**’ che accede dal nodo *dinkel.brot.dg*. Deve essere subito chiaro che non si tratta di un indirizzo di posta elettronica e nemmeno di un’utenza Unix. Pertanto, anche l’utente ‘**root**’ (l’amministratore del DBMS), deve essere qualificato meglio e solitamente si fa riferimento a ‘**root@localhost**’ (l’utente ‘**root**’ che accede al servente attraverso lo stesso elaboratore che offre il servizio).

76.1.1 Configurazione del servente

Come accennato, la configurazione generale di MySQL è contenuta nel file ‘`/etc/mysql/my.cnf`’, oppure ‘`/etc/my.cnf`’, a seconda dell’organizzazione della propria distribuzione GNU, con la possibilità di usare il file ‘`~mysql/my.cnf`’ per ciò che riguarda strettamente il funzionamento del servente. Molto probabilmente, il file di configurazione generale è già predisposto per consentire un accesso locale al servente MySQL; di solito può essere utile verificare

o ritoccare solo alcune direttive, specialmente per ciò che riguarda l'abilitazione dell'accesso attraverso la rete.

Il file di configurazione contiene direttive che assomigliano in pratica all'assegnamento di variabili, nella forma seguente:

```
nome = valore
```

Queste direttive sono suddivise in sezioni, dichiarate tra parentesi quadre:

```
[sezione]
direttiva
...
```

È attraverso la presenza di queste sezioni che è possibile distinguere le direttive relative al servente da quelle di altre componenti.

Come spesso accade nei file di configurazione comuni, il simbolo '`#`' introduce un commento, fino alla fine della riga; inoltre, le righe bianche o vuote sono ignorate.

Le direttive che riguardano il funzionamento del servente sono contenute nelle sezioni '`[mysqld_safe]`' e '`[mysqld]`'. Per il momento conviene soffermarsi solo su alcune della seconda di queste sezioni:

```
[mysqld]
user            = mysql
pid-file        = /var/run/mysqld/mysqld.pid
socket          = /var/run/mysqld/mysqld.sock
port            = 3306
log             = /var/log/mysql.log
basedir         = /usr
datadir         = /var/lib/mysql
tmpdir          = /tmp
language        = /usr/share/mysql/english
```

Si comprende intuitivamente il significato di queste direttive: il demone '**`mysqld`**' viene avviato con i privilegi dell'utente '**`mysql`**'; viene usato il file '`/var/run/mysqld/mysqld.pid`' per annotare il numero PID del demone in funzione; viene utilizzato il file '`/var/run/mysqld/mysqld.sock`' come socket di dominio Unix per le comunicazioni locali, oppure la porta 3306 per le comunicazioni attraverso la rete; nel file '`/var/log/mysql.log`' vengono annotate le informazioni relative al suo funzionamento; la directory '`/usr/`' è il punto di partenza dell'installazione dei programmi e di altre componenti; la directory '`/var/lib/mysql/`' è il punto di inizio dei file che compongono le basi di dati (equivale a '`~mysql/`'); la directory usata per i file temporanei è '`/tmp/`'; infine, i messaggi mostrati dal demone sono quelli contenuti nella directory '`/usr/share/mysql/english/`', ovvero quelli espressi in lingua inglese.

Alcune direttive particolari non hanno la forma dell'assegnamento e contengono una sola stringa, nella forma seguente:

```
skip-nome
```

La presenza di queste direttive indica la disabilitazione di ciò che

viene descritto sinteticamente dalla parte finale del loro nome; per esempio, la direttiva '**skip-networking**' serve a disabilitare l'accesso attraverso la rete. Di solito, la configurazione predefinita di MySQL prevede proprio l'uso di questa direttiva per impedire l'accesso attraverso la rete, rendendo necessaria la modifica di questo file per consentirlo in modo esplicito.

Un altro tipo di direttiva, che comunque rientra nel genere normale di assegnamento, serve a dichiarare delle variabili, intese come opzioni di funzionamento, a cui si assegna anche un valore. Si tratta di direttive che hanno l'aspetto seguente:

```
set-variable = nome=valore
```

L'utilizzo di queste variabili può dipendere dal modo in cui si compilano i sorgenti di MySQL; pertanto può essere necessario stabilire su cosa si può intervenire, avviando il demone '**mysqld**' con l'opzione '**--help**', anche senza privilegi particolari:

```
$ /usr/bin/mysqld --help [Invio]
```

Oltre alla sintassi relativa alla riga di comando, si dovrebbe osservare la presenza dell'elenco seguente, che qui viene abbreviato:

```
Variables (--variable-name=value)
and boolean options {FALSE|TRUE}  Value (after reading options)
--------------------------------- ----------------------------
auto-rehash                       TRUE
character-sets-dir                (No default value)
default-character-set             (No default value)
compress                          FALSE
database                          (No default value)
…
net_buffer_length                 16384
select_limit                      1000
max_join_size                     1000000
```

76.1.1.1 Controllo dell'accesso dalla rete

Un problema comune che si può avvertire quando si cerca di mettere in funzione un servente MySQL, sta nel concedere gli accessi attraverso la rete.

Generalmente, la configurazione contenuta nel file '`/etc/mysql/my.cnf`' (o solo '`/etc/my.cnf`') prevede una direttiva che limita gli accessi al solo elaboratore locale. Ci sono due possibilità:

```
[mysqld]
…
skip-networking
…
bind-address = 127.0.0.1
…
```

Se si usa la direttiva '**`skip-networking`**', si intende concedere esclusivamente un accesso locale, tramite un socket di dominio Unix; se si usa la direttiva '**`bind-address = 127.0.0.1`**' (senza però usare anche '**`skip-networking`**'), si concede l'accesso

attraverso la rete, ma in pratica si concede esclusivamente un accesso locale, tramite l'indirizzo 127.0.0.1.

In pratica, per consentire un accesso remoto al DBMS, occorre eliminare (o commentare) entrambe queste direttive.

76.1.2 Avvio e arresto del servente

«

L'avvio e l'arresto del servente dovrebbe essere gestito da uno script della procedura di inizializzazione del sistema, che in generale potrebbe corrispondere a '`/etc/init.d/mysql`'. Se le cose stanno così, è sufficiente avviare il servizio con il comando seguente:

```
# /etc/init.d/mysql start [Invio]
```

Nello stesso modo, per arrestare il servizio basta il comando seguente:

```
# /etc/init.d/mysql stop [Invio]
```

Può essere interessante approfondire cosa succede realmente all'interno di questo script per comprendere come è organizzato MySQL nell'ambito del proprio sistema operativo.

In generale ci sono da considerare due aspetti: il demone '**mysqld**' non viene avviato direttamente, ma attraverso un altro programma, '**mysqld_safe**'; in secondo luogo, l'arresto del funzionamento del servente avviene attraverso '**mysqladmin**', che richiede l'indicazione di una parola d'ordine.

L'avvio del servente attraverso '**mysqld_safe**' non richiede la comprensione di altre questioni, a parte la necessità di accertarsi che non ci sia già un servente MySQL in funzione. Comunque, la presenza di questo programma ('**mysqld_safe**') fa sì che esista anche

una sezione apposita nella configurazione, denominata nello stesso modo:

```
[mysqld_safe]
err-log         = /var/log/mysql/mysql.err
socket          = /var/run/mysqld/mysqld.sock
```

Il problema dell'arresto del servizio è invece più complesso, in quanto si deve usare un altro programma, '**mysqladmin**', che può portare a termine l'operazione soltanto se si utilizzano i privilegi dell'amministratore del servizio, per ottenere i quali occorre fornire la parola d'ordine relativa:

```
# mysqladmin --password="ciaociao" shutdown [Invio]
```

L'amministratore di MySQL ha il nome '**root**' (in questo caso sarebbe precisamente '**root@localhost**'), che, come accennato nella premessa, coincide volutamente con il nome dell'amministrazione di un sistema Unix, pur non essendo la stessa cosa. Dal momento che ogni utente di MySQL può predisporre nella propria directory personale un file di configurazione personalizzato, corrispondente a '`~/.my.cnf`', in questo file si può anche inserire la propria parola d'ordine, con una direttiva della sezione '**[client]**', che in questo caso è riferita all'utente '**root**':

```
[client]
user      = root
password  = ciaociao
```

Così facendo, come viene chiarito in seguito, quando un utente del sistema operativo accede a un servente MySQL locale, se l'utenza di MySQL coincide con il nominativo usato nell'ambito del sistema operativo, può fare a meno di fornire la propria parola d'ordine perché già definita nella configurazione personale. Secondo questo

principio, l'utente '**root**' del sistema operativo potrebbe fare altrettanto per consentire a script che vengono avviati automaticamente di svolgere il loro compito.

Esistono comunque dei raggiri differenti, per evitare di costringere l'utente '**root**' del sistema operativo a inserire la parola d'ordine nel file di configurazione '`~/.my.cnf`'. In particolare, la distribuzione GNU/Linux Debian definisce un utente speciale, denominato '**debian-sys-maint**' ('**debian-sys-maint@localhost**'), con i privilegi necessari, a cui associa il file di configurazione '`/etc/mysql/debian.cnf`', avviando poi '**mysqladmin**' con l'opzione '**--defaults-extra-file**':

```
# mysqladmin --defaults-extra-file=/etc/mysql/debian.cnf ... [Invio]
```

Naturalmente, il file '`/etc/mysql/debian.cnf`' non deve essere leggibile agli utenti comuni del sistema operativo, dal momento che contiene la parola d'ordine per le funzioni amministrative gestite in modo automatico dal sistema stesso:

```
# Automatically generated for Debian scripts. DO NOT TOUCH!
[client]
host     = localhost
user     = debian-sys-maint
password = sddgetGRFhyjftuP
```

76.1.2.1 Struttura iniziale dei dati

«

Perché il servente MySQL possa essere avviato e funzionare, è necessario che sia stata predisposta una struttura iniziale di dati, che serve successivamente ad accogliere le basi di dati. Si tratta di file di una base di dati speciale, che si colloca assieme alle altre nella directory '`~mysql/`'.

Questa struttura iniziale si crea con il comando '**mysql_install_db**', che corrisponde a uno script. Si usa semplicemente così:

```
# mysql_install_db [Invio]
```

Può succedere che la propria distribuzione GNU, pur predisponendo un pacchetto per il servente MySQL, non provveda alla creazione di questa struttura iniziale dei dati; in tal caso non si riesce ad avviare il servente e occorre provvedere da soli a usare il comando '**mysql_install_db**'.

76.1.3 Utenze

MySQL distingue le proprie utenze in base a un nominativo e al nome dell'elaboratore da cui questi accedono. Il nominativo può anche essere assente e in tal caso si parla di utenze anonime. L'accesso può essere sottoposto al controllo di una parola d'ordine; inoltre, può anche essere consentito l'accesso a una base di dati senza essere utenti conosciuti.

Dal momento che l'utente è sempre riferito a un certo elaboratore, diventa necessario provvedere a un sistema di risoluzione dei nomi, anche se si tratta solo del file '`/etc/hosts`'; è comunque assolutamente indispensabile che sia stato definito il nome '**localhost**', riferito all'indirizzo IPv4 127.0.0.1, per consentire almeno l'accesso all'utente '**root@localhost**' che è l'amministratore quando accede dallo stesso elaboratore che ospita il servente MySQL.

Un primo controllo di accesso si ottiene con la configurazione del servente MySQL, dove è possibile escludere l'accesso attraverso la rete con la direttiva '**skip-networking**', cosa che è utile fare quando il sistema di utenze e dei privilegi relativi non è ancora stato definito.

Per poter iniziare a organizzare le basi di dati e le utenze di MySQL, occorre agire con i privilegi dell'utente '**root**'. Quando si installa MySQL per la prima volta, è molto probabile che questo utente risulti essere sprovvisto di parola d'ordine, consentendo in pratica a qualunque utente di dichiararsi '**root**', senza bisogno di altre forme di riconoscimento (ecco perché inizialmente è bene che la configurazione impedisca almeno l'accesso dall'esterno, attraverso la rete). Per associare una parola d'ordine all'utente '**root**' presso l'elaboratore che ospita il servente MySQL (quindi '**root@localhost**') quando questo ne è ancora sprovvisto, basta usare il comando seguente:

```
# mysqladmin -u root password 'ciaociao' [Invio]
```

Come si intende, la parola d'ordine che appare nell'esempio è proprio «ciaociao».

A partire dal momento in cui la parola d'ordine è stata definita, ogni accesso al servente compiuto da questo utente richiede la sua indicazione, a meno che questa parola d'ordine risulti inserita nel file di configurazione personale '`~/.my.cnf`':

```
[client]
user     = root
password = ciaociao
```

In questo caso, facendo riferimento alla sezione '**[client]**', la con-

figurazione riguarda qualunque situazione in cui si accede al servente; in pratica riguarda qualunque programma. Eventualmente, si possono definire contesti più precisi; per esempio come nel caso seguente, in cui si consente di non specificare la parola d'ordine solo quando si utilizza il programma '**mysqladmin**':

```
[mysqladmin]
user     = root
password = ciaociao
```

Il sistema di gestione delle utenze di MySQL è abbastanza complesso, dove per esempio, è possibile anche concedere dei privilegi di accesso a utenti qualunque provenienti da un certo nodo, o da un certo gruppo di nodi.

Inizialmente dovrebbe essere previsto l'utente '**root@localhost**', come già descritto più volte, ma potrebbe esistere anche un utente '**root@*hostname***', ovvero l'utente '**root**' che accede dal nodo corrispondente al nome che si ottiene con il comando '**hostname**'. Entrambe queste utenze hanno tutti i privilegi possibili ed entrambe potrebbero essere inizialmente senza parola d'ordine.
Assieme alle utenze '**root@**...' potrebbero essere stati definiti dei privilegi di accesso molto limitati per qualunque nominativo-utente che accede dal nodo locale, ovvero '**@localhost**' (ma si rappresenta necessariamente come '**''@localhost**'). Di solito ciò consente di fare degli esperimenti con la base di dati denominata '**test**', prima ancora di avere studiato i criteri di controllo degli accessi.

Per quanto riguarda l'indicazione dei nominativi-utente, esistono solo due possibilità: indicare un nome preciso, oppure non indicarlo affatto. Nel secondo caso si intende fare riferimento a qualunque utente nell'ambito del nodo o del gruppo di nodi a cui la definizione dei privilegi è associata.

Il nodo di accesso può essere indicato per nome, per numero, oppure si può fare riferimento a un gruppo di nodi con l'uso di caratteri jolly oppure associando a un indirizzo una maschera di rete. I caratteri jolly in questione sono il simbolo di percentuale ('`%`') e il trattino basso ('_'). Segue una tabellina che descrive alcuni esempi di associazione tra utenti e nodi di accesso.

utente*@*nodo	Descrizione
tizio@dinkel.brot.dg	L'utente '**`tizio`**' che accede dal nodo *dinkel.brot.dg*.
@dinkel.brot.dg	Qualunque utente che accede dal nodo *dinkel.brot.dg*.
tizio@%	L'utente '**`tizio`**' che accede da qualunque nodo.
@%	Qualunque utente che accede da qualunque nodo.
tizio@%.brot.dg	L'utente '**`tizio`**' che accede da un nodo che appartiene al dominio *brot.dg*.
tizio@brot.brot.%	L'utente '**`tizio`**' che accede da un nodo che appartiene a domini che iniziano per *dinkel.brot.*$*$.
tizio@111.112.113.114	L'utente '**`tizio`**' che accede dal nodo corrispondente all'indirizzo IPv4 111.112.113.114.

utente@*nodo*	Descrizione
tizio@111.112.113.%	L'utente '**tizio**' che accede da nodi con indirizzi IPv4 111.112.113.∗.
tizio@111.112.113.0/255.255.255.0	L'utente '**tizio**' che accede da nodi con indirizzi IPv4 111.112.113.∗ (come nell'esempio precedente).

A seconda del contesto, le coordinate di un utente si indicano come nella tabella, inserendo il carattere '**@**' per separare le due parti, oppure in campi distinti. In molti casi, nell'ambito delle istruzioni di MySQL è necessario proteggere il nome dell'utente e l'indicazione del nodo o dei nodi di accesso attraverso apici singoli. Per esempio, per poter indicare qualunque utente, come nel secondo esempio della tabella, è necessario delimitare la stringa nulla: '**''@dinkel.brot.dg**'. Così, quando si usa il carattere jolly '**%**' è indispensabile delimitare l'indicazione riferita al nodo: '**tizio@'%.brot.dg'**'. In generale non si sbaglia se si delimitano sempre queste due componenti, anche se non dovesse servire: '**''tizio'@'dinkel.brot.dg'**'.

76.1.4 Accesso comune al servente

Prima di poter definire delle politiche di accesso, è necessario prendere un po' di confidenza con il programma '**mysql**', che consente di interagire con il servente MySQL. Si è già accennato al fatto che inizialmente dovrebbe risultare permesso l'accesso da parte di

qualunque utente, inoltre dovrebbe essere disponibile la base di dati '**test**', a cui qualunque utente può accedere.

```
mysql [opzioni] [base_di_dati]
```

```
cat file_sql | mysql [opzioni] [base_di_dati]
```

I modelli sintattici mostrano la possibilità di indicare delle opzioni ed eventualmente il nome di una base di dati da aprire inizialmente. Nel secondo modello, si vede in particolare come si può alimentare il programma '**mysql**' con uno script SQL. La tabella successiva descrive alcune delle opzioni che possono essere utilizzate.

Opzione	Descrizione
`-h` ***nodo*** `--host=`***nodo***	Specifica il nome del nodo a cui ci si vuole connettere.
`-p` [***parola_d'ordine***] `--password` [`=`***parola_d'ordine***]	Specifica la parola d'ordine da fornire per ottenere l'accesso; se non viene indicata come argomento dell'opzione, viene richiesta in modo interattivo. In generale è meglio evitare di fornire la parola d'ordine già nella riga di comando, per non renderla evidente nell'elenco dei processi elaborativi.

Opzione	Descrizione
`-P` ***n_porta*** `--port=`***n_porta***	Specifica la porta da utilizzare per accedere al servizio; in modo predefinito è la numero 3306.
`-u` ***utente*** `--user=`***utente***	Specifica il nominativo-utente con il quale si vuole accedere.

Il programma '**`mysql`**' è uno di quelli che prende in considerazione la configurazione di MySQL, soprattutto per quanto riguarda il file '`~/.my.cnf`', dove gli utenti possono inserire il proprio nominativo-utente da utilizzare con MySQL e la parola d'ordine, per non dover ogni volta utilizzare le opzioni '**`-u`**' e '**`-p`**':

```
[client]
user     = tizio
password = supersegreta
```

Per esempio, se un certo utente del sistema operativo ha la necessità di identificarsi con il nominativo '**`tizio`**' e la sua parola d'ordine:

`$` **`mysql -u tizio -p`** … [*Invio*]

`Enter password:` ***digitazione_all'oscuro*** [*Invio*]

Oppure, disponendo del file '`~/.my.cnf`' mostrato sopra, può fare a meno di queste opzioni e dell'indicazione della parola d'ordine.

Viene mostrato un esempio riferito all'utente '**`tizio`**' che accede a un servente MySQL locale:

`$` **`mysql -u tizio`** [*Invio*]

Molto probabilmente non è necessario inserire alcuna parola d'ordine, dal momento che inizialmente dovrebbe essere stata inserita automaticamente l'utenza anonima '**@localhost**' senza parola d'ordine:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 14 to server version: 4.0.13-log

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

Da questo momento appare un invito speciale, rappresentato dalla stringa '**mysql>**'. La prima cosa che può essere conveniente fare è verificare la disponibilità di basi di dati a cui si può accedere:

```
mysql> SHOW DATABASES; [Invio]
```

Molto probabilmente si ottiene precisamente quanto segue, ovvero il riferimento all'unica base di dati, '**test**', a cui è consentito accedere con questo utente:

```
+----------+
| Database |
+----------+
| test     |
+----------+
1 row in set (0.03 sec)
```

Si può selezionare la base di dati con il comando seguente:

```
mysql> USE test; [Invio]

Database changed
```

Quindi si può tentare di consultare l'elenco delle relazioni disponibili, con il comando seguente, ma inizialmente la basi di dati '**test**' non ne contiene alcuna:

```
mysql> SHOW TABLES; [Invio]

  Empty set (0.00 sec)
```

A titolo di esempio si vuole creare la relazione '**Indirizzi**' con il codice SQL seguente:

```
CREATE TABLE Indirizzi (
        Codice          INTEGER,
        Cognome         CHAR(40),
        Nome            CHAR(40),
        Indirizzo       VARCHAR(60),
        Telefono        VARCHAR(40)
    );
```

Con il programma '**mysql**' si può fare così:

```
mysql> CREATE TABLE Indirizzi ( [Invio]

    -> Codice INTEGER, [Invio]

    -> Cognome CHAR(40), [Invio]

    -> Nome CHAR(40), [Invio]

    -> Indirizzo VARCHAR(60), [Invio]

    -> Telefono VARCHAR(40) [Invio]

    -> ); [Invio]

  Query OK, 0 rows affected (0.06 sec)
```

Come si può osservare, finché l'istruzione SQL non risulta completa, appare un invito differente: '**->**'.

Per completare l'esempio si può inserire qualche dato e poi si può visualizzare il contenuto della relazione:

```
mysql> INSERT INTO Indirizzi VALUES (01, 'Pallino', 'Pinco', [Invio]
    -> 'Via Biglie 1', '0222,222222'); [Invio]

  Query OK, 1 row affected (0.06 sec)

mysql> SELECT * FROM Indirizzi; [Invio]

  +--------+---------+-------+-------------+-------------+
  | Codice | Cognome | Nome  | Indirizzo   | Telefono    |
  +--------+---------+-------+-------------+-------------+
  |      1 | Pallino | Pinco | Via Biglie 1 | 0222,222222 |
  +--------+---------+-------+-------------+-------------+
  1 row in set (0.00 sec)
```

Per concludere il funzionamento di '**mysql**' basta il comando interno '**quit**', che si può esprimere anche come '**\q**':

```
mysql> \q[Invio]

  Bye
```

76.1.5 Base di dati amministrativa

«

MySQL gestisce le proprie informazioni amministrative all'interno della base di dati '**mysql**', a cui si dovrebbe poter accedere soltanto in qualità di utente '**root**' (possibilmente '**root@localhost**'), o comunque solo attraverso utenze speciali.

```
# mysql -u root -p[Invio]

Enter password: digitazione all'oscuro [Invio]
```

In questo modo si vuole accedere al servente MySQL locale, in qualità di utente '**root**' ('**root@localhost**'), fornendo anche la

parola d'ordine.

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 15 to server version: 4.0.13-log

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

Si controlla la disponibilità di basi di dati esistenti:

```
mysql> SHOW DATABASES; [Invio]

+----------+
| Database |
+----------+
| mysql    |
| test     |
+----------+
2 rows in set (0.01 sec)
```

Si accede alla base di dati '**mysql**':

```
mysql> USE mysql; [Invio]

Database changed
```

Si elencano le relazioni disponibili:

```
mysql> SHOW TABLES; [Invio]
```

```
+-----------------+
| Tables_in_mysql |
+-----------------+
| columns_priv    |
| db              |
| func            |
| host            |
| tables_priv     |
| user            |
+-----------------+
6 rows in set (0.00 sec)
```

Il significato dettagliato di queste relazioni può essere studiato nella documentazione originale che accompagna MySQL. Ciò che è importante comprendere è che non si può consentire l'accesso a queste relazioni a utenti che non hanno un ruolo amministrativo. Tuttavia, durante la fase di studio di MySQL da parte di chi deve poi amministrarne il servizio, è utile imparare a consultare queste relazioni. Per esempio, è utile essere al corrente delle utenze che sono effettivamente previste:

```
mysql> SELECT User, Host, Password FROM user; [Invio]

+------------------+-----------+------------------+
| user             | host      | password         |
+------------------+-----------+------------------+
| root             | localhost | 576gtd435e967361 |
| root             | dinkel    |                  |
|                  | localhost |                  |
|                  | dinkel    |                  |
| debian-sys-maint | localhost | 69b3c178kbvcd325 |
+------------------+-----------+------------------+
5 rows in set (0.00 sec)
```

Quello che si vede qui è quanto si potrebbe ottenere dopo aver installato MySQL in una distribuzione GNU/Linux Debian, attraverso pacchetti appositi, in un elaboratore dove il comando '**hostname**' restituisce il nome '**dinkel**'. Si può notare la presenza dell'utente speciale '**debian-sys-maint**', a cui si è già accennato, ma si deve portare attenzione alle utenze «normali». Si può osservare che ci sono due utenti '**root**'; precisamente si tratta di '**root@localhost**' e di '**root@dinkel**'. La seconda di queste due utenze rappresenta l'utente '**root**' che accede localmente, ma attraverso la rete, da un'interfaccia che è associata al nome dell'elaboratore (in questo caso *dinkel.brot.dg*). Può anche darsi che un'utenza annotata in questo modo non risulti funzionante, ma rimane il problema, dato il fatto che si tratta di un'utenza importante e senza parola d'ordine per accedere (per quanto riguarda l'utenza '**root@localhost**' è già stato mostrato come definire la parola d'ordine ed è per questo che appare qualcosa nella colonna '**password**'). Eventualmente, si possono fare considerazioni simili per l'utenza anonima '**@dinkel**'.

Probabilmente è superfluo precisare che nella colonna '**password**', se esiste una parola d'ordine associata all'utenza, appare una stringa che rappresenta la parola d'ordine cifrata.

```
mysql> \q[Invio]

  Bye
```

Durante la fase della prima installazione di MySQL uno script si occupa di creare le directory che devono ospitare i file che compongono le basi di dati, inserendo la base di dati amministrativa ('**mysql**'). In pratica, è questo script che definisce anche le utenze iniziali, senza parola d'ordine. Questo script è '**mysql_install_db**'.

76.2 Gestione del DBMS

Una volta superata la fase della configurazione del servizio di MySQL; una volta compreso come utilizzare gli strumenti essenziali per interagire con questo, si può passare alla gestione del DBMS, che implica l'amministrazione delle utenze e delle basi di dati.

76.2.1 Controllo delle utenze e degli accessi

Il modo normale per creare un'utenza con MySQL è quello di concedere dei privilegi di accesso, attraverso l'istruzione '**`GRANT`**'; se poi quell'utenza esiste già, i privilegi in questione vengono solo aggiunti a quelli già esistenti. Per eliminare un'utenza, invece, non è sufficiente privarla di tutti i privilegi con l'istruzione '**`REVOKE`**', perché occorre anche eliminare la tupla corrispondente all'utenza nella relazione '**`user`**' della base di dati '**`mysql`**'.

Il comportamento di MySQL è abbastanza diverso rispetto allo standard ANSI, per quanto riguarda l'identificazione delle utenze e l'attribuzione o l'eliminazione dei privilegi relativi. In particolare, è importante il fatto che MySQL distingua le utenze in base al nodo di provenienza; inoltre è possibile concedere privilegi complessivi: per una base di dati completa, o anche per tutte le basi di dati esistenti. A questo si aggiunga che è possibile fare riferimento a una relazione di una base di dati indicando esplicitamente la base di dati relativa, con la forma: '***base_di_dati*** **.** ***relazione***'.

Gli schemi seguenti mostrano la sintassi semplificata per l'uso delle istruzioni '**`GRANT`**' e '**`REVOKE`**' con MySQL:

```
GRANT privilegio [, privilegio] ...
  ON {relazione | * | base_di_dati.* | *.*}
  TO utenza [IDENTIFIED BY [PASSWORD] 'parola_d'ordine']
    [, utenza [IDENTIFIED BY [PASSWORD] 'parola_d'ordine']] ...
  [WITH GRANT OPTION]
```

```
REVOKE privilegio [, privilegio] ...
    ON {relazione | * | base_di_dati.* | *.*}
    FROM utenza [, utenza] ...
```

I privilegi che possono essere concessi o revocati sono espressi attraverso una parola chiave. Alcuni di questi privilegi sono descritti nell'elenco seguente:

Privilegio	Descrizione
`ALL [PRIVILEGES]`	concede tutti i privilegi disponibili;
`ALTER`	consente la modifica delle relazioni;
`CREATE`	consente la creazione delle relazioni;
`DELETE`	consente la cancellazione dei dati contenuti nelle relazioni;
`DROP`	consente l'eliminazione delle relazioni;
`INDEX`	consente di creare ed eliminare degli indici;
`INSERT`	consente l'inserimento di dati nelle relazioni;

Privilegio	Descrizione
`SELECT`	consente la lettura dei dati nelle relazioni;
`UPDATE`	consente la modifica dei dati all'interno delle relazioni;
`USAGE`	serve solo a creare un utente senza privilegi;
`GRANT OPTION`	consente di dare ad altri i propri privilegi.

Si osservi che MySQL prevede anche privilegi «particolari», che dipendono dalle proprie specificità rispetto allo standard ANSI. In generale, si può considerare che l'utente '**`root`**' deve possedere tutti i privilegi, mentre possono esistere delle utenze amministrative fittizie (come nel caso di '**`debian-sys-maint`**') con privilegi particolari legati alla possibilità di arrestare il funzionamento del servente MySQL.

Lo standard ANSI prevede la concessione di privilegi su relazioni singole, mentre MySQL permette di fare riferimento a basi di dati complete. Pertanto, nel modello sintattico mostrato sono apparse delle notazioni speciali:

Modello	Descrizione
relazione	rappresenta una relazione singola, che può contenere anche l'indicazione della basi di dati che la contiene, nella forma '***base_di_dati . relazione***';
`*`	l'asterisco rappresenta tutte le relazioni della basi di dati attiva, ma se non è stata selezionata una base di dati in precedenza, si fa riferimento a tutte le basi di dati;

Modello	Descrizione
base_di_dati . *	l'indicazione di una base di dati con un asterisco al posto del nome della relazione, serve a fare riferimento a tutta la base di dati nel complesso;
* . *	l'indicazione di due asterischi separati da un punto serve a fare riferimento esplicito a tutte le relazioni di tutte le basi di dati.

Quando si utilizza l'istruzione '**GRANT**' è consentito l'uso dei caratteri jolly '_' e '%', con il significato comune nell'ambito del linguaggio SQL. Ciò consente di fare riferimento a gruppi di relazioni e a gruppi di basi di dati, secondo la corrispondenza del modello. Tuttavia, nel caso questi simboli debbano essere usati in modo letterale, è necessario proteggerli con la barra obliqua inversa: '_' e '\%' (in generale, è improbabile che si dia un nome a una base di dati o a una relazione che contenga il simbolo di percentuale, ma è più probabile che si pensi invece di usare il trattino basso).

Come già accennato altre volte, l'utenza tiene conto anche della provenienza dell'accesso, secondo la forma seguente:

nominativo_utente [@*nodo_di_provenienza*]

Pertanto, l'indicazione del nominativo è obbligatoria, mentre si può omettere la specificazione del nodo di provenienza, se si vuole fare riferimento a qualunque origine.

Sono già stati descritti i modi in cui si può rappresentare un utente secondo il modello '*utente*@*nodo*'; in particolare è già stato descrit-

to l'utilizzo dei caratteri jolly (anche se sono stati mostrati soltanto esempi con il simbolo '`%`').

Si ricorda comunque che si possono usare i caratteri jolly soltanto nella parte che descrive i nodi di provenienza

Infine, è già stata indicata la necessità di usare gli apici singoli per delimitare separatamente il nominativo e la specificazione del nodo di provenienza quando questi nomi contengono caratteri «particolari», compresi i caratteri jolly '`%`' e '_'.

Si possono presentare delle ambiguità nell'individuazione dei privilegi delle utenze. Per esempio, può esistere l'utente anonimo '**`''@dinkel.brot.gd`**', l'utente '**`tizio@'%'`**' e l'utente '**`tizio@dinkel.brot.gd`**': quando un utente accede valgono per lui i privilegi più specifici che gli si possono individuare, altrimenti, in presenza di utenze anonime, i privilegi di queste prevarrebbero.

Come si vede dal modello sintattico dell'istruzione '**`GRANT`**', è possibile specificare una parola d'ordine. Quando si concedono dei privilegi a un utente che non è ancora stato definito e non si stabilisce la parola d'ordine, l'utenza in questione rimane priva di parola d'ordine; pertanto, per accedere non deve essere fornita. Se invece l'utenza esiste già, i privilegi vengono aggiunti e la presenza di una parola d'ordine eventuale serve solo per cambiare quella preesistente. Tuttavia, è possibile cambiare una parola d'ordine anche con l'i-

struzione '**SET PASSWORD**', se si tratta della propria o se si hanno i privilegi dell'amministratore:

```
SET PASSWORD FOR utenza = PASSWORD('parola_d'ordine')
```

In alternativa, è anche possibile usare il comando '**mysqladmin**' (dal sistema operativo), ma solo per cambiare la propria parola d'ordine:

```
$ mysqladmin password 'supersegreta' [Invio]
```

Se si utilizza l'opzione '**GRANT OPTION**', o '**WITH GRANT OPTION**', si permette all'utenza a cui si fa riferimento di concedere ad altri gli stessi privilegi di cui si dispone.

Come accennato, l'eliminazione di un'utenza richiede prima l'eliminazione di tutti i privilegi e quindi la cancellazione dalla relazione '**user**' della base di dati '**mysql**'.

Nel seguito vengono descritti alcuni esempi attraverso una sequenza lineare di operazioni, a cominciare dall'avvio del programma '**mysql**' per interagire con il servente.

```
$ mysql -u root -p [Invio]

Enter password: parola_d'ordine [Invio]

  Welcome to the MySQL monitor.  Commands end with ; or \g.
  Your MySQL connection id is 6 to server version: 4.0.13-log

  Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

Si crea una base di dati nuova:

```
mysql> CREATE DATABASE Magazzino; [Invio]
```

```
Query OK, 1 row affected (0.26 sec)
```

Si crea un utente amministratore per la base di dati appena creata, che può accedere da dove vuole:

```
mysql> GRANT ALL ON Magazzino.* TO amministratore@'%' ↩
↪      IDENTIFIED BY 'segreta' WITH GRANT OPTION; [Invio]

Query OK, 0 rows affected (0.35 sec)
```

Si osservi che se esiste un'utenza anonima riferita al nodo locale (utenza che verrebbe indicata come '**''@localhost**'), se l'utente appena creato volesse accedere localmente, non verrebbe identificato come amministratore, ma solo come utente anonimo. Per risolvere il problema si potrebbe aggiungere l'utente '**amministratore@localhost**', ma in questo caso si preferisce eliminare l'utenza anonima che interferisce e non si vuole mantenere:

```
mysql> REVOKE ALL ON *.* FROM ''@localhost; [Invio]

Query OK, 0 rows affected (0.15 sec)

mysql> USE mysql; [Invio]

Database changed

mysql> DELETE FROM user WHERE user = '' ↩
↪      AND host = 'localhost'; [Invio]

Query OK, 0 rows affected (0.15 sec)
```

Si crea un'altra utenza in grado di consultare e di modificare i dati delle relazioni che deve contenere la base di dati '**Magazzino**'; anche in questo caso si consente l'accesso da qualunque nodo:

```
mysql> GRANT DELETE, INSERT, SELECT, UPDATE ON Magazzino.* ↩
↪      TO tizio@'%' IDENTIFIED BY 'ottimo'; [Invio]

  Query OK, 0 rows affected (0.05 sec)
```

Supponendo di avere installato MySQL nell'elaboratore *dinkel.brot.dg*, dovrebbero essere presenti anche le utenze '**''@dinkel**' e '**root@dinkel**', che si preferisce eliminare:

```
mysql> REVOKE ALL ON *.* FROM ''@dinkel; [Invio]

  Query OK, 0 rows affected (0.15 sec)

mysql> REVOKE ALL ON *.* FROM root@dinkel; [Invio]

  Query OK, 0 rows affected (0.15 sec)

mysql> USE mysql; [Invio]

  Database changed

mysql> DELETE FROM user WHERE user = '' ↩
↪       AND host = 'dinkel'; [Invio]

  Query OK, 0 rows affected (0.15 sec)

mysql> DELETE FROM user WHERE user = 'root' ↩
↪       AND host = 'dinkel'; [Invio]

  Query OK, 0 rows affected (0.15 sec)

mysql> \q [Invio]

  Bye
```

La documentazione di MySQL descrive anche come compiere tutte queste operazioni amministrative intervenendo esclusivamente nella base di dati '**mysql**'. Nel caso si preferisca intervenire in

quel modo, è necessario concludere le operazioni di modifica o aggiornamento delle relazioni amministrative con l'istruzione '**`FLUSH PRIVILEGES`**'.

76.2.2 Amministrazioni varie attraverso il sistema operativo

«

In questo capitolo si è fatto riferimento più volte al programma '**`mysqladmin`**' a proposito della possibilità di assegnare e modificare la parola d'ordine di un utente. Questo programma ha anche altre funzionalità; in particolare consente di creare ed eliminare una base di dati e di arrestare il funzionamento del servente MySQL, senza bisogno di utilizzare istruzioni SQL. Vengono sintetizzate le sintassi da utilizzare per le varie occasioni:

Comando	Descrizione
`mysqladmin [-u root] [-p] ↩` `↪[-h` ***nodo*** `] ↩` `↪create database` ***base_di_dati***	crea la base di dati indicata;
`mysqladmin [-u root] [-p] ↩` `↪[-h` ***nodo*** `] ↩` `↪drop database` ***base_di_dati***	elimina la base di dati indicata con tutto il suo contenuto;
`mysqladmin [-u` ***utente*** `] [-p] ↩` `↪[-h` ***nodo*** `] ↩` `↪password` ***parola_d'ordine***	assegna o cambia la parola d'ordine per accedere;
`mysqladmin [-u root] [-p] ↩` `↪[-h` ***nodo*** `] shutdown`	arresta il funzionamento del servente.

76.2.3 Ripristino della parola d'ordine dell'amministratore

Nel caso fosse necessario modificare la parola d'ordine dell'amministratore del DBMS ('**root**'), senza poter conoscere quella precedente, esiste un procedimento che è bene annotare. Per prima cosa si deve arrestare il servente, nel caso questo fosse in funzione; dovrebbe essere possibile farlo così:

```
# /etc/init.d/mysql stop[Invio]
```

Successivamente si deve avviare '**mysqld_safe**', con l'opzione '**--skip-grant-tables**', in modo da ignorare completamente il controllo dei privilegi concessi (o negati) agli utenti del DBMS:

```
# mysqld_safe --skip-grant-tables &[Invio]
```

Se il servizio si avvia regolarmente, è possibile accedere alle basi di dati senza vincoli; pertanto è possibile cambiare parola d'ordine intervenendo direttamente nella base di dati amministrativa '**mysql**':

```
# mysql -u root[Invio]

  Welcome to the MySQL monitor.  Commands end with ; or \g.
  Your MySQL connection id is 2 to server version: ↩
↪4.0.24_Debian-10-log

  Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE mysql;[Invio]

  Reading table information for completion of table and column
  names. You can turn off this feature to get a quicker startup
  with -A

  Database changed
```

```
mysql> UPDATE user SET password=password("supersegreta") ↩
↪      WHERE user="root"; [Invio]

  Query OK, 2 rows affected (0.00 sec)
  Rows matched: 2  Changed: 2  Warnings: 0

mysql> \q[Invio]

  Bye
```

Un modo alternativo, ma drastico, di ripristinare l'utenza dell'amministratore senza parola d'ordine, consiste nell'eliminazione «manuale» della base di dati '**mysql**', da ricostruire con l'aiuto di '**mysql_install_db**'. Naturalmente, in questo modo si perdono le informazioni su tutti gli altri utenti del DBMS.

76.2.4 Archiviazione e recupero delle basi di dati

«

MySQL gestisce le sue basi di dati come sottodirectory di '`~mysql/`', che di solito corrisponde a '`/var/lib/mysql/`'. Per esempio, la base di dati '**prova**' corrisponde alla struttura che si articola a partire da '`~mysql/prova/`'.

Per archiviare una base di dati è sufficiente fare una copia della struttura che la riguarda; per ripristinare una base di dati è sufficiente rimpiazzare la struttura esistente con la sua copia fatta in precedenza; per duplicare una base di dati è sufficiente fare la copia della struttura di origine, utilizzando un nome differente. Per esempio, disponendo della base di dati '**prova**', è possibile creare un'altra identica, ma con nome differente, così:

```
# cp -dpRv ~mysql/prova ~mysql/prova2[Invio]
```

In base all'esempio si ottiene una seconda base di dati con il nome '**prova2**', con lo stesso contenuto di '**prova**'.

Tuttavia, con il procedere dello sviluppo di MySQL bisogna considerare la possibilità che il formato dei file usati per descrivere le relazioni delle basi di dati cambi nel tempo. Pertanto, per archiviare una base di dati in modo abbastanza duraturo, è necessario creare un file di testo contenente comandi SQL. Si ottiene questo con il programma '**mysqldump**':

```
# mysqldump -u root -p prova > prova.sql [Invio]
```

L'esempio mostra in che modo archiviare la base di dati '**prova**' nel file '`prova.sql`'. Per fare questo, logicamente, ci si identifica in qualità di amministratore (con il nominativo '**root**') e nell'esempio si usa l'opzione '**-p**', per richiedere l'inserimento della parola d'ordine, supponendo che ciò sia necessario.

Per ripristinare un file ottenuto in questo modo, occorre prima creare o ricreare la base di dati; nell'esempio seguente si cancella prima la base di dati '**prova**', quindi la si ricrea vuota:

```
# mysql -u root -p [Invio]

Enter password: digitazione_all'oscuro [Invio]

  Welcome to the MySQL monitor.  Commands end with ; or \g.
  Your MySQL connection id is 6 to server version: 4.0.13-log

  Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> DROP DATABASE prova; [Invio]

  Query OK, 0 rows affected (0.00 sec)
```

```
mysql> CREATE DATABASE prova; [Invio]

  Query OK, 1 row affected (0.00 sec)

mysql> \q[Invio]

  Bye
```

Per recuperare la base di dati archiviata in forma di comandi SQL, dopo un controllo visivo del suo contenuto, si può procedere così:

```
# mysql -u root -p prova < prova.sql [Invio]

Enter password: digitazione_all'oscuro [Invio]
```

In questo modo, si fa leggere a '**mysql**' il contenuto del file '`prova.sql`', che dovrebbe contenere le istruzioni per la rigenerazione delle relazioni della base di dati originaria.

Si osservi che nella riga di comando di '**mysql**' appare l'indicazione della base di dati di destinazione; pertanto, si potrebbe benissimo ricreare una base di dati con un nome differente da quello che aveva nel momento dell'archiviazione in forma di comandi SQL. Di conseguenza, questa tecnica di archiviazione e recupero è anche quella più appropriata per duplicare una base di dati.

76.3 Riferimenti

«

- *MySQL reference manual*, http://dev.mysql.com/doc/mysql/en/index.html

[1] **MySQL** GNU GPL

77.1 Utilizzo generale 2104
77.1.1 Utilizzo di «sqlite3» 2104
77.1.2 Copie di sicurezza 2106
77.1.3 Accesso simultaneo a più basi di dati 2108
77.2 Esempi comuni 2109
77.2.1 Creazione di una relazione 2109
77.2.2 Modifica della relazione 2110
77.2.3 Inserimento dati in una relazione 2110
77.2.4 Eliminazione di una relazione 2111
77.2.5 Interrogazioni semplici 2111
77.2.6 Interrogazioni simultanee di più relazioni 2114
77.2.7 Alias 2116
77.2.8 Viste 2117
77.2.9 Aggiornamento delle tuple 2117
77.2.10 Cancellazione delle tuple 2118
77.2.11 Inserimento in una relazione esistente 2118
77.3 Riferimenti 2119

77.1 Utilizzo generale

«

SQLite[1] è una libreria in grado di fornire le funzionalità di un DBMS relazionale, basato sul linguaggio SQL, utilizzando come basi di dati dei file singoli. In pratica, SQLite gestisce una basi di dati intera in un file singolo, senza alcuna amministrazione esterna; pertanto, le utenze e i privilegi sono definiti dal sistema operativo, in quanto gli accessi sono regolati dai permessi del file che contiene la basi di dati.

In qualità di libreria, SQL consente ai programmi che la utilizzano di avere accesso a una basi di dati anche senza bisogno di collegarsi a un servizio DBMS tradizionale, facilitando così la realizzazione di piccoli progetti.

Di norma, la libreria SQL è accompagnata da un programma per l'esecuzione interattiva di istruzioni SQLite, che consente di accedere alle sue basi di dati in modo generalizzato.

77.1.1 Utilizzo di «sqlite3»

«

Per creare o accedere a una basi di dati di SQL, si può usare il programma '**`sqlite3`**', o '**`sqlite`**' nelle versioni più vecchie, che accompagna normalmente la libreria vera e propria:

```
sqlite3 [opzioni] [file_base_di_dati]
```

Generalmente, alla fine della riga di comando si indica il nome del file base di dati a cui si vuole accedere; se poi il file non dovesse esistere, il fatto di nominarlo implicherebbe comunque la sua creazione (come base di dati vuota). Se il nome di questo file non viene fornito attraverso la riga di comando, occorre usare un'istruzione apposita,

durante il funzionamento di '**sqlite**'. L'esempio seguente mostra l'avvio e la conclusione del programma, allo scopo di accedere alla base di dati contenuta nel file '`mia_prova.db`':

```
$ sqlite3 mia_prova.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite3> .quit[Invio]
```

Come si può intendere, si tratta di un programma che si comporta sostanzialmente come altri programmi frontali simili, usati per accedere in modo diretto ad altri DBMS basati su SQL.

Tabella 77.2. Alcune opzioni per l'avvio di '**sqlite3**'.

Opzione	Descrizione
`-init` ***file***	Consente di indicare un file contenente comandi per '**sqlite3**'; può trattarsi di comandi interni al programma, oppure di istruzioni SQL.
`-column`	Richiede di mostrare i risultati in modo incolonnato.
`-html`	Richiede di generare i risultati delle interrogazioni in forma di tabella HTML.

Oltre all'uso dell'opzione '**-init**', il programma '**sqlite3**' può ricevere un file contenente comandi e istruzioni SQL semplicemente dallo standard input.

Tabella 77.3. Alcuni comandi interni di '**sqlite3**'.

Comando	Descrizione
`.help`	Mostra una guida interna all'uso del programma.
`.exit` `.quit`	Termina il funzionamento del programma.
`.dump` [*relazione*] ...	Consente di «scaricare» il contenuto della base di dati, oppure delle sole relazioni indicate, generando le istruzioni necessarie a ricostruire le informazioni relative.
`.read` *file*	Legge ed esegue i comandi contenuti nel file indicato.
`.mode column`	Fa in modo di visualizzare il risultato delle interrogazioni in una forma incolonnata.
`.mode list`	Fa in modo di visualizzare il risultato delle interrogazioni nel modo usuale di '**sqlite3**'.
`.header on` `.header off`	attiva o disattiva l'intestazione delle colonne.

77.1.2 Copie di sicurezza

«

È evidente che la copia di una base di dati realizzata con SQLite è un'operazione molto semplice, essendo tutto contenuto in un file. Tuttavia, per garantire la trasferibilità in un'architettura differente, è necessario procedere alla generazione di un file di testo contenente le istruzioni SQL con cui poi ricostruire la base di dati. Si ottiene lo scarico in questa forma usando il comando '**.dump**':

```
echo ".dump" | sqlite3 file_base_di_dati > file_sql
```

Per esempio, per generare il file ‘`backup.sql`’ dalla base di dati contenuta nel file ‘`mio.db`’, si potrebbe usare il comando seguente:

```
$ echo ".dump" | sqlite3 mio.db > backup.sql [Invio]
```

Quello che si ottiene nel file ‘`backup.sql`’ potrebbe avere l’aspetto seguente:

```
BEGIN TRANSACTION;
CREATE TABLE Indirizzi (codice integer, cognome char(40), ↩
↪nome char(40), indirizzo varchar(60), telefono varchar(40));
INSERT INTO "Indirizzi" VALUES(1, 'Pallino', 'Pinco', ↩
↪'Via Biglie, 1', '0222,222222');
…
CREATE TABLE presenze (codice integer, giorno date, ↩
↪ingresso time, uscita time);
INSERT INTO "presenze" VALUES(1, '2005-09-17', '11:20:00', '13:30:00');
…
CREATE VIEW ingressi as select indirizzi.cognome, indirizzi.nome, ↩
↪presenze.giorno, presenze.ingresso ↩
↪from indirizzi, presenze ↩
↪where indirizzi.codice = presenze.codice;
COMMIT;
```

Per ricostruire una base di dati da un file del genere, basta fornire il file a ‘**`sqlite3`**’ dallo standard input:

```
sqlite3 file_base_di_dati < file_sql
```

Per esempio, per creare una base di dati nuova nel file ‘`nuova.db`’, a partire dallo stesso file appena creato, si potrebbe usare il comando seguente:

```
$ sqlite3 nuova.db < backup.sql [Invio]
```

77.1.3 Accesso simultaneo a più basi di dati

«

SQLite consente di accedere a più di una basi di dati alla volta. Per fare questo è disponibile un'istruzione SQL che non è standard:

```
ATTACH [DATABASE] file_base_di_dati AS nome_interno
```

In pratica, si indica il file della basi di dati a cui collegarsi, tenendo conto che può essere necessario indicarlo tra virgolette, ma le si deve associare un nome. Nell'esempio seguente, durante il funzionamento di '**sqlite3**' viene collegata la basi di dati contenuta nel file '`seconda.db`', associandola al nome '**seconda**':

```
sqlite> ATTACH 'seconda.db' AS seconda; [Invio]
```

La base di dati già aperta all'atto dell'avvio di '**sqlite3**' acquista automaticamente il nome interno '**main**'. La distinzione delle basi di dati con questi nomi interni, consente di accedere a relazioni che altrimenti avrebbero lo stesso nome: per indicare la relazione '**indirizzi**' della base di dati '**seconda**', si indica '**seconda.indirizzi**'; per indicare la relazione con lo stesso nome, contenuta nella base di dati aperta all'avvio del programma, si indica '**main.indirizzi**'. Comunque, si osservi che non è necessario indicare dei nomi completi della base di dati se non ci sono ambiguità tra le relazioni.

Eventualmente, per chiudere il collegamento con una base di dati c'è un'altra istruzione:

```
DETACH [DATABASE] nome_interno
```

77.2 Esempi comuni

Nelle sezioni seguenti vengono mostrati alcuni esempi comuni di utilizzo del linguaggio SQL, limitato alle possibilità di SQLite. La sintassi non viene descritta. Negli esempi si fa riferimento frequentemente a una relazione di indirizzi, il cui contenuto è visibile nella figura successiva.

Figura 77.5. La relazione '**`Indirizzi (Codice, Cognome, Nome, Indirizzo, Telefono)`**' usata negli esempi del capitolo.

Indirizzi				
Codice	**Cognome**	**Nome**	**Indirizzo**	**Telefono**
1	Pallino	Pinco	Via Biglie 1	0222,222222
2	Tizi	Tizio	Via Tazi 5	0555,555555
3	Cai	Caio	Via Caini 1	0888,888888
4	Semproni	Sempronio	Via Sempi 7	0999,999999

77.2.1 Creazione di una relazione

La relazione di esempio, denominata '**`Indirizzi`**', potrebbe essere creata nel modo seguente:

```
sqlite> CREATE TABLE Indirizzi ( [Invio]
   ...>         Codice          integer, [Invio]
   ...>         Cognome         char(40), [Invio]
   ...>         Nome            char(40), [Invio]
```

```
...>          Indirizzo       varchar(60), [Invio]

...>          Telefono        varchar(40) [Invio]

...>      ); [Invio]
```

77.2.2 Modifica della relazione

SQLite consente soltanto l'aggiunta di attributi alle relazioni, mentre la loro eliminazione o il cambiamento del dominio (il tipo), non è ammissibile. Segue un esempio con cui si aggiunge un attributo:

```
sqlite> ALTER TABLE Indirizzi ADD COLUMN Comune char(30);
[Invio]
```

77.2.3 Inserimento dati in una relazione

L'esempio seguente mostra l'inserimento dell'indirizzo dell'impiegato «Pinco Pallino»:

```
sqlite> INSERT INTO Indirizzi VALUES ([Invio]

...>          01, [Invio]

...>          'Pallino', [Invio]

...>          'Pinco', [Invio]

...>          'Via Biglie 1', [Invio]

...>          '0222,222222'); [Invio]
```

In questo caso, si presuppone che i valori inseriti seguano la sequenza degli attributi, così come è stata creata la relazione in origine; tuttavia, si osservi che se la relazione ha degli attributi in più, si ottiene una segnalazione di errore e l'inserimento viene rifiutato.

Per indicare un comando più leggibile, evitando anche problemi quando dovessero esserci attributi ulteriori, che però non si vogliono prendere in considerazione, occorre aggiungere l'indicazione della sequenza degli attributi da compilare, come nell'esempio seguente:

```
sqlite> INSERT INTO Indirizzi (Codice, Cognome, Nome, [Invio]
   ...>                        Indirizzo, Telefono) [Invio]
   ...>        VALUES (01, 'Pallino', 'Pinco', [Invio]
   ...>                'Via Biglie 1', '0222,222222'); [Invio]
```

In questo modo, gli attributi che non vengono indicati, ricevono il valore predefinito (se esiste), oppure '**NULL**' in mancanza d'altro.

77.2.4 Eliminazione di una relazione

Una relazione può essere eliminata completamente attraverso l'istruzione '**DROP**'. L'esempio seguente elimina la relazione degli indirizzi degli esempi già mostrati:

```
sqlite> DROP TABLE Indirizzi; [Invio]
```

77.2.5 Interrogazioni semplici

L'esempio seguente emette tutto il contenuto della relazione degli indirizzi già vista negli esempi precedenti:

```
sqlite> SELECT * FROM Indirizzi; [Invio]
```

Il risultato può apparire in formati differenti; di solito si ottiene così:

```
1|Pallino|Pinco|Via Biglie 1|0222,222222
2|Tizi|Tizio|Via Tazi 5|0555,555555
3|Cai|Caio|Via Caini 1|0888,888888
4|Semproni|Sempronio|Via Sempi 7|0999,999999
```

Per ottenere un elenco incolonnato, occorre usare il comando '**.mode column**', ma in tal caso l'ampiezza delle colonne è fissa e le informazioni potrebbero apparire troncate:

```
sqlite> .mode column[Invio]

sqlite> SELECT * FROM Indirizzi;[Invio]

1          Pallino     Pinco       Via Biglie 1   0222,222222
2          Tizi        Tizio       Via Tazi 5     0555,555555
3          Cai         Caio        Via Caini 1    0888,888888
4          Semproni    Sempronio   Via Sempi 7    0999,999999
```

Per visualizzare anche l'intestazione delle colonne che appaiono, occorre utilizzare il comando '**.header on**':

```
sqlite> .header on[Invio]

sqlite> SELECT * FROM Indirizzi;[Invio]

Codice    Cognome     Nome        Indirizzo      Telefono
--------  ----------  ----------  -------------  -----------
1         Pallino     Pinco       Via Biglie 1   0222,222222
2         Tizi        Tizio       Via Tazi 5     0555,555555
3         Cai         Caio        Via Caini 1    0888,888888
4         Semproni    Sempronio   Via Sempi 7    0999,999999
```

Per ottenere un elenco ordinato in base al cognome e al nome (in caso di ambiguità), lo stesso comando si completa nel modo seguente:

```
sqlite> SELECT * FROM Indirizzi ORDER BY Cognome, Nome;[Invio]

  Codice    Cognome     Nome         Indirizzo      Telefono
  --------  ----------  ----------  -------------  -----------
  3         Cai         Caio        Via Caini 1    0888,888888
  1         Pallino     Pinco       Via Biglie 1   0222,222222
  4         Semproni    Sempronio   Via Sempi 7    0999,999999
  2         Tizi        Tizio       Via Tazi 5     0555,555555
```

La selezione degli attributi permette di ottenere un risultato che contenga solo quelli desiderati, permettendo anche di cambiarne l'intestazione. L'esempio seguente permette di mostrare solo i nominativi e il telefono, cambiando un po' le intestazioni:

```
sqlite> SELECT Cognome as cognomi, Nome as nomi,[Invio]

   ...>         Telefono as numeri_telefonici[Invio]

   ...>         FROM Indirizzi;[Invio]
```

Quello che si ottiene è simile all'elenco seguente:

```
  cognomi     nomi        numeri_telefonici
  ----------  ----------  -----------------
  Pallino     Pinco       0222,222222
  Tizi        Tizio       0555,555555
  Cai         Caio        0888,888888
  Semproni    Sempronio   0999,999999
```

La selezione delle tuple può essere fatta attraverso la condizione che segue la parola chiave '**WHERE**'. Nell'esempio seguente vengono selezionate le tuple in cui l'iniziale dei cognomi è compresa tra '**N**' e '**T**':

```
sqlite> SELECT * FROM Indirizzi[Invio]
```

```
   ...>        WHERE Cognome >= 'N' AND Cognome <= 'T'; [Invio]
```

Dall'elenco che si ottiene, si osserva che '**Caio**' è stato escluso:

```
Codice     Cognome      Nome         Indirizzo       Telefono
--------   ----------   ----------   -------------   -----------
1          Pallino      Pinco        Via Biglie 1    0222,222222
2          Tizi         Tizio        Via Tazi 5      0555,555555
4          Semproni     Sempronio    Via Sempi 7     0999,999999
```

Per evitare ambiguità possono essere indicati i nomi degli attributi prefissati dal nome della relazione a cui appartengono, separando le due parti con l'operatore punto ('.'). Nell'esempio seguente si selezionano solo il cognome, il nome e il numero telefonico, specificando il nome della relazione a cui appartengono gli attributi:

```
sqlite> SELECT Indirizzi.Cognome, Indirizzi.Nome,
Indirizzi.Telefono [Invio]

   ...>        FROM Indirizzi; [Invio]
```

Ecco il risultato:

```
Cognome      Nomi         Telefono
----------   ----------   -----------
Pallino      Pinco        0222,222222
Tizi         Tizio        0555,555555
Cai          Caio         0888,888888
Semproni     Sempronio    0999,999999
```

77.2.6 Interrogazioni simultanee di più relazioni

«

Se dopo la parola chiave '**FROM**' si indicano più relazioni (ciò vale anche se si indica più volte la stessa relazione), si intende fare riferimento a una relazione generata dal «prodotto» di queste. Si immagi-

ni di abbinare alla relazione '**Indirizzi**' la relazione '**Presenze**' contenente i dati visibili nella figura 77.13.

Figura 77.13. La relazione '**Presenze (Codice, Giorno, Ingresso, Uscita)**'.

Presenze			
Codice	**Giorno**	**Ingresso**	**Uscita**
1	01/01/2012	07:30	13:30
2	01/01/2012	07:35	13:37
3	01/01/2012	07:45	14:00
4	01/01/2012	08:30	16:30
1	01/02/2012	07:35	13:38
2	01/02/2012	08:35	14:37
4	01/02/2012	07:30	13:30

Ecco le istruzioni per crearla e per inserire la prima tupla dell'esempio:

```
sqlite> CREATE TABLE Presenze (Codice INTEGER, Giorno DATE,
[Invio]

   ...>                              Ingresso TIME, USCITA Time);
[Invio]

sqlite> INSERT INTO Presenze[Invio]

   ...>     VALUES (1, '2012-01-01', '07:30', '13:30');[Invio]
```

Come si può intendere, il primo attributo, '**Codice**', serve a identificare la persona per la quale è stata fatta l'annotazione dell'ingresso

e dell'uscita. Tale codice viene interpretato in base al contenuto della relazione '**Indirizzi**'. Si immagini di volere ottenere un elenco contenente tutti gli ingressi e le uscite, indicando chiaramente il cognome e il nome della persona a cui si riferiscono.

```
sqlite> SELECT Presenze.Giorno, Presenze.Ingresso,
Presenze.Uscita, [Invio]
   ...>        Indirizzi.Cognome, Indirizzi.Nome[Invio]
   ...>        FROM Presenze, Indirizzi[Invio]
   ...>        WHERE Presenze.Codice = Indirizzi.Codice; [Invio]
```

Ecco quello che si dovrebbe ottenere:

```
giorno      ingresso    uscita      cognome     nome
----------  ----------  ----------  ----------  ----------
01-01-2012    07:30:00    13:30:00    Pallino     Pinco
01-01-2012    07:35:00    13:37:00    Tizi        Tizio
01-01-2012    07:45:00    14:00:00    Cai         Caio
01-01-2012    08:30:00    16:30:00    Semproni    Sempronio
01-02-2012    07:35:00    13:38:00    Pallino     Pinco
01-02-2012    08:35:00    14:37:00    Tizio       Tizi
01-02-2012    07:40:00    13:30:00    Semproni    Sempronio
```

77.2.7 Alias

«

Una stessa relazione può essere presa in considerazione come se si trattasse di due o più relazioni differenti. Per distinguere tra questi punti di vista diversi, si devono usare degli alias, che sono in pratica dei nomi alternativi. Gli alias si possono usare anche solo per questioni di leggibilità. L'esempio seguente è la semplice ripetizione di quello mostrato nella sezione precedente, con l'aggiunta però della definizione degli alias '**Pre**' e '**Nom**'.

```
sqlite> SELECT Pre.Giorno, Pre.Ingresso, Pre.Uscita, [Invio]
   ...>      Nom.Cognome, Nom.Nome [Invio]
   ...>      FROM Presenze AS Pre, Indirizzi AS Nom [Invio]
   ...>      WHERE Pre.Codice = Nom.Codice; [Invio]
```

77.2.8 Viste

Attraverso una vista, è possibile definire una relazione virtuale:

```
sqlite> CREATE VIEW Presenze_dettagliate AS [Invio]
   ...>      SELECT Presenze.Giorno, Presenze.Ingresso, [Invio]
   ...>             Presenze.Uscita, [Invio]
   ...>             Indirizzi.Cognome, Indirizzi.Nome [Invio]
   ...>          FROM Presenze, Indirizzi [Invio]
   ...>          WHERE Presenze.Codice = Indirizzi.Codice;
[Invio]
```

L'esempio mostra la creazione della vista '**Presenze_dettagliate**', ottenuta dalle relazioni '**Presenze**' e '**Indirizzi**'. In pratica, questa vista permette di interrogare direttamente la relazione virtuale '**Presenze_dettagliate**', invece di utilizzare ogni volta un comando '**SELECT**' molto complesso, per ottenere lo stesso risultato.

77.2.9 Aggiornamento delle tuple

La modifica di tuple già esistenti avviene attraverso l'istruzione '**UPDATE**', la cui efficacia viene controllata dalla condizione posta

dopo la parola chiave '**WHERE**'. Se tale condizione manca, l'effetto delle modifiche si riflette su tutte le tuple della relazione.

L'esempio seguente, aggiunge un attributo alla relazione degli indirizzi, per contenere il nome del comune di residenza degli impiegati; successivamente viene inserito il nome del comune '**Sferopoli**' in base al prefisso telefonico.

```
sqlite> ALTER TABLE Indirizzi ADD COLUMN Comune char(30);
[Invio]

sqlite> UPDATE Indirizzi[Invio]

   ...>     SET Comune='Sferopoli'[Invio]

   ...>     WHERE Telefono >= '022' AND Telefono < '023';
[Invio]
```

In pratica, viene aggiornata solo la tupla dell'impiegato '**Pinco Pallino**'.

77.2.10 Cancellazione delle tuple

«

L'esempio seguente elimina dalla relazione delle presenze le tuple riferite alle registrazioni del giorno 01/01/2012 e le eventuali antecedenti.

```
sqlite> DELETE FROM Presenze WHERE Giorno <= '2012-01-01';
[Invio]
```

77.2.11 Inserimento in una relazione esistente

«

L'esempio seguente aggiunge alla relazione dello storico delle presenze le registrazioni vecchie che poi vengono cancellate:

```
sqlite> INSERT INTO PresenzeStorico ([Invio]
```

```
   ...>          PresenzeStorico.Codice, [Invio]
   ...>          PresenzeStorico.Giorno, [Invio]
   ...>          PresenzeStorico.Ingresso, [Invio]
   ...>          PresenzeStorico.Uscita) [Invio]
   ...>      SELECT Presenze.Codice, Presenze.Giorno, [Invio]
   ...>             Presenze.Ingresso, Presenze.Uscita [Invio]
   ...>          FROM Presenze [Invio]
   ...>          WHERE Presenze.Giorno <= '2012-01-01'; [Invio]
sqlite> DELETE FROM Presenze [Invio]
   ...>         WHERE Giorno <= '2012-01-01'; [Invio]
```

77.3 Riferimenti

«

- *SQLite*, *http://www.sqlite.org*

[1] **SQLite** dominio pubblico

ODBC

ODBC, ovvero *Open database connectivity* è un metodo standardizzato per l'accesso ai DBMS. Si attua inserendo un servizio intermedio, tra i DBMS e le applicazioni che devono accedere ai dati: le applicazioni comunicano con il servizio ODBC, mentre il servizio ODBC comunica con i DBMS sottostanti, preoccupandosi di adattarsi alle loro particolarità. Ciò consente di scrivere applicazioni che, attraverso ODBC, sono in grado di utilizzare qualsiasi DBMS con cui il servizio ODBC sia in grado di interagire.

Generalmente, il sistema ODBC non fa tutto da solo, in quanto di solito si avvale di librerie aggiuntive (*driver*) per la gestione dei DBMS reali.

78.1 DSN

Un DSN, ovvero *Data source name*, è una base di dati virtuale, del sistema ODBC, individuata da un nome.

Generalmente, un DSN fa riferimento a una base di dati di un certo DBMS reale, con cui il sistema ODBC è in grado di interagire; tuttavia, teoricamente, potrebbe trattarsi di qualunque cosa in grado di comportarsi come una base di dati vera e propria.

78.2 unixODBC

Nei sistemi Unix è disponibile unixODBC,[1] che consente di interagire con un discreto numero di DBMS comuni, attraverso delle librerie aggiuntive per la gestione dei vari DBMS, alcune delle quali vengono elencate qui brevemente:

- PostgreSQL ODBC,[2] per la comunicazione con DBMS PostgreSQL;
- MyODBC,[3] per la comunicazione con DBMS MySQL.

Una volta installato unixODBC e le librerie per la comunicazione con i DBMS di proprio interesse, occorre provvedere a configurare unixODBC in modo da poterle utilizzare. Per fare questo si interviene nel file '`/etc/odbcinst.ini`'; l'esempio seguente riguarda le librerie per comunicare con MySQL e PostgreSQL rispettivamente:

```
[MySQL]
Description     = MySQL driver
Driver          = /usr/lib/odbc/libmyodbc.so
Setup           = /usr/lib/odbc/libodbcmyS.so
CPTimeout       =
CPReuse         =
FileUsage       = 1

[PostgreSQL]
Description     = PostgreSQL ODBC driver
Driver          = /usr/lib/odbc/psqlodbc.so
Setup           = /usr/lib/odbc/libodbcpsqlS.so
Debug           = 0
CommLog         = 1
FileUsage       = 1
```

Come si può vedere e intuire, alcune direttive non sono comuni per tutti i tipi di DBMS; in pratica, l'inserimento di una sezione per l'accesso a un DBMS richiede un modello da copiare e adattare, per quel caso specifico.

Una volta configurate le librerie per l'accesso ai DBMS, è possibile

definire dei DSN, con i quali fare riferimento a delle basi di dati reali presso tali DBMS. unixODBC offre tre modi per memorizzare le informazioni sui DSN, in base al campo di azione previsto per questi: il sistema nel suo insieme, una rete locale, il singolo utente.

Attraverso il file '`/etc/odbc.ini`' è possibile dichiarare dei DSN validi nell'ambito del sistema locale, per tutti gli utenti; attraverso dei file che corrispondono al modello '`/etc/ODBCDataSources/`***nome***`.dsn`', è possibile dichiarare un DSN che, teoricamente, potrebbe essere condiviso da più elaboratori in una rete locale, con la stessa condivisione della directory '`/etc/ODBCDataSources/`'; attraverso i file '`~/.odbc.ini`', ogni utente può dichiarare i propri DSN personali.

Questi file di configurazione dei DSN, possono contenere direttive raggruppate in sezioni, corrispondenti al nome del DSN. Per esempio, l'estratto seguente dichiara l'accesso alla base di dati, presso l'elaboratore locale, denominata '**`nanodb`**', accedendo con l'utenza '**`pgnanouser`**' (secondo il DBMS); il nome del DSN è '**`mio`**':

```
[mio]
Description             = PostgreSQL
Driver                  = PostgreSQL
Trace                   = No
TraceFile               =
Database                = nanodb
Servername              = localhost
Username                = pgnanouser
Password                =
Port                    = 5432
Protocol                = 6.4
ReadOnly                = No
RowVersioning           = No
```

```
ShowSystemTables        = No
ShowOidColumn           = No
FakeOidIndex            = No
ConnSettings            =
```

L'estratto seguente, invece, dichiara l'accesso alla base di dati, presso l'elaboratore locale, denominata '**nanodb**', accedendo con l'utenza '**mynanouser**' (secondo il DBMS); il nome del DSN è '**mio2**':

```
[mio2]
Description     = MySQL
Driver          = MySQL
Server          = localhost
Database        = nanodb
Username        = mynanouser
Port            =
Socket          =
Option          =
Stmt            =
```

78.3 ODBCConfig

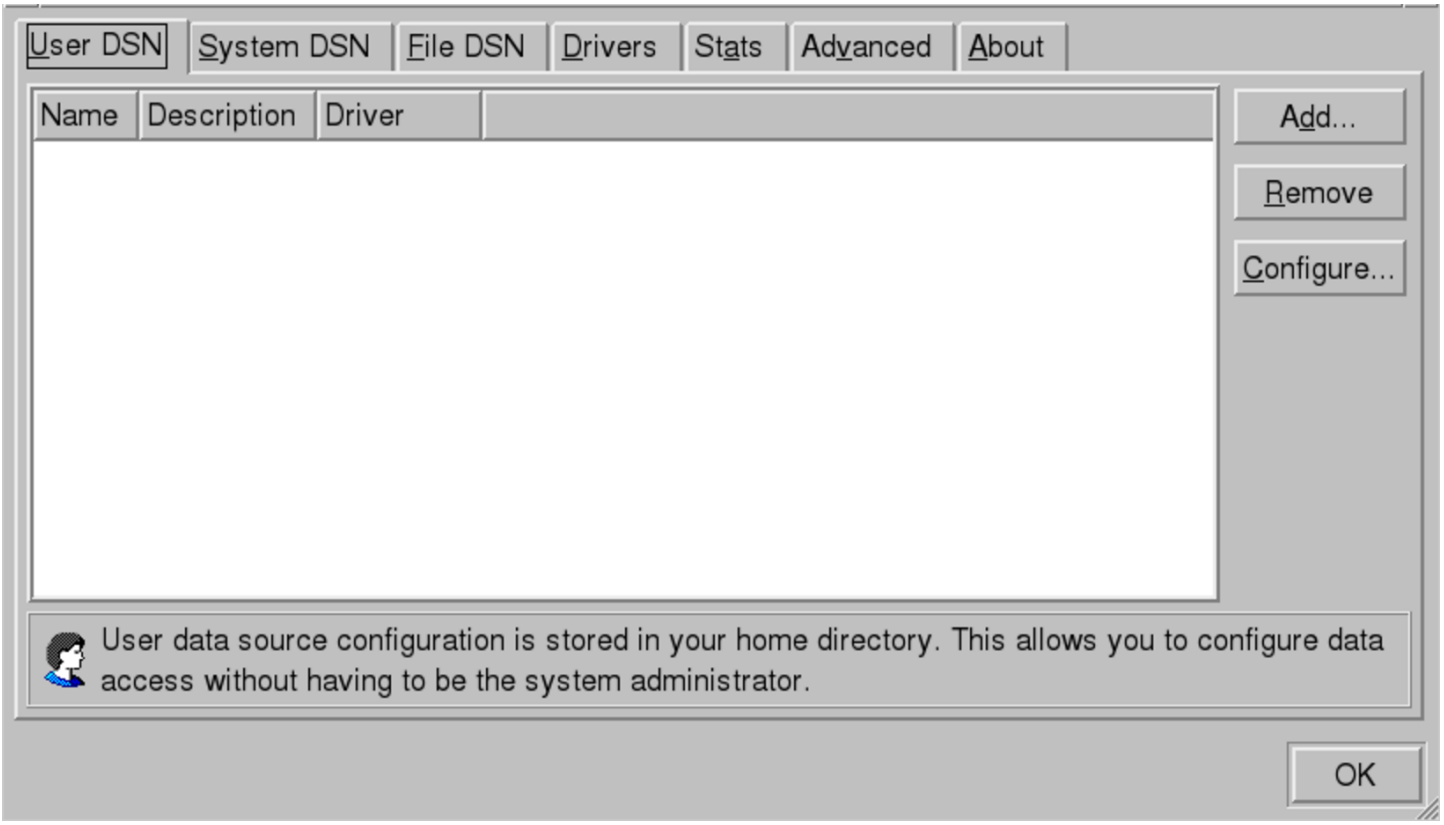
«

Il programma ODBCConfig, che fa parte di unixODBC, consente di configurare i DSN in modo guidato, proponendo dei valori predefiniti appropriati al tipo di DBMS a cui questi vanno associati. Eventualmente, sarebbe possibile anche intervenire nella configurazione di '`/etc/odbcinst.ini`', ma questo non è conveniente, perché in tal caso manca la guida necessaria. Il programma si avvia generalmente senza argomenti:

```
ODBCConfig
```

Se il programma viene avviato con i privilegi necessari, può intervenire nella configurazione di ‘`/etc/odbc.ini`’ o dei file contenuti nella directory ‘`/etc/ODBCDataSources/`’, altrimenti può operare esclusivamente nel file personale ‘`~/.odbc.ini`’.

Figura 78.4. Aspetto di ODBCConfig all’avvio.

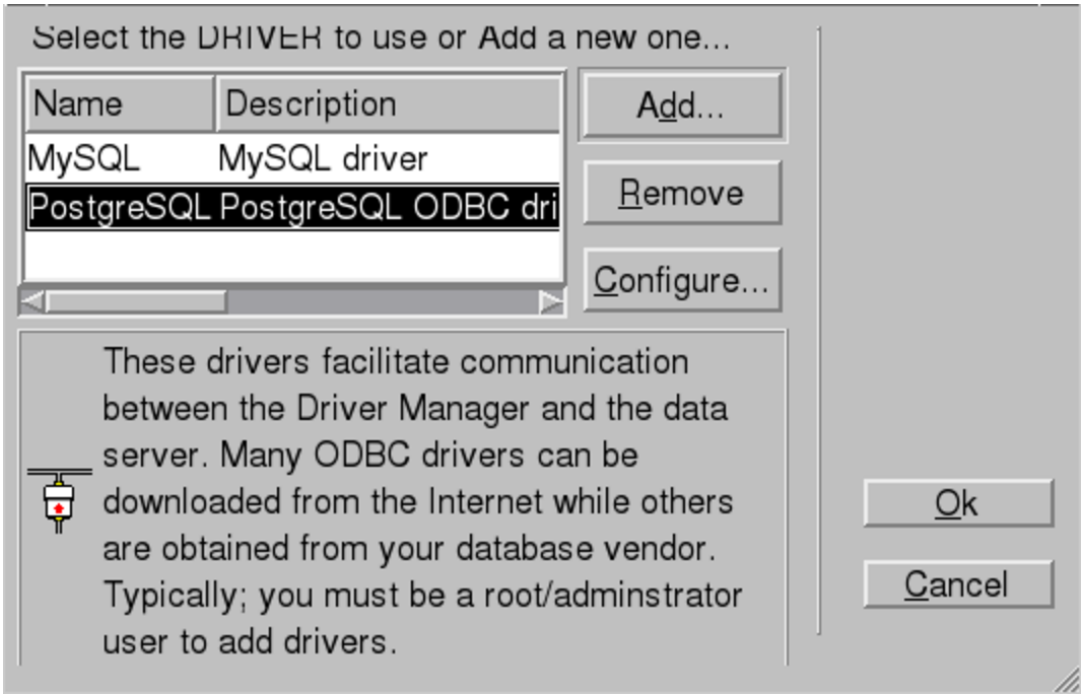


Osservando il programma in funzione, si vedono alcuni lembi posti sul lato superiore. I primi tre (*User DSN*, *System DSN*, *File DSN*) permettono di selezionare una scheda riferita, rispettivamente, alla configurazione dei DSN personali (‘`~/.odbc.ini`’), di quelli di sistema (‘`/etc/odbc.ini`’) e di quelli condivisibili (‘`/etc/ODBCDataSources/`’). La configurazione con una qualsiasi di queste tre schede è uniforme alle altre; quello che cambia sono i privilegi

necessari a modificare i file di configurazione rispettivi.

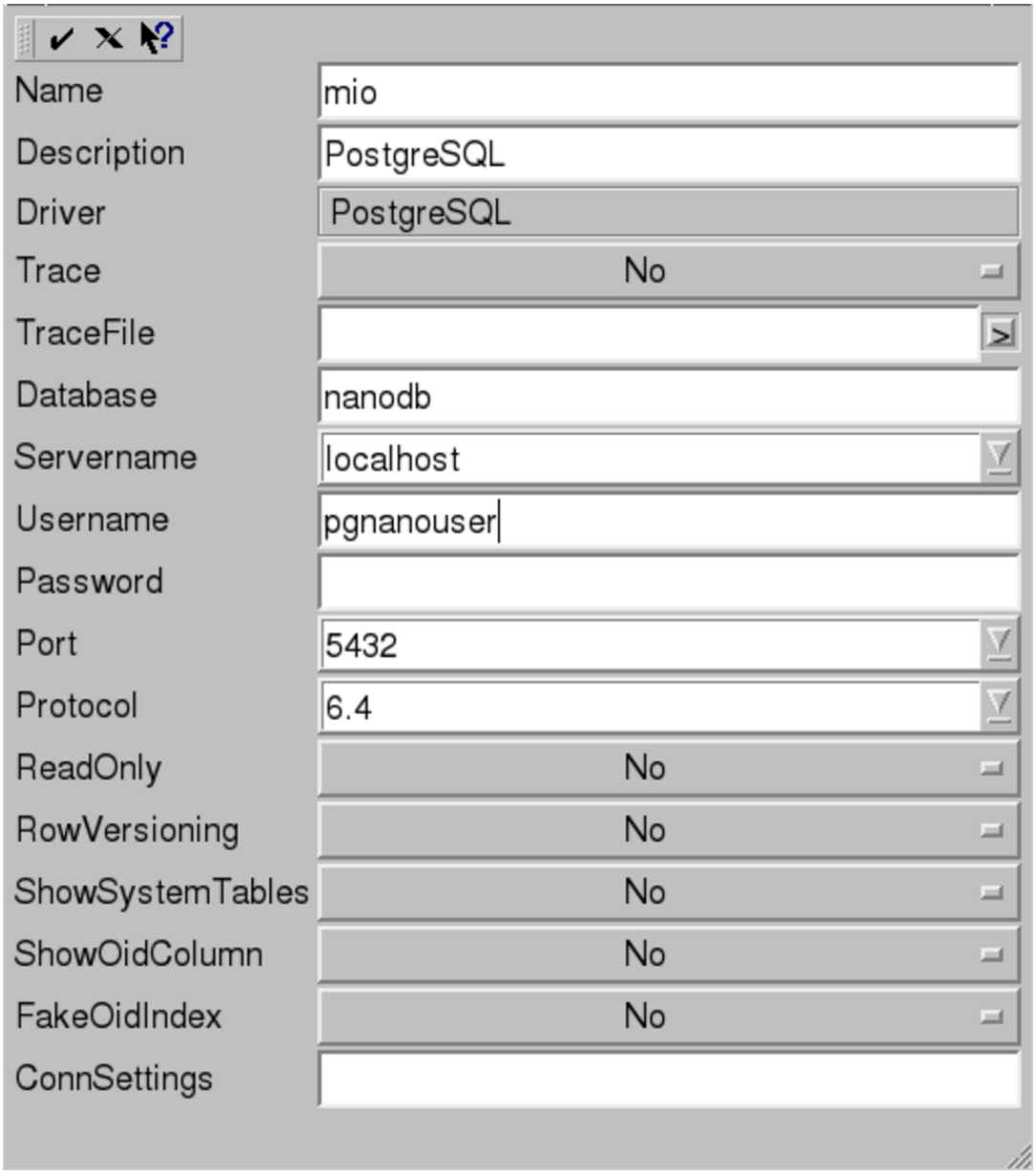
Avendo selezionato una delle tre schede che consentono di intervenire sui DSN, sul lato destro della finestra appaiono dei pulsanti grafici, con i quali è possibile creare, eliminare o modificare dei DSN. Volendo creare un DSN (pulsante ADD), appare la richiesta di specificare il tipo di DBMS (*driver*).

Figura 78.5. La maschera con cui si specifica la libreria adatta a comunicare con il DBMS di proprio interesse. Una volta evidenziato, come si vede in questo caso a proposito di PostgreSQL, si deve selezionare il pulsante grafico OK.



Successivamente viene proposta una maschera che riproduce, sostanzialmente, le direttive specifiche per quel tipo di libreria, da inserire nel file di configurazione. Fortunatamente, la maschera contiene già dei valori predefiniti appropriati per la maggior parte delle direttive.

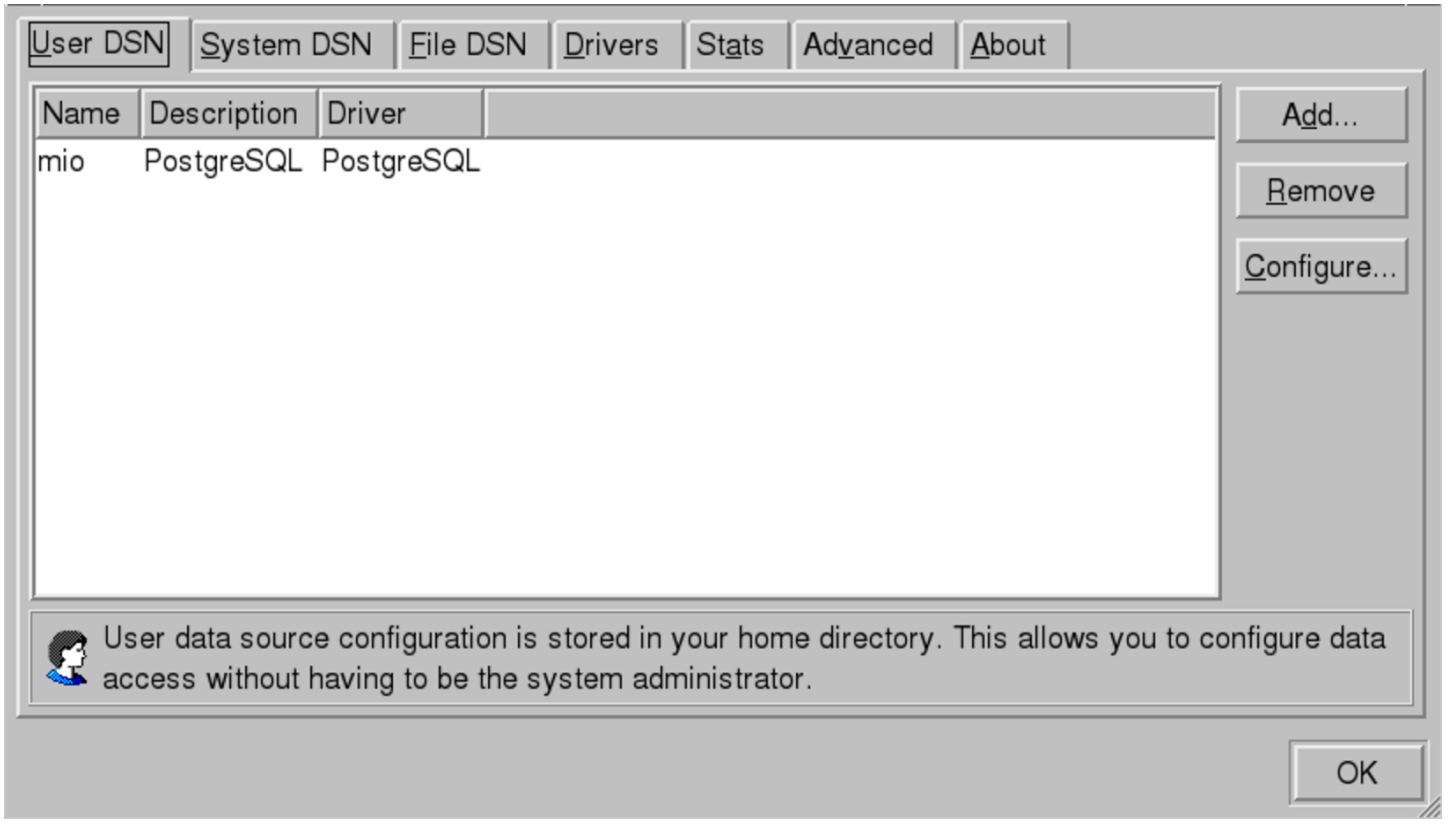
Figura 78.6. La maschera con cui si definiscono le direttive per il DSN, nel caso della libreria per il collegamento a und DBMS PostgreSQL. In questo caso viene dichiarato il DSN denominato '**mio**', abbinato alla base di dati '**nanodb**', presso l'elaboratore locale, a cui si accede con l'utenza '**pgnanouser**' (la parola d'ordine, ammesso che sia necessaria per accedere, rimane da specificare al momento del collegamento.



Nella maschera di inserimento delle direttive, appare un menù di icone, di solito sul lato superiore sinistro. Con l'icona che mostra una

«X», si annullano le modifiche, mentre con quella che assomiglia a una «V», si confermano gli inserimenti.

Figura 78.7. Aspetto di ODBCConfig dopo l'inserimento del DSN '**mio**' nella configurazione personale dell'utente.



Tra le altre schede del programma, è interessante osservare ciò che è contenuto in quella denominata *Advanced*. Lì dovrebbe essere visibile il percorso di un file usato per annotare l'esito delle interrogazioni avvenute con le basi di dati reali, per poter risalire alla causa di un problema, quando l'utilizzo di ODBC sembra fallire senza un motivo apparente. In ogni caso, questo file dovrebbe corrispondere a '`/tmp/sql.log`'.

78.4 Accesso a ODBC tramite «isql» o «iusql»

unixODBC include due programmi equivalenti, per accedere a un DSN attraverso istruzioni SQL impartite interattivamente. Attraverso questi programmi si può verificare in pratica il funzionamento di un certo DSN:

```
isql nome_dsn [nome_utente [parola_d'ordine]] [opzioni]
```

```
iusql nome_dsn [nome_utente [parola_d'ordine]] [opzioni]
```

La differenza tra i due programmi dovrebbe consistere nella migliore disposizione del secondo verso la codifica universale.

Dalla sintassi mostrata sull'uso dei due programmi, si può osservare che è obbligatorio l'inserimento del nome del DSN con cui ci si vuole connettere, mentre gli altri dati sono facoltativi, perché potrebbero essere memorizzati nella configurazione del DSN stesso, oppure, nel caso della parola d'ordine, potrebbe non essere richiesta dal DBMS per accedere. A titolo di esempio, si suppone di collegarsi al DSN '**mio**', per il quale è già stato specificato il nominativo utente da usare e la parola d'ordine non è richiesta:

```
$ isql mio[Invio]
```

```
+--------------------------------------+
| Connected!                           |
|                                      |
| sql-statement                        |
| help [tablename]                     |
| quit                                 |
|                                      |
+--------------------------------------+
```

Come si vede, viene suggerito ciò che si può fare: inserire istruzioni SQL, usare il comando '**help**' o '**quit**'. A questo punto non c'è molto da spiegare; si comprende che possono essere impartite delle istruzioni SQL (secondo i canoni di ODBC) e che al termine si può concludere con il comando '**quit**'.

`SQL>` **quit** [*Invio*]

Il problema nell'uso di un programma come questo, semmai, sta nel fatto che, di fronte a un errore, la spiegazione che si ottiene è estremamente scarna e occorre leggere il file in cui i messaggi del DBMS reale vengono scaricati (di solito è '`/tmp/sql.log`').

78.5 Riferimenti

«

- *unixODBC*, *http://www.unixodbc.org*

[1] **unixODBC** GNU GPL e GNU LGPL

[2] **PostgreSQL ODBC** GNU GPL

[3] **MyODBC** GNU GPL

79.1 Creazione ed eliminazione delle relazioni 2140
79.1.1 Creazione di una relazione 2141
79.1.2 Eliminazione di una relazione 2143
79.1.3 Verifica sulla creazione e popolazione della relazione «Articoli» 2144
79.1.4 Verifica sulla creazione e popolazione della relazione «Causali» 2145
79.1.5 Verifica sulla creazione e popolazione della relazione «Fornitori» 2147
79.1.6 Verifica sulla creazione e popolazione della relazione «Clienti» 2149
79.1.7 Verifica sulla creazione e popolazione della relazione «Movimenti» 2152
79.1.8 Conclusione 2154
79.2 Interrogazione semplice di una relazione 2155
79.2.1 Interrogazione completa 2155
79.2.2 Interrogazione con selezione di alcuni attributi ... 2157
79.2.3 Stampa del contenuto di una relazione 2159
79.2.4 Verifica sull'interrogazione della relazione «Articoli» 2161
79.2.5 Verifica sull'interrogazione delle relazioni «Fornitori» e «Clienti» 2162
79.2.6 Conclusione 2164

79.3 Interrogazione ordinata di una relazione 2164
79.3.1 Interrogazione ordinata 2164
79.3.2 Verifica sull'interrogazione ordinata della relazione «Articoli» 2166
79.3.3 Verifica sull'interrogazione ordinata della relazione «Clienti» 2167
79.3.4 Verifica sull'interrogazione ordinata della relazione «Causali» 2169
79.4 Interrogazione selettiva di una relazione 2170
79.4.1 Interrogazione selettiva 2173
79.4.2 Verifica sull'interrogazione selettiva della relazione «Articoli» 2174
79.4.3 Verifica sull'interrogazione selettiva e ordinata della relazione «Articoli» 2176
79.4.4 Verifica sull'interrogazione selettiva della relazione «Causali» 2177
79.5 Interrogazioni simultanee di più relazioni 2179
79.5.1 Interrogazione simultanea delle relazioni «Movimenti» e «Articoli» 2179
79.5.2 Interrogazione simultanea delle relazioni «Movimenti», «Articoli» e «Causali» 2181
79.5.3 Verifica sull'interrogazione simultanea delle relazioni «Movimenti» e «Causali» 2182
79.5.4 Verifica sull'interrogazione simultanea delle relazioni «Movimenti», «Causali» e «Clienti» 2184

79.5.5 Verifica sull'interrogazione ordinata e simultanea delle relazioni «Movimenti», «Causali» e «Clienti» 2185

79.6 Interrogazioni simultanee di più relazioni e alias 2187

79.6.1 Interrogazione simultanea delle relazioni «Movimenti», «Articoli» e «Causali» 2187

79.6.2 Verifica sull'interrogazione simultanea delle relazioni «Movimenti» e «Causali» 2189

79.6.3 Verifica sull'interrogazione simultanea delle relazioni «Movimenti», «Causali» e «Clienti» 2190

79.6.4 Conclusione 2192

79.7 Viste 2192

79.7.1 Creazione della vista «Listino» 2192

79.7.2 Creazione della vista «Resi» 2195

79.7.3 Verifica sulla creazione della vista «Acquisti» ... 2197

79.7.4 Verifica sulla creazione della vista «Vendite» ... 2199

79.7.5 Conclusione 2201

79.8 Modifica del contenuto delle tuple 2202

79.8.1 Modifica di una causale di magazzino 2202

79.8.2 Modifica di diverse causali di magazzino 2204

79.8.3 Verifica sulla modifica della relazione «Articoli» 2206

79.8.4 Verifica sulla modifica delle relazioni «Clienti» e «Fornitori» 2209

79.8.5 Conclusione 2211

79.9 Eliminazione delle tuple 2211

79.9.1 Cancellazione di una causale di magazzino 2211
79.9.2 Cancellazione di diverse causali di magazzino .. 2213
79.9.3 Verifica sulla cancellazione di alcuni articoli 2215
79.9.4 Conclusione 2217
79.10 Grilletti per il controllo del dominio degli attributi .. 2217
79.10.1 Creazione dei grilletti «Causali_ins» e «Causali_upd» 2218
79.10.2 Creazione del grilletto «Articoli_ins» e «Articoli_upd» 2221
79.10.3 Verifica sulla creazione dei grilletti «Movimenti_ins» e «Movimenti_upd» 2224
79.10.4 Conclusione 2226
79.11 Grilletti per il controllo della validità esterna 2227
79.11.1 Controllo del codice articolo tra la relazione «Movimenti» e la relazione «Articoli» 2227
79.11.2 Controllo del codice cliente tra la relazione «Movimenti» e la relazione «Clienti» 2230
79.11.3 Verifica sulla creazione dei grilletti «Movimenti_ins», «Movimenti_upd» e «Causali_del» 2234
79.11.4 Verifica sulla creazione dei grilletti «Movimenti_ins», «Movimenti_upd» e «Fornitori_del» 2236
79.11.5 Conclusione 2239
79.12 Selezione di attributi virtuali, ottenuti da un'espressione 2240

79.12.1 Interrogazione della relazione «Movimenti» in modo da ottenere il valore unitario 2242
79.12.2 Vista della relazione «Movimenti» in modo da ottenere il valore unitario 2244
79.12.3 Verifica sulla creazione della vista «MovimentiExtra» 2246
79.12.4 Conclusione 2250
79.13 Aggregazioni 2250
79.13.1 Aggregazioni banali 2252
79.13.2 Verifica sulla creazione della vista «SituazioneMagazzino» 2254
79.13.3 Verifica sulla creazione della vista «SituazioneMagazzino» 2256
79.13.4 Verifica sulla creazione della vista «SituazioneMagazzino» 2259
79.13.5 Conclusione 2261
79.14 Inserimento automatico del costo medio 2262
79.14.1 Vista «CostoMedioValido» 2263
79.14.2 Grilletto «ValorizzazioneScarichi» 2263

Prima di poter iniziare a eseguire gli esercizi di questo capitolo, dedicato alle basi di dati e al linguaggio SQL, è necessario verificare di disporre degli strumenti adatti ed essere sicuri di saperli utilizzare.

Per facilitare l'esecuzione di queste esercitazioni, sia gli esercizi, sia le verifiche sono realizzabili con SQLite, attraverso il program-

ma '**sqlite3**'. Pertanto gli esercizi prevedono l'uso di basi di dati personali, ognuna contenuta tutta in un file.

Le verifiche associate a queste esercitazioni portano a produrre dei fogli stampati, che gli studenti devono avere la cura di controllarle in base a quanto indicato nella traccia delle verifiche stesse, prima della consegna all'insegnante.

Per poter svolgere gli esercizi e le verifiche, ogni studente deve essere in grado di scrivere e modificare file di testo, con un programma adatto (per esempio va bene il programma Gedit). In questi file di testo vanno inserite le istruzioni SQL necessarie allo svolgimento del lavoro; per evitare confusione, i file che contengono codice SQL vengono nominati con l'estensione '`.sql`'.

Per eseguire le istruzioni SQL contenute in un file, si usa il programma '**sqlite3**' nel modo seguente:

`$` **sqlite3** ***file_db*** **<** ***file_sql*** [*Invio*]

In questo caso, le istruzioni contenute nel file ***file_sql***, vengono applicate alla base di dati contenuta nel file ***file_db***.

Per essere certi di sapere usare gli strumenti occorre fare una prova. Si realizzi il file di testo denominato '`prova-istruzioni.sql`', contenente quanto segue, sostituendo le metavariabili ***cognome***, ***nome***, ***classe*** e ***data*** con qualcosa di appropriato:

```
-- Esercizio di prova di: cognome  nome  classe
-- Data: data
-- File: prova-istruzioni.sql

CREATE TABLE Prova (Codice  INTEGER,
                    Cognome VARCHAR(60),
                    Nome    VARCHAR(60));
```

Una volta salvato il file con il nome stabilito, lo si esegue nella base di dati contenuta nel file '`prova.db`'. Dal momento che il file '`prova.db`' non esiste, essendo la prima volta che viene utilizzato questo nome, l'esecuzione delle istruzioni comporta automaticamente la creazione della base di dati relativa:

```
$ sqlite3 prova.db < prova-istruzioni.sql[Invio]
```

Se non si vedono segnalazioni di alcun genere, le istruzioni contenute nel file '`prova-istruzioni.sql`' sono state eseguite tutte con successo.

Le istruzioni contenute nel file '`prova-istruzioni.sql`' servono a produrre una ***relazione***, denominata '**`Prova`**', contenente alcuni ***attributi*** ('**`Codice`**', '**`Cognome`**' e '**`Nome`**').

Se il file contenente le istruzioni SQL contiene degli errori, o viene eseguito quando ciò non deve essere fatto, è probabile vedere apparire dei messaggi, che vanno letti attentamente. Per esempio, se venisse eseguito nuovamente il file '`prova-istruzioni.sql`' nella stessa base di dati, si otterrebbe una segnalazione che avvisa del fatto che la relazione '**`Prova`**' esiste già (e non può essere creata nuovamente):

```
$ sqlite3 prova.db < prova-istruzioni.sql[Invio]
```

```
CREATE TABLE Prova (Codice  INTEGER,
                    Cognome VARCHAR(60),
                    Nome    VARCHAR(60));
SQL error: table Prova already exists
```

Il programma '**sqlite3**' può essere usato anche interattivamente. Per farlo, si avvia senza indicare il file contenente le istruzioni SQL. Si proceda in questo modo:

```
$ sqlite3 prova.db[Invio]
```

A questo punto appare l'invito del programma '**sqlite3**', che indica la sua attesa di comandi o di istruzioni SQL, digitati direttamente:

```
sqlite>
```

Con il comando '**.schema**' (si osservi il fatto che il comando inizia con un punto) è possibile visualizzare l'elenco delle relazioni esistenti, in forma di istruzione SQL. Si proceda inserendo questo comando:

```
sqlite> .schema[Invio]

CREATE TABLE Prova (Codice  INTEGER,
                    Cognome VARCHAR(60),
                    Nome    VARCHAR(60));
```

Si proceda inserendo l'istruzione necessaria a eliminare la relazione '**Prova**' creata poco prima:

```
sqlite> DROP TABLE Prova;[Invio]
```

Si conclude con il funzionamento di '**sqlite3**' con il comando '**.quit**':

```
sqlite> .quit [Invio]
```

Prima di passare alle sezioni successive, vanno eliminati i file ‘`prova-istruzioni.sql`’ e ‘`prova.db`’ che non servono più.

In questi esercizi vengono creati i file seguenti, elencati in ordine alfabetico, con il riferimento alla sezione in cui sono utilizzati:

cancella-articoli.sql 2215
creazione-articoli.sql 2144
creazione-causali.sql 2145
creazione-clienti.sql 2149
creazione-fornitori.sql 2147
creazione-movimenti.sql 2152
grilletti-articoli.sql 2221
grilletti-causali.sql 2218
grilletti-movimenti.sql 2224
grilletti-movimenti-articoli.sql 2227
grilletti-movimenti-causali.sql 2234
grilletti-movimenti-clienti.sql 2230
grilletti-movimenti-fornitori.sql 2236
grilletto-valorizzazione-scarichi.sql 2263
interr-artico-01.sql 2161
interr-artico-02.sql 2166
interr-artico-03.sql 2174
interr-artico-04.sql 2176
interr-caus-01.sql 2169
interr-caus-02.sql 2177
interr-clie-01.sql 2167
interr-forn-clie-01.sql 2162
interr-movi-caus-01.sql 2182

interr-movi-caus-02.sql 2189
interr-movi-caus-clienti-01.sql 2184
interr-movi-caus-clienti-02.sql 2185
interr-movi-caus-clienti-03.sql 2190
modifica-articoli.sql 2206
modifica-clienti-fornitori.sql 2209
prova.db 2131
prova-cancella-causali.sql 2211 2213
prova-creazione-articoli.sql 2141
prova-interrogazione-movimenti-vu.sql 2242
prova-interr-movi-arti.sql 2179 2181 2187
prova-istruzioni.sql 2131
prova-modifica-causali.sql 2202 2204
prova-stampa-artico-caus-01.sql 2159
prova-vista-listino.sql 2192
prova-vista-resi.sql 2195
vista-acquisti.sql 2197
vista-costo-medio-valido.sql 2263
vista-movimenti-extra.sql 2244 2246
vista-situazione-magazzino-1.sql 2254
vista-situazione-magazzino-2.sql 2256
vista-situazione-magazzino-3.sql 2259
vista-vendite.sql 2199

79.1 Creazione ed eliminazione delle relazioni

«

Una ***relazione*** è un insieme di ***tuple***, suddivise in ***attributi***, tutte nello stesso modo. Una relazione si rappresenta normalmente in forma

tabellare, dove le tuple sono costituite dalle righe e gli attributi dalle colonne.

Figura 79.4. Relazione «Articoli (Articolo, Descrizione, UM, Listino, ScortaMin)».

Articolo	**Descrizione**	**UM**	**Listino**	**ScortaMin**
1	Dischetti da 9 cm 1440 Kibyte	pz	0,20	500
2	Dischetti da 9 cm 1440 Kibyte colorati	pz	0,25	500
101	CD-R 16x	pz	0,50	500
102	CD-R 52x	pz	1,00	500
201	CD-RW 4x	pz	1,00	200
202	CD-RW 8x	pz	1,50	200
301	DVD-R 8x	pz	1,00	200
302	DVD-R 16x	pz	2,00	200
401	DVD+R 8x	pz	1,00	200
402	DVD+R 16x	pz	2,00	200
501	DVD-RW 8x	pz	2,00	200
601	DVD+RW 8x	pz	2,00	200

I valori che si possono inserire nelle celle della tabella dipendono dal ***dominio*** dell'attributo relativo. Per esempio, l'attributo '**`Listino`**' (corrispondente alla quarta colonna) della relazione '**`Articoli`**', può contenere solo valori numerici positivi, con un massimo di due decimali.

79.1.1 Creazione di una relazione

Con l'ausilio di un programma per la scrittura e la modifica di file di testo puro, si crei il file '`prova-creazione-articoli.`

`sql`', contenente il testo seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Creazione della relazione "Articoli"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: prova-creazione-articoli.sql

CREATE TABLE Articoli (Articolo    INTEGER        NOT NULL,
                       Descrizione CHAR(60)       NOT NULL,
                       UM          CHAR(7)        DEFAULT 'pz',
                       Listino     NUMERIC(14,2)  DEFAULT 0,
                       ScortaMin   NUMERIC(12,3)  DEFAULT 0,
                       PRIMARY KEY (Articolo));

INSERT INTO Articoli VALUES
    (1, 'Dischetti da 9 cm 1440 Kibyte', 'pz', 0.2, 500);
INSERT INTO Articoli VALUES
    (2, 'Dischetti da 9 cm 1440 Kibyte colorati', 'pz', 0.25, 500);
```

L'istruzione '**`CREATE TABLE`**' permette la creazione della relazione '**`Articoli`**', stabilendo dei vincoli, per cui gli attributi '**`Articolo`**' e '**`Descrizione`**' non possono contenere un valore nullo; inoltre viene stabilito il valore predefinito per gli altri attributi. Si stabilisce anche che l'attributo '**`Articolo`**' deve essere una chiave primaria, comportando la necessità che non appaiano tuple con lo stesso codice articolo.

Le istruzioni '**`INSERT`**', inseriscono le prime due tuple della relazione. A questo proposito, si osservi che i dati numerici, come il prezzo di listino e il livello della scorta minima, si indicano così come sono, con l'accortezza di usare **il punto per la separazione dei decimali**, mentre **le stringhe** (le informazioni testuali) **vanno delimitate da apici singoli**.

Si controlli di avere scritto il file ‘`prova-creazione-articoli.sql`’ in modo corretto, rispettando anche la punteggiatura; si controlli di avere salvato il file con li nome previsto, quindi si proceda con il comando seguente:

```
$ sqlite3 mag.db < prova-creazione-articoli.sql[Invio]
```

Se il programma mostra dei messaggi, si tratta di errori, che devono essere verificati attentamente, prima di proseguire.

Si ricorda che il file ‘`mag.db`’, contenente la base di dati, viene creato automaticamente se non dovesse già essere presente.

79.1.2 Eliminazione di una relazione

Per eliminare una relazione si usa l’istruzione ‘**`DROP TABLE`**’, come nell’esempio seguente:

```
DROP TABLE Articoli;
```

Si vuole eliminare la relazione ‘**`Articoli`**’ appena creata nella base di dati contenuta nel file ‘`mag.db`’, ma trattandosi di un’operazione molto semplice, è meglio usare il programma ‘**`sqlite3`**’ in modo interattivo. Si avvii il programma ‘**`sqlite3`**’ e si eseguano i comandi successivi, come descritto qui di seguito, utilizzando anche il comando ‘**`.schema`**’ per avere l’elenco delle relazioni esistenti, prima di cancellare effettivamente quella stabilita:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
```

```
sqlite> .schema [Invio]

  CREATE TABLE Articoli (Articolo     INTEGER         NOT NULL,
                         Descrizione  CHAR(60)        NOT NULL,
                         UM           CHAR(7)         DEFAULT 'pz',
                         Listino      NUMERIC(14,2)   DEFAULT 0,
                         ScortaMin    NUMERIC(12,3)   DEFAULT 0,
                         PRIMARY KEY (Articolo));

sqlite> DROP TABLE Articoli; [Invio]

sqlite> .quit [Invio]
```

Si ricorda che se, a seguito dell'inserimento dell'istruzione '**DROP TABLE**', il programma mostra dei messaggi, si tratta di errori che devono essere verificati attentamente, prima di proseguire.

79.1.3 Verifica sulla creazione e popolazione della relazione «Articoli»

«

Per poter svolgere questa verifica, gli studenti devono essere in grado di realizzare un file di testo contenente codice SQL, con le istruzioni necessarie alla creazione di una relazione e con quelle che permettono l'inserimento delle tuple. Inoltre, devono essere in grado di utilizzare il programma '**sqlite3**' in modo interattivo, per visualizzare l'elenco delle relazioni esistenti nella base di dati e per eliminare una relazione.

Si riprenda il file '`prova-creazione-articoli.sql`' e lo si salvi con il nome '`creazione-articoli.sql`'. Il file '`creazione-articoli.sql`' va poi modificato aggiungendo le istruzioni necessarie a completare l'inserimento degli articoli che sono visibili nella figura 79.4.

Una volta completato e salvato il file ‘`creazione-articoli.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < creazione-articoli.sql[Invio]
```

Se si ottengono degli errori, si deve eliminare la relazione ‘**`Articoli`**’ dalla base di dati contenuta nel file ‘`mag.db`’, utilizzando il programma ‘**`sqlite3`**’ in modo interattivo, quindi, dopo le correzioni, si deve riprovare.

Una volta eseguita l’operazione con successo, si stampi il file ‘`creazione-articoli.sql`’ e lo si consegni per la correzione all’insegnante.

Nella valutazione viene controllata la correttezza del contenuto del file e la coerenza estetica nella scrittura delle istruzioni SQL.

79.1.4 Verifica sulla creazione e popolazione della relazione «Causali»

Prima di svolgere questa verifica, è necessario avere svolto quella precedente, della quale valgono anche gli stessi requisiti.

Si crei il file ‘`creazione-causali.sql`’, inserendo le istruzioni necessarie a creare la relazione ‘**`Causali`**’, con il contenuto che si vede nella figura 79.9, tenendo conto che:

1. l’attributo ‘**`Causale`**’ è di tipo ‘**`INTEGER`**’, non ammette il valore nullo e costituisce la chiave primaria;
2. l’attributo ‘**`Descrizione`**’ è di tipo ‘**`CHAR`**’ a 60 caratteri e non ammette il valore nullo;
3. l’attributo ‘**`Variazione`**’ è di tipo ‘**`NUMERIC`**’, a una sola cifra, senza decimali, con un valore predefinito pari a zero.

Figura 79.9. Relazione **`Causali (Causale, Descrizione, Variazione)`**.

Causale	Descrizione	Variazione
1	Carico per acquisto	+1
2	Scarico per vendita	-1
3	Reso da cliente	+1
4	Reso a fornitore	-1
5	Rettifica aumento acquisto	+1
6	Rettifica aumento vendite	-1
7	Rettifica diminuzione vendite	+1
8	Rettifica diminuzione acquisti	-1
9	Carico da produzione	+1
10	Scarico a produzione	-1
11	Carico da altro magazzino	+1
12	Scarico ad altro magazzino	-1
13	Saldo iniziale	+1

Figura 79.10. Scheletro del file ‘`creazione-causali.sql`’, da completare.

```
-- Creazione della relazione "Causali"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: creazione-causali.sql

CREATE TABLE Causali …

INSERT INTO Causali …
…
```

Una volta completato e salvato il file ‘`creazione-causali.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < creazione-causali.sql [Invio]
```

Se si ottengono degli errori, si deve eliminare la relazione '**Causali**' dalla base di dati contenuta nel file '`prova.db`', utilizzando il programma '**sqlite3**' in modo interattivo, quindi, dopo le correzioni, si deve riprovare.

Una volta eseguita l'operazione con successo, si stampi il file '`creazione-causali.sql`' e lo si consegni per la correzione all'insegnante.

Nella valutazione viene controllata la correttezza del contenuto del file e la coerenza estetica nella scrittura delle istruzioni SQL.

79.1.5 Verifica sulla creazione e popolazione della relazione «Fornitori»

Prima di svolgere questa verifica, è necessario avere svolto quelle precedenti, delle quali valgono anche gli stessi requisiti.

Si crei il file '`creazione-fornitori.sql`', inserendo le istruzioni necessarie a creare la relazione '**Fornitori**', con il contenuto che si vede nella figura 79.11, tenendo conto che:

1. l'attributo '**Fornitore**' è di tipo '**INTEGER**', non ammette il valore nullo e costituisce la chiave primaria;
2. l'attributo '**RagioneSociale**' è di tipo '**VARCHAR**' a 120 caratteri e non ammette il valore nullo;
3. l'attributo '**Paese**' è di tipo '**CHAR**' a 30 caratteri e il suo valore predefinito è '**ITALIA**';
4. l'attributo '**Indirizzo**' è di tipo '**VARCHAR**' a 120 caratteri e non ammette il valore nullo;

5. l'attributo '**CAP**' è di tipo '**CHAR**' a 10 caratteri;
6. l'attributo '**Citta**' è di tipo '**VARCHAR**' a 120 caratteri e non ammette il valore nullo;
7. l'attributo '**Prov**' è di tipo '**CHAR**' a 2 caratteri e il suo valore predefinito è costituito da una stringa di dimensione nulla;
8. l'attributo '**Telefono**' è di tipo '**CHAR**' a 20 caratteri e il suo valore predefinito è costituito da una stringa di dimensione nulla;
9. l'attributo '**Fax**' è di tipo '**CHAR**' a 20 caratteri e il suo valore predefinito è costituito da una stringa di dimensione nulla;
10. l'attributo '**CFPI**' (codice fiscale o partita IVA) è di tipo '**CHAR**' a 30 caratteri e il suo valore predefinito è costituito da una stringa di dimensione nulla.

Figura 79.11. Relazione **Fornitori (Fornitore, RagioneSociale, Paese, Indirizzo, CAP, Citta, Prov, Telefono, Fax, CFPI)**.

Fornitore	RagioneSociale	Paese	Indirizzo	CAP	Citta	Prov	Telefono	Fax	CFPI
1	Tizio Tizi	ITALIA	via Tazio, 11	31100	Treviso	TV	0422,111111	0422,222222	12345678901
2	Caio Cai	ITALIA	via Caino, 22	31033	Castelfranco Veneto	TV	0423,222222	0423,333333	23456789012
3	Sempronio Semproni	ITALIA	via Salina, 33	31057	Silea	TV	0422,333333	0422,444444	34567890123

Figura 79.12. Scheletro del file ‘`creazione-fornitori.sql`’, da completare.

```
-- Creazione della relazione "Fornitori"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: creazione-fornitori.sql

CREATE TABLE Fornitori …

INSERT INTO Fornitori …
…
```

Una volta completato e salvato il file ‘`creazione-fornitori.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < creazione-fornitori.sql [Invio]
```

Se si ottengono degli errori, si deve eliminare la relazione ‘**`Fornitori`**’ dalla base di dati contenuta nel file ‘`mag.db`’, utilizzando il programma ‘**`sqlite3`**’ in modo interattivo, quindi, dopo le correzioni, si deve riprovare.

Una volta eseguita l’operazione con successo, si stampi il file ‘`creazione-fornitori.sql`’ e lo si consegni per la correzione all’insegnante.

Nella valutazione viene controllata la correttezza del contenuto del file e la coerenza estetica nella scrittura delle istruzioni SQL.

79.1.6 Verifica sulla creazione e popolazione della relazione «Clienti»

Prima di svolgere questa verifica, è necessario avere svolto quelle precedenti, delle quali valgono anche gli stessi requisiti.

Si crei il file ‘`creazione-clienti.sql`’, inserendo le istruzioni necessarie a creare la relazione ‘**`Clienti`**’, con il contenuto che si vede nella figura 79.13, tenendo conto che:

1. l’attributo ‘**`Cliente`**’ è di tipo ‘**`INTEGER`**’, non ammette il valore nullo e costituisce la chiave primaria;
2. l’attributo ‘**`RagioneSociale`**’ è di tipo ‘**`VARCHAR`**’ a 120 caratteri e non ammette il valore nullo;
3. l’attributo ‘**`Paese`**’ è di tipo ‘**`CHAR`**’ a 30 caratteri e il suo valore predefinito è ‘**`ITALIA`**’;
4. l’attributo ‘**`Indirizzo`**’ è di tipo ‘**`VARCHAR`**’ a 120 caratteri e non ammette il valore nullo;
5. l’attributo ‘**`CAP`**’ è di tipo ‘**`CHAR`**’ a 10 caratteri;
6. l’attributo ‘**`Citta`**’ è di tipo ‘**`VARCHAR`**’ a 120 caratteri e non ammette il valore nullo;
7. l’attributo ‘**`Prov`**’ è di tipo ‘**`CHAR`**’ a 2 caratteri e il suo valore predefinito è costituito da una stringa di dimensione nulla;
8. l’attributo ‘**`Telefono`**’ è di tipo ‘**`CHAR`**’ a 20 caratteri e il suo valore predefinito è costituito da una stringa di dimensione nulla;
9. l’attributo ‘**`Fax`**’ è di tipo ‘**`CHAR`**’ a 20 caratteri e il suo valore predefinito è costituito da una stringa di dimensione nulla;
10. l’attributo ‘**`CFPI`**’ (codice fiscale o partita IVA) è di tipo ‘**`CHAR`**’ a 30 caratteri e il suo valore predefinito è costituito da una stringa di dimensione nulla.

Figura 79.13. Relazione **`Clienti (Cliente, RagioneSociale, Paese, Indirizzo, CAP, Citta, Prov, Telefono, Fax, CFPI)`**.

Cliente	RagioneSociale	Paese	Indirizzo	CAP	Citta	Prov	Telefono	Fax	CFPI
1	Mevio Mevi	ITALIA	via Mare, 11	31050	Morgano	TV	0422,444444	0422,555555	45678901234
2	Filano Filani	ITALIA	via Farfalle, 22	31032	Feltre	BL	0439,555555	0439,666666	56789012345
3	Martino Martini	ITALIA	via Marte, 33	31010	Mareno di Piave	TV	0438,666666	0438,777777	67890123456

Figura 79.14. Scheletro del file ‘`creazione-clienti.sql`’, da completare.

```
-- Creazione della relazione "Clienti"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: creazione-clienti.sql

CREATE TABLE Clienti …

INSERT INTO Clienti …
…
```

Una volta completato e salvato il file ‘`creazione-clienti.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < creazione-clienti.sql [Invio]
```

Se si ottengono degli errori, si deve eliminare la relazione ‘**`Clienti`**’ dalla base di dati contenuta nel file ‘`mag.db`’, utilizzando il programma ‘**`sqlite3`**’ in modo interattivo, quindi, dopo le correzioni, si deve riprovare.

Una volta eseguita l’operazione con successo, si stampi il file ‘`creazione-clienti.sql`’ e lo si consegni per la correzione all’insegnante.

Nella valutazione viene controllata la correttezza del contenuto del file e la coerenza estetica nella scrittura delle istruzioni SQL.

79.1.7 Verifica sulla creazione e popolazione della relazione «Movimenti»

«

Prima di svolgere questa verifica, è necessario avere svolto quelle precedenti, delle quali valgono anche gli stessi requisiti.

Si crei il file '`creazione-movimenti.sql`', inserendo le istruzioni necessarie a creare la relazione '**`Movimenti`**', con il contenuto che si vede nella figura 79.15, tenendo conto che:

1. l'attributo '**`Movimento`**' è di tipo '**`INTEGER`**', non ammette il valore nullo e costituisce la chiave primaria;
2. l'attributo '**`Articolo`**' è di tipo '**`INTEGER`**' e non ammette il valore nullo;
3. l'attributo '**`Causale`**' è di tipo '**`INTEGER`**' e non ammette il valore nullo;
4. l'attributo '**`Data`**' è di tipo '**`DATE`**' e non ammette il valore nullo;
5. l'attributo '**`Cliente`**' è di tipo '**`INTEGER`**';
6. l'attributo '**`Fornitore`**' è di tipo '**`INTEGER`**';
7. l'attributo '**`Quantita`**' è di tipo '**`NUMERIC`**' a 15 cifre, di cui cinque sono usate per i decimali, e non ammette il valore nullo;
8. l'attributo '**`Valore`**' è di tipo '**`NUMERIC`**' a 14 cifre, di cui due sono usate per i decimali, e non ammette il valore nullo.

Figura 79.15. Relazione **`Movimenti (Movimento, Articolo, Causale, Data, Cliente, Fornitore, Quantita, Valore)`**.

Movimento	Articolo	Causale	Data	Cliente	Fornitore	Quantita	Valore
1	2	1	2012-01-15	NULL	3	10000	100,00
2	2	2	2012-01-16	2	NULL	1000	10,00
3	102	1	2012-01-17	NULL	2	1000	200,00
4	102	2	2012-01-18	1	NULL	100	20,00
5	401	1	2012-01-19	NULL	1	1000	200,00
6	401	2	2012-01-20	3	NULL	200	40,00
7	401	4	2012-01-20	NULL	1	100	20,00
8	102	4	2012-01-20	NULL	2	100	20,00
9	601	1	2012-01-21	NULL	3	2000	1000,00
10	601	2	2012-01-25	1	NULL	1000	500,00

Figura 79.16. Scheletro del file ‘`creazione-movimenti.sql`’, da completare.

```
-- Creazione della relazione "Movimenti"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: creazione-movimenti.sql

CREATE TABLE Movimenti …

INSERT INTO Movimenti …
…
```

Si ricorda che i valori numerici vanno indicati come sono, con l’accortezza di usare il punto per la separazione dei decimali; le date, come le stringhe (i valori testuali) vanno delimitate con apici singoli. In questo caso, quando viene a mancare il valore per il cliente o il fornitore, si inserisce il valore «nullo», che si scrive con la parola chiave ‘**`NULL`**’, come appare nella figura 79.15, senza usare apici.

Una volta completato e salvato il file ‘`creazione-movimenti.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < creazione-movimenti.sql[Invio]
```

Se si ottengono degli errori, si deve eliminare la relazione '**Movimenti**' dalla base di dati contenuta nel file 'mag.db', utilizzando il programma '**sqlite3**' in modo interattivo, quindi, dopo le correzioni, si deve riprovare.

Una volta eseguita l'operazione con successo, si stampi il file 'creazione-movimenti.sql' e lo si consegni per la correzione all'insegnante.

Nella valutazione viene controllata la correttezza del contenuto del file e la coerenza estetica nella scrittura delle istruzioni SQL.

79.1.8 Conclusione

«

Prima di passare alla sezione successiva, si deve realizzare un file contenente le istruzioni con cui creare e popolare le relazioni descritte in questa. In pratica, si tratta di copiare il contenuto dei file 'creazione-articoli.sql', 'creazione-causali.sql', 'creazione-fornitori.sql', 'creazione-clienti.sql' e 'creazione-movimenti.sql', in un file completo, denominato 'magazzino.sql'.

Una volta realizzato il file 'magazzino.sql', si deve cancellare il file 'mag.db' e ricreare a partire dalle istruzioni contenute nel file 'magazzino.sql':

```
$ sqlite3 mag.db < magazzino.sql[Invio]
```

Se vengono segnalati degli errori, occorre correggere il file 'magazzino.sql', cancellare nuovamente il file 'mag.db', quindi si deve ripetere l'operazione. La base di dati contenuta nel file 'mag.db', viene usata ancora e non si può proseguire se non si riesce a ricrearla correttamente.

All'inizio della sezione è stato creato il file '`prova-creazione-articoli.sql`' che a questo punto non serve più e va eliminato.

79.2 Interrogazione semplice di una relazione

Attraverso l'istruzione '**`SELECT`**' è possibile estrarre il contenuto di una o più relazioni simultaneamente. In questa sezione si mostrano alcune situazioni riferite a una sola relazione.

79.2.1 Interrogazione completa

Si ottiene l'elenco completo di una relazione utilizzando l'istruzione seguente:

```
SELECT * FROM nome_relazione
```

Si eseguano i passaggi seguenti, per ottenere la visualizzazione del contenuto complessivo della relazione '**`Articoli`**' e della relazione '**`Causali`**', così come dovrebbero essere contenute nella base di dati del file '**`mag.db`**':

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM Articoli;[Invio]
```

```
Articolo    Descrizione                  UM          Listino     ScortaMin
----------  ---------------------------  ----------  ----------  ----------
1           Dischetti da 9 cm 1440 Kibyte  pz          0.2         500
2           Dischetti da 9 cm 1440 Kibyte  pz          0.25        500
101         CD-R 16x                     pz          0.5         500
102         CD-R 52x                     pz          1           500
201         CD-RW 4x                     pz          1           200
202         CD-RW 8x                     pz          1.5         200
301         DVD-R 8x                     pz          1           200
302         DVD-R 16x                    pz          2           200
401         DVD+R 8x                     pz          1           200
402         DVD+R 16x                    pz          2           200
501         DVD-RW 8x                    pz          2           200
601         DVD+RW 8x                    pz          2           200
```

sqlite> **SELECT * FROM Causali;** [*Invio*]

```
Causale     Descrizione          Variazione
----------  -------------------  ----------
1           Carico per acquisto  1
2           Scarico per vendita  -1
3           Reso da cliente      1
4           Reso a fornitore     -1
5           Rettifica aumento a  1
6           Rettifica aumento v  -1
7           Rettifica diminuzio  1
8           Rettifica diminuzio  -1
9           Carico da produzion  1
10          Scarico a produzion  -1
11          Carico da altro mag  1
12          Scarico ad altro ma  -1
13          Saldo iniziale       1
```

sqlite> **.quit** [*Invio*]

Si osservi che i comandi '**.headers on**' e '**.mode column**' servono a ottenere un elenco incolonnato con le intestazioni, altrimenti, il

risultato sarebbe poco gradevole esteticamente.

79.2.2 Interrogazione con selezione di alcuni attributi

Si ottiene l'elenco di tutte le tuple di una relazione, limitatamente a un certo gruppo di attributi, mettendo, al posto dell'asterisco, i nomi degli attributi desiderati:

```
SELECT attributo [, attributo] ... FROM nome_relazione
```

Si eseguano i passaggi seguenti, per ottenere la visualizzazione del contenuto di tutte le tuple della relazione '**Articoli**', limitatamente agli attributi '**Articolo**', '**Descrizione**' e '**Listino**', così come dovrebbero essere contenute nella base di dati del file '**mag.db**':

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT Articolo, Descrizione, Listino[Invio]

      >         FROM Articoli; [Invio]
```

```
Articolo    Descrizione                    Listino
----------  -----------------------------  ----------
1           Dischetti da 9 cm 1440 Kibyte  0.2
2           Dischetti da 9 cm 1440 Kibyte  0.25
101         CD-R 16x                       0.5
102         CD-R 52x                       1
201         CD-RW 4x                       1
202         CD-RW 8x                       1.5
301         DVD-R 8x                       1
302         DVD-R 16x                      2
401         DVD+R 8x                       1
402         DVD+R 16x                      2
501         DVD-RW 8x                      2
601         DVD+RW 8x                      2
```

Intuitivamente, si comprende che si può anche cambiare l'ordine di visualizzazione degli attributi:

```
sqlite> SELECT Descrizione, Articolo, Listino[Invio]

      >         FROM Articoli;[Invio]
```

```
Descrizione                    Articolo    Listino
-----------------------------  ----------  ----------
Dischetti da 9 cm 1440 Kibyte  1           0.2
Dischetti da 9 cm 1440 Kibyte  2           0.25
CD-R 16x                       101         0.5
CD-R 52x                       102         1
CD-RW 4x                       201         1
CD-RW 8x                       202         1.5
DVD-R 8x                       301         1
DVD-R 16x                      302         2
DVD+R 8x                       401         1
DVD+R 16x                      402         2
DVD-RW 8x                      501         2
DVD+RW 8x                      601         2
```

Si conclude il funzionamento interattivo di '**sqlite3**' con il comando '**.quit**':

```
sqlite> .quit[Invio]
```

79.2.3 Stampa del contenuto di una relazione

Per ottenere la stampa del contenuto di una o di più relazioni, conviene scrivere le istruzioni necessarie in un file di testo, come già fatto in precedenza. Si proceda con la creazione del file '`prova-stampa-artico-caus.sql`', con il contenuto seguente, che ricalca quanto già mostrato nelle sezioni precedenti:

```
-- Stampa del contenuto delle relazioni "Articoli" e "Causali"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: prova-stampa-artico-caus.sql

.headers on
.mode column
```

```
SELECT * FROM Articoli;
SELECT * FROM Causali;
```

Per verificare il funzionamento delle istruzioni contenute nel file ‘`stampa-artico-caus.sql`’, si può utilizzare il comando seguente, che interviene nella base di dati contenuta nel file ‘`mag.db`’, limitandosi a visualizzare il risultato:

```
$ sqlite3 mag.db < prova-stampa-artico-caus.sql [Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Articolo    Descrizione                  UM          Listino     ScortaMin
----------  ---------------------------  ----------  ----------  ----------
1           Dischetti da 9 cm 1440 Kibyte  pz          0.2         500
2           Dischetti da 9 cm 1440 Kibyte  pz          0.25        500
101         CD-R 16x                     pz          0.5         500
102         CD-R 52x                     pz          1           500
201         CD-RW 4x                     pz          1           200
202         CD-RW 8x                     pz          1.5         200
301         DVD-R 8x                     pz          1           200
302         DVD-R 16x                    pz          2           200
401         DVD+R 8x                     pz          1           200
402         DVD+R 16x                    pz          2           200
501         DVD-RW 8x                    pz          2           200
601         DVD+RW 8x                    pz          2           200
Causale     Descrizione          Variazione
----------  -------------------  ----------
1           Carico per acquisto  1
2           Scarico per vendita  -1
3           Reso da cliente      1
4           Reso a fornitore     -1
5           Rettifica aumento a  1
6           Rettifica aumento v  -1
7           Rettifica diminuzio  1
8           Rettifica diminuzio  -1
9           Carico da produzion  1
10          Scarico a produzion  -1
11          Carico da altro mag  1
```

```
12          Scarico ad altro ma  -1
13          Saldo iniziale       1
```

Per ottenere il risultato stampato su carta, basta modificare leggermente il comando:

```
$ sqlite3 mag.db < prova-stampa-artico-caus.sql ↩
↪| lpr[Invio]
```

79.2.4 Verifica sull'interrogazione della relazione «Articoli»

Si prepari il file '`interr-artico-01.sql`', seguendo lo scheletro seguente, tenendo conto che si vuole ottenere l'elenco di tutte le tuple della relazione '**`Articoli`**', ordinando gli attributi in questo modo: '**`Descrizione`**', '**`Articolo`**', '**`UM`**', '**`ScortaMin`**' e '**`Listino`**'.

Figura 79.25. Scheletro del file '`interr-artico-01.sql`', da completare.

```
-- Interrogazione della relazione "Articoli"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-artico-01.sql

.headers on
.mode column

SELECT …
```

Una volta completato e salvato il file '`interr-artico-01.sql`', se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < interr-artico-01.sql[Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
Descrizione                  Articolo    UM          ScortaMin   Listino
---------------------------  ----------  ----------  ----------  ----------
Dischetti da 9 cm 1440 Kibyte  1          pz          500         0.2
Dischetti da 9 cm 1440 Kibyte  2          pz          500         0.25
CD-R 16x                     101         pz          500         0.5
CD-R 52x                     102         pz          500         1
CD-RW 4x                     201         pz          200         1
CD-RW 8x                     202         pz          200         1.5
DVD-R 8x                     301         pz          200         1
DVD-R 16x                    302         pz          200         2
DVD+R 8x                     401         pz          200         1
DVD+R 16x                    402         pz          200         2
DVD-RW 8x                    501         pz          200         2
DVD+RW 8x                    601         pz          200         2
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-artico-01.sql | lpr[Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo, assieme alla stampa del file ‘`interr-artico-01.sql`’.

79.2.5 Verifica sull’interrogazione delle relazioni «Fornitori» e «Clienti»

«

Si prepari il file ‘`interr-forn-clie-01.sql`’, seguendo lo scheletro seguente, tenendo conto che si vuole ottenere l’elenco di tutte le tuple delle relazioni ‘**`Fornitori`**’ e ‘**`Clienti`**’, limitatamente agli attributi: ‘**`Fornitore`**’ (nel caso della relazione ‘**`Fornitori`**’) o ‘**`Cliente`**’ (nel caso della relazione ‘**`Clienti`**’), ‘**`RagioneSociale`**’, ‘**`Telefono`**’ e ‘**`Fax`**’.

Figura 79.27. Scheletro del file ‘`interr-forn-clie-01.sql`’, da completare.

```
-- Interrogazione delle relazioni "Fornitori" e "Clienti"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-forn-clie-01.sql

.headers on
.mode column

SELECT …
SELECT …
```

Una volta completato e salvato il file ‘`interr-forn-clie-01.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < interr-forn-clie-01.sql[Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
Fornitore   RagioneSociale  Telefono     Fax
----------  --------------  -----------  -----------
1           Tizio Tizi      0422,111111  0422,222222
2           Caio Cai        0423,222222  0423,333333
3           Sempronio Semp  0422,333333  0422,444444
Cliente     RagioneSociale  Telefono     Fax
----------  --------------  -----------  -----------
1           Mevio Mevi      0422,444444  0422,555555
2           Filano Filani   0439,555555  0439,666666
3           Martino Martin  0438,666666  0438,777777
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-forn-clie-01.sql ↩
↪         | lpr[Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo, assieme alla stampa del file ‘`interr-forn-clie-01.sql`’.

79.2.6 Conclusione

«

Il file ‘`prova-stampa-artico-caus.sql`’, non serve più nelle sezioni successive, pertanto va eliminato.

79.3 Interrogazione ordinata di una relazione

«

Attraverso l’istruzione ‘**`SELECT`**’, aggiungendo l’opzione ‘**`ORDERED BY`**’, è possibile specificare gli attributi secondo i quali ordinare il risultato. In mancanza dell’indicazione di questa opzione, l’elenco delle tuple si ottiene secondo un ordine «casuale», che solitamente coincide con la sequenza di inserimento.

79.3.1 Interrogazione ordinata

«

A titolo di esempio, si vuole ottenere l’elenco delle tuple della relazione ‘**`Articoli`**’, in ordine di descrizione. Si può utilizzare il programma ‘**`sqlite3`**’ in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM Articoli ↩
↪                 ORDER BY Descrizione;[Invio]
```

```
Articolo    Descrizione  UM          Listino     ScortaMin
----------  -----------  ----------  ----------  ----------
101         CD-R 16x     pz          0.5         500
102         CD-R 52x     pz          1           500
201         CD-RW 4x     pz          1           200
202         CD-RW 8x     pz          1.5         200
402         DVD+R 16x    pz          2           200
401         DVD+R 8x     pz          1           200
601         DVD+RW 8x    pz          2           200
302         DVD-R 16x    pz          2           200
301         DVD-R 8x     pz          1           200
501         DVD-RW 8x    pz          2           200
1           Dischetti d  pz          0.2         500
2           Dischetti d  pz          0.25        500
```

Con la relazione '**Articoli**', potrebbe essere interessante un ordinamento per listino, ma in questo caso si aggiunge anche la descrizione, quando il prezzo di listino risulta uguale:

```
sqlite> SELECT * FROM Articoli ORDER BY Listino, Descrizione;
```
[*Invio*]

```
Articolo    Descrizione                    UM          Listino     ScortaMin
----------  -----------------------------  ----------  ----------  ----------
1           Dischetti da 9 cm 1440 Kibyte  pz          0.2         500
2           Dischetti da 9 cm 1440 Kibyte  pz          0.25        500
101         CD-R 16x                       pz          0.5         500
102         CD-R 52x                       pz          1           500
201         CD-RW 4x                       pz          1           200
401         DVD+R 8x                       pz          1           200
301         DVD-R 8x                       pz          1           200
202         CD-RW 8x                       pz          1.5         200
402         DVD+R 16x                      pz          2           200
601         DVD+RW 8x                      pz          2           200
302         DVD-R 16x                      pz          2           200
501         DVD-RW 8x                      pz          2           200
```

```
sqlite> .quit
```
[*Invio*]

Si osservi che l'ordinamento dipende dal tipo di informazione che l'attributo relativo può contenere. Per esempio, nel caso della relazione '**Articoli**', il riordino per descrizione avviene in modo lessicografico, mentre il riordino per listino avviene in base al valore numerico.

79.3.2 Verifica sull'interrogazione ordinata della relazione «Articoli»

«

Si prepari il file '`interr-artico-02.sql`', seguendo lo scheletro seguente, tenendo conto che si vuole ottenere l'elenco di tutte le tuple della relazione '**Articoli**', ordinate in base al livello di scorta minima e di descrizione; inoltre, si vogliono ottenere solo alcuni attributi, secondo la sequenza: '**ScortaMin**', '**Descrizione**', '**Articolo**'.

Figura 79.32. Scheletro del file '`interr-artico-02.sql`', da completare.

```
-- Interrogazione della relazione "Articoli"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-artico-02.sql

.headers on
.mode column

SELECT …
```

Una volta completato e salvato il file '`interr-artico-02.sql`', se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < interr-artico-02.sql[Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
ScortaMin   Descrizione  Articolo
----------  -----------  ----------
200         CD-RW 4x     201
200         CD-RW 8x     202
200         DVD+R 16x    402
200         DVD+R 8x     401
200         DVD+RW 8x    601
200         DVD-R 16x    302
200         DVD-R 8x     301
200         DVD-RW 8x    501
500         CD-R 16x     101
500         CD-R 52x     102
500         Dischetti d  1
500         Dischetti d  2
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-artico-02.sql | lpr[Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo, assieme alla stampa del file ‘`interr-artico-02.sql`’.

79.3.3 Verifica sull’interrogazione ordinata della relazione «Clienti»

Si prepari il file ‘`interr-clie-01.sql`’, seguendo lo scheletro seguente, tenendo conto che si vuole ottenere l’elenco di tutte le tuple della relazione ‘**`Clienti`**’, ordinate in base alla denominazione della ragione sociale, limitatamente agli attributi ‘**`Cliente`**’ e ‘**`RagioneSociale`**’.

Figura 79.34. Scheletro del file ‘`interr-clie-01.sql`’, da completare.

```
-- Interrogazione della relazione "Clienti"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-clie-01.sql

.headers on
.mode column

SELECT …
```

Una volta completato e salvato il file ‘`interr-clie-01.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < interr-clie-01.sql[Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
Cliente     RagioneSociale
----------  --------------
2           Filano Filani
3           Martino Martin
1           Mevio Mevi
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-clie-01.sql | lpr[Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo, assieme alla stampa del file ‘`interr-clie-01.sql`’.

79.3.4 Verifica sull'interrogazione ordinata della relazione «Causali»

Si prepari il file '`interr-caus-01.sql`', seguendo lo scheletro seguente, tenendo conto che si vuole ottenere l'elenco di tutte le tuple della relazione '**`Causali`**', ordinate in base al fatto che si tratti di movimenti in diminuzione o in aumento (l'attributo '**`Variazione`**').

Figura 79.36. Scheletro del file '`interr-caus-01.sql`', da completare.

```
-- Interrogazione della relazione "Causali"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-caus-01.sql

.headers on
.mode column

SELECT …
```

Una volta completato e salvato il file '`interr-caus-01.sql`', se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < interr-clie-01.sql[Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
Causale     Descrizione                Variazione
----------  -------------------------  ----------
12          Scarico ad altro magazzino  -1
10          Scarico a produzione        -1
8           Rettifica diminuzione acqu  -1
6           Rettifica aumento vendite   -1
4           Reso a fornitore            -1
2           Scarico per vendita         -1
```

```
13          Saldo iniziale              1
11          Carico da altro magazzino   1
9           Carico da produzione        1
7           Rettifica diminuzione vend  1
5           Rettifica aumento acquisto  1
3           Reso da cliente             1
1           Carico per acquisto         1
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-caus-01.sql | lpr[Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo, assieme alla stampa del file ‘`interr-caus-01.sql`’.

79.4 Interrogazione selettiva di una relazione

«

Attraverso l’istruzione ‘**`SELECT`**’, aggiungendo l’opzione ‘**`WHERE`**’, è possibile specificare una condizione per la selezione delle tuple desiderate. In mancanza dell’indicazione di questa opzione, l’elenco delle tuple è sempre completo. La parola chiave ‘**`WHERE`**’ precede un’espressione logica, che viene valutata per ogni tupla: se l’espressione risulta valida (*Vero*), allora la tupla viene presa in considerazione.

In queste lezioni non viene descritto in modo dettagliato come scrivere delle espressioni logiche; tuttavia, vengono raccolte qui delle tabelle riassuntive per la loro realizzazione. Possono essere usate in modo intuitivo, ma nelle verifiche non si richiede altro che utilizzare o modificare leggermente degli esempi già mostrati.

Tabella 79.38. Operatori di confronto.

Operatore e operandi	Descrizione
op1 `=` ***op2***	*Vero* se gli operandi si equivalgono.
op1 `<>` ***op2***	*Vero* se gli operandi sono differenti.
op1 `<` ***op2***	*Vero* se il primo operando è minore del secondo.
op1 `>` ***op2***	*Vero* se il primo operando è maggiore del secondo.
op1 `<=` ***op2***	*Vero* se il primo operando è minore o uguale al secondo.
op1 `>=` ***op2***	*Vero* se il primo operando è maggiore o uguale al secondo.

Tabella 79.39. Operatori logici.

Operatore e operandi	Descrizione
`NOT` ***op***	Inverte il risultato logico dell'operando.
op1 `AND` ***op2***	*Vero* se entrambi gli operandi restituiscono il valore *Vero*.
op1 `OR` ***op2***	*Vero* se almeno uno degli operandi restituisce il valore *Vero*.

Tabella 79.40. Espressioni sulle stringhe di caratteri.

Espressioni e modelli	Descrizione
stringa `LIKE` ***modello***	Restituisce *Vero* se il modello corrisponde alla stringa. Si osservi che SQLite non accetta la forma '**`IS LIKE`**'.

Espressioni e modelli	Descrizione
stringa `NOT LIKE` ***modello***	Restituisce *Vero* se il modello non corrisponde alla stringa. Si osservi che SQLite non accetta la forma '**`IS NOT LIKE`**'.
_	Rappresenta un carattere qualsiasi.
%	Rappresenta una sequenza indeterminata di caratteri.

Tabella 79.41. Espressioni di verifica dei valori indeterminati.

Operatori	Descrizione
espressione `IS NULL`	Restituisce *Vero* se l'espressione genera un risultato indeterminato.
espressione `IS NOT NULL`	Restituisce *Vero* se l'espressione non genera un risultato indeterminato.

Tabella 79.42. Espressioni per la verifica dell'appartenenza di un valore a un intervallo o a un elenco.

Operatori e operandi	Descrizione
op1 `IN (`***elenco***`)`	*Vero* se il primo operando è contenuto nell'elenco.
op1 `NOT IN (`***elenco***`)`	*Vero* se il primo operando non è contenuto nell'elenco.
op1 `BETWEEN` ***op2*** `AND` ***op3***	*Vero* se il primo operando è compreso tra il secondo e il terzo.
op1 `NOT BETWEEN` ***op2*** `AND` ***op3***	*Vero* se il primo operando non è compreso nell'intervallo.

79.4.1 Interrogazione selettiva

A titolo di esempio, si vuole ottenere l'elenco delle tuple della relazione '**Articoli**', selezionando solo quelle che riportano un prezzo di listino maggiore o uguale a 1,00 €. Si può utilizzare il programma '**sqlite3**' in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM Articoli WHERE Listino >= 1;[Invio]

  Articolo    Descrizione  UM          Listino     ScortaMin
  ----------  -----------  ----------  ----------  ----------
  102         CD-R 52x     pz          1           500
  201         CD-RW 4x     pz          1           200
  202         CD-RW 8x     pz          1.5         200
  301         DVD-R 8x     pz          1           200
  302         DVD-R 16x    pz          2           200
  401         DVD+R 8x     pz          1           200
  402         DVD+R 16x    pz          2           200
  501         DVD-RW 8x    pz          2           200
  601         DVD+RW 8x    pz          2           200
```

La condizione di selezione potrebbe essere più articolata; per esempio si potrebbe decidere di selezionare gli articoli che hanno un prezzo di listino maggiore o uguale a 1,00 € e che hanno una descrizione che inizia con «DVD»:

```
sqlite> SELECT * FROM Articoli[Invio]
```

```
...>          WHERE Listino >= 1 [Invio]

...>            AND Descrizione LIKE 'DVD%'; [Invio]

Articolo    Descrizione  UM          Listino     ScortaMin
----------  -----------  ----------  ----------  ----------
301         DVD-R 8x     pz          1           200
302         DVD-R 16x    pz          2           200
401         DVD+R 8x     pz          1           200
402         DVD+R 16x    pz          2           200
501         DVD-RW 8x    pz          2           200
601         DVD+RW 8x    pz          2           200
```

Come sempre, si conclude il funzionamento interattivo di '**sqlite3**' con il comando '**.quit**':

```
sqlite> .quit [Invio]
```

79.4.2 Verifica sull'interrogazione selettiva della relazione «Articoli»

«

Si prepari il file '`interr-artico-03.sql`', seguendo lo scheletro seguente, tenendo conto che si vuole ottenere l'elenco delle tuple della relazione '**Articoli**', corrispondenti a dei «DVD», che abbiano un prezzo minore o uguale a 1,00 € (l'operatore da usare per rappresentare il confronto «minore o uguale» è '**<=**').

Figura 79.46. Scheletro del file ‘`interr-artico-03.sql`’, da completare.

```
-- Interrogazione della relazione "Articoli"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-artico-03.sql

.headers on
.mode column

SELECT …
       FROM …
       WHERE …
```

Una volta completato e salvato il file ‘`interr-artico-03.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < interr-artico-03.sql[Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
Articolo    Descrizione  UM          Listino     ScortaMin
----------  -----------  ----------  ----------  ----------
301         DVD-R 8x     pz          1           200
401         DVD+R 8x     pz          1           200
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-artico-03.sql | lpr[Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo, assieme alla stampa del file ‘`interr-artico-03.sql`’.

79.4.3 Verifica sull'interrogazione selettiva e ordinata della relazione «Articoli»

«

Si prepari il file '`interr-artico-04.sql`', seguendo lo scheletro seguente, tenendo conto che si vuole ottenere l'elenco ordinato per livello di scorta minima delle tuple della relazione '**`Articoli`**', che corrispondono a dei «CD».

Figura 79.48. Scheletro del file '`interr-artico-04.sql`', da completare.

```
-- Interrogazione della relazione "Articoli"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-artico-04.sql

.headers on
.mode column

SELECT …
       FROM …
       WHERE …
       ORDER BY …
```

Una volta completato e salvato il file '`interr-artico-04.sql`', se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < interr-artico-04.sql [Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
Articolo    Descrizione  UM          Listino     ScortaMin
----------  -----------  ----------  ----------  ----------
202         CD-RW 8x     pz          1.5         200
201         CD-RW 4x     pz          1           200
102         CD-R 52x     pz          1           500
```

```
101         CD-R 16x     pz          0.5         500
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-artico-04.sql | lpr[Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo, assieme alla stampa del file ‘`interr-artico-04.sql`’.

79.4.4 Verifica sull’interrogazione selettiva della relazione «Causali»

Si prepari il file ‘`interr-caus-02.sql`’, seguendo lo scheletro seguente, tenendo conto che si vuole ottenere l’elenco delle tuple della relazione ‘**`Causali`**’ che comportano un aumento (contabile) della quantità di un articolo in magazzino. Le causali che rappresentano un aumento della quantità sono quelle che, nell’attributo ‘**`Variazione`**’ hanno il valore 1 (ovvero +1); pertanto, per selezionare le tuple in questione, è sufficiente verificare che questo valore sia esattamente pari a uno (utilizzando l’operatore ‘**=**’).

Figura 79.50. Scheletro del file ‘`interr-caus-02.sql`’, da completare.

```
-- Interrogazione della relazione "Causali"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-caus-02.sql

.headers on
.mode column

SELECT …
       FROM …
       WHERE …
```

Una volta completato e salvato il file ‘`interr-caus-02.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < interr-caus-02.sql [Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
Causale     Descrizione          Variazione
----------  -------------------  ----------
1           Carico per acquisto  1
3           Reso da cliente      1
5           Rettifica aumento a  1
7           Rettifica diminuzio  1
9           Carico da produzion  1
11          Carico da altro mag  1
13          Saldo iniziale       1
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-caus-02.sql | lpr [Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo,

assieme alla stampa del file ‘`interr-caus-02.sql`’.

79.5 Interrogazioni simultanee di più relazioni

Quando si realizzano delle relazioni, spesso si considerano dei collegamenti tra queste, per evitare di ripetere le stesse informazioni in relazioni differenti. La relazione ‘**`Movimenti`**’, creata all’inizio di queste lezioni, contiene diversi attributi che, in pratica, fanno riferimento a tuple di altre relazioni.

Figura 79.52. La relazione ‘**`Movimenti`**’, già apparsa nella figura 79.15.

Movimento	Articolo	Causale	Data	Cliente	Fornitore	Quantita	Valore
1	2	1	2012-01-15	NULL	3	10000	100,00
2	2	2	2012-01-16	2	NULL	1000	10,00
3	102	1	2012-01-17	NULL	2	1000	200,00
4	102	2	2012-01-18	1	NULL	100	20,00
5	401	1	2012-01-19	NULL	1	1000	200,00
6	401	2	2012-01-20	3	NULL	200	40,00
7	401	4	2012-01-20	NULL	1	100	20,00
8	102	4	2012-01-20	NULL	2	100	20,00
9	601	1	2012-01-21	NULL	3	2000	1000,00
10	601	2	2012-01-25	1	NULL	1000	500,00

Intuitivamente si comprende che i dati usati per creare il collegamento con un’altra relazione, devono essere sufficienti a individuare le tuple in modo univoco. Quindi, sulla base di questa univocità, si possono collegare effettivamente i dati attraverso delle interrogazioni che coinvolgono tutte le relazioni interessate, per generare un listato con le informazioni desiderate.

79.5.1 Interrogazione simultanea delle relazioni «Movimenti» e «Articoli»

Con l’ausilio di un programma per la scrittura e modifica di file di testo puro, si crei il file ‘`prova-interr-movi-arti.sql`’, conte-

nente il testo seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Interrogazione delle relazioni "Movimenti" e "Articoli"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: prova-interr-movi-arti.sql

.headers on
.mode column

SELECT Movimenti.Data, Articoli.Descrizione,
       Movimenti.Causale, Movimenti.Quantita
       FROM Movimenti, Articoli
       WHERE Movimenti.Articolo = Articoli.Articolo;
```

Come si può vedere, per evitare ambiguità, i nomi degli attributi sono preceduti dal nome della relazione a cui appartengono, separati da un punto.

Si controlli di avere scritto il file '`prova-interr-movi-arti.sql`' in modo corretto, rispettando anche la punteggiatura; si controlli di avere salvato il file con il nome previsto, quindi si proceda con il comando seguente:

```
$ sqlite3 mag.db < prova-interr-movi-arti.sql[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Data        Descrizione                              Causale     Quantita
----------  ---------------------------------------  ----------  ----------
2012-01-15  Dischetti da 9 cm 1440 Kibyte colorati   1           10000
2012-01-16  Dischetti da 9 cm 1440 Kibyte colorati   2           1000
2012-01-17  CD-R 52x                                 1           1000
2012-01-18  CD-R 52x                                 2           100
2012-01-19  DVD+R 8x                                 1           1000
2012-01-20  DVD+R 8x                                 2           200
```

```
2012-01-20  DVD+R 8x                    4           100
2012-01-20  CD-R 52x                    4           100
2012-01-21  DVD+RW 8x                   1           2000
2012-01-25  DVD+RW 8x                   2           1000
```

79.5.2 Interrogazione simultanea delle relazioni «Movimenti», «Articoli» e «Causali»

Si riprenda il file ‘`prova-interr-movi-arti.sql`’ e lo si modifichi in modo da avere il contenuto seguente:

```
-- Interrogazione delle relazioni "Movimenti", "Articoli"
--   e "Causali"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: prova-interr-movi-arti.sql

.headers on
.mode column

SELECT Movimenti.Data, Articoli.Descrizione,
       Causali.Descrizione
       FROM Movimenti, Articoli, Causali
       WHERE Movimenti.Articolo = Articoli.Articolo
         AND Movimenti.Causale = Causali.Causale;
```

Si controlli di avere modificato il file ‘`prova-interr-movi-arti.sql`’ in modo corretto, rispettando anche la punteggiatura; si controlli di avere salvato il file con il nome previsto, quindi si proceda con il comando seguente:

```
$ sqlite3 mag.db < prova-interr-movi-arti.sql [Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Data        Descrizione                                 Descrizione
----------  ------------------------------------------  ------------------
2012-01-15  Dischetti da 9 cm 1440 Kibyte colorati      Carico per acquisto
2012-01-16  Dischetti da 9 cm 1440 Kibyte colorati      Scarico per vendita
2012-01-17  CD-R 52x                                    Carico per acquisto
2012-01-18  CD-R 52x                                    Scarico per vendita
2012-01-19  DVD+R 8x                                    Carico per acquisto
2012-01-20  DVD+R 8x                                    Scarico per vendita
2012-01-20  DVD+R 8x                                    Reso a fornitore
2012-01-20  CD-R 52x                                    Reso a fornitore
2012-01-21  DVD+RW 8x                                   Carico per acquisto
2012-01-25  DVD+RW 8x                                   Scarico per vendita
```

79.5.3 Verifica sull'interrogazione simultanea delle relazioni «Movimenti» e «Causali»

«

Si prepari il file '`interr-movi-caus-01.sql`', seguendo lo scheletro seguente, tenendo conto che si vuole ottenere l'elenco di tutte le tuple per le quali si possa stabilire un abbinamento in base al codice della causale. Precisamente, si vuole ottenere l'attributo '**`Articolo`**' dalla relazione '**`Movimenti`**'; l'attributo '**`Descrizione`**' dalla relazione '**`Causali`**'; l'attributo '**`Data`**' dalla relazione '**`Movimenti`**'.

Figura 79.57. Scheletro del file ‘`interr-movi-caus-01.sql`’, da completare.

```
-- Interrogazione delle relazioni "Movimenti" e "Causali"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-movi-caus-01.sql

.headers on
.mode column

SELECT …
       FROM …
       WHERE …
```

Una volta completato e salvato il file ‘`interr-movi-caus-01.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < interr-movi-caus-01.sql[Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
Articolo    Descrizione          Data
----------  -------------------  ----------
2           Carico per acquisto  2012-01-15
2           Scarico per vendita  2012-01-16
102         Carico per acquisto  2012-01-17
102         Scarico per vendita  2012-01-18
401         Carico per acquisto  2012-01-19
401         Scarico per vendita  2012-01-20
401         Reso a fornitore     2012-01-20
102         Reso a fornitore     2012-01-20
601         Carico per acquisto  2012-01-21
601         Scarico per vendita  2012-01-25
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-movi-caus-01.sql | lpr[Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo, assieme alla stampa del file ‘`interr-movi-caus-01.sql`’.

79.5.4 Verifica sull’interrogazione simultanea delle relazioni «Movimenti», «Causali» e «Clienti»

Si prepari il file ‘`interr-movi-caus-clienti-01.sql`’, seguendo lo scheletro seguente, tenendo conto che si vuole ottenere l’elenco di tutte le tuple per le quali si possa stabilire un abbinamento in base al codice della causale e in base al codice del cliente. Precisamente, si vuole ottenere l’attributo ‘**`Articolo`**’ dalla relazione ‘**`Movimenti`**’; l’attributo ‘**`Descrizione`**’ dalla relazione ‘**`Causali`**’; l’attributo ‘**`Data`**’ dalla relazione ‘**`Movimenti`**’; l’attributo ‘**`RagioneSociale`**’ dalla relazione ‘**`Clienti`**’.

Figura 79.59. Scheletro del file ‘`interr-movi-caus-clienti-01.sql`’, da completare.

```
-- Interrogazione delle relazioni "Movimenti", "Causali"
-- e "Clienti"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-movi-caus-clienti-01.sql

.headers on
.mode column

SELECT …
       FROM …
       WHERE …
```

Una volta completato e salvato il file ‘`interr-movi-caus-clienti-01.sql`’, se ne controlli il funzionamento con la

base di dati:

```
$ sqlite3 mag.db < interr-movi-caus-clienti-01.sql[Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
Articolo    Descrizione          Data        RagioneSociale
----------  -------------------  ----------  --------------
2           Scarico per vendita  2012-01-16  Filano Filani
102         Scarico per vendita  2012-01-18  Mevio Mevi
401         Scarico per vendita  2012-01-20  Martino Martin
601         Scarico per vendita  2012-01-25  Mevio Mevi
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-movi-caus-clienti-01.sql ↩
↪         | lpr[Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo, assieme alla stampa del file ‘`interr-movi-caus-clienti-01.sql`’.

79.5.5 Verifica sull’interrogazione ordinata e simultanea delle relazioni «Movimenti», «Causali» e «Clienti»

Si prepari il file ‘`interr-movi-caus-clienti-02.sql`’, che deve avere gli stessi requisiti della verifica precedente, facendo in modo, però, che il risultato dell’interrogazione avvenga in modo ordinato, in base alla ragione sociale dei clienti.

Figura 79.61. Scheletro del file ‘interr-movi-caus-clienti-02.sql’, da completare.

```
-- Interrogazione delle relazioni "Movimenti", "Causali"
-- e "Clienti"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-movi-caus-clienti-02.sql

.headers on
.mode column

SELECT …
       FROM …
       WHERE …
       ORDER BY …
```

Una volta completato e salvato il file ‘interr-movi-caus-clienti-02.sql’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < interr-movi-caus-clienti-02.sql[Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
Articolo    Descrizione          Data        RagioneSociale
----------  -------------------  ----------  --------------
2           Scarico per vendita  2012-01-16  Filano Filani
401         Scarico per vendita  2012-01-20  Martino Martin
601         Scarico per vendita  2012-01-25  Mevio Mevi
102         Scarico per vendita  2012-01-18  Mevio Mevi
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-movi-caus-clienti-02.sql ↩
↪         | lpr[Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo, assieme alla stampa del file ‘`interr-movi-caus-clienti-02.sql`’.

79.6 Interrogazioni simultanee di più relazioni e alias

Quando si interrogano simultaneamente più relazioni, può succedere che il risultato che si ottiene contenga degli attributi di relazioni differenti, ma con lo stesso nome, oppure potrebbe non essere abbastanza esplicito il suo contenuto. Nell’istruzione ‘**`SELECT`**’ con cui si esegue l’interrogazione, è possibile dichiarare dei nomi alternativi agli attributi, secondo le modalità descritte in questa sezione.

79.6.1 Interrogazione simultanea delle relazioni «Movimenti», «Articoli» e «Causali»

Si riprenda il file ‘`prova-interr-movi-arti.sql`’ e lo si modifichi in modo da avere il contenuto seguente:

```
-- Interrogazione delle relazioni "Movimenti", "Articoli"
-- e "Causali"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: prova-interr-movi-arti.sql

.headers on
.mode column

SELECT Movimenti.Data,
       Articoli.Descrizione AS Articolo,
       Causali.Descrizione AS Causale
       FROM Movimenti, Articoli, Causali
       WHERE Movimenti.Articolo = Articoli.Articolo
         AND Movimenti.Causale = Causali.Causale;
```

Si controlli di avere modificato il file '`prova-interr-movi-arti.sql`' in modo corretto, rispettando anche la punteggiatura; si controlli di avere salvato il file con il nome previsto, quindi si proceda con il comando seguente:

```
$ sqlite3 mag.db < prova-interr-movi-arti.sql[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Data        Articolo                                Causale
----------  --------------------------------------  ------------------
2012-01-15  Dischetti da 9 cm 1440 Kibyte colorati  Carico per acquisto
2012-01-16  Dischetti da 9 cm 1440 Kibyte colorati  Scarico per vendita
2012-01-17  CD-R 52x                                Carico per acquisto
2012-01-18  CD-R 52x                                Scarico per vendita
2012-01-19  DVD+R 8x                                Carico per acquisto
2012-01-20  DVD+R 8x                                Scarico per vendita
2012-01-20  DVD+R 8x                                Reso a fornitore
2012-01-20  CD-R 52x                                Reso a fornitore
2012-01-21  DVD+RW 8x                               Carico per acquisto
2012-01-25  DVD+RW 8x                               Scarico per vendita
```

Come si può osservare, l'attributo '**Descrizione**' della relazione '**Articoli**' appare con il nome '**Articolo**', mentre l'attributo '**Descrizione**' della relazione '**Causali**' appare con il nome '**Causale**'.

79.6.2 Verifica sull'interrogazione simultanea delle relazioni «Movimenti» e «Causali»

Si prepari il file '`interr-movi-caus-02.sql`', seguendo lo scheletro seguente, tenendo conto che si vuole ottenere l'elenco di tutte le tuple per le quali si possa stabilire un abbinamento in base al codice della causale. Precisamente, si vuole ottenere l'attributo '**Articolo**' dalla relazione '**Movimenti**'; l'attributo '**Descrizione**' dalla relazione '**Causali**'; l'attributo '**Data**' dalla relazione '**Movimenti**'. Inoltre, si vuole che l'attributo '**Descrizione**' della relazione '**Causali**', appaia con il nome '**Causale**'.

Figura 79.65. Scheletro del file '`interr-movi-caus-02.sql`', da completare.

```
-- Interrogazione delle relazioni "Movimenti" e "Causali"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-movi-caus-02.sql

.headers on
.mode column

SELECT …
       FROM …
       WHERE …
```

Una volta completato e salvato il file '`interr-movi-caus-02.sql`', se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < interr-movi-caus-02.sql [Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
Articolo    Causale              Data
----------  -------------------  ----------
2           Carico per acquisto  2012-01-15
2           Scarico per vendita  2012-01-16
102         Carico per acquisto  2012-01-17
102         Scarico per vendita  2012-01-18
401         Carico per acquisto  2012-01-19
401         Scarico per vendita  2012-01-20
401         Reso a fornitore     2012-01-20
102         Reso a fornitore     2012-01-20
601         Carico per acquisto  2012-01-21
601         Scarico per vendita  2012-01-25
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-movi-caus-02.sql | lpr [Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo, assieme alla stampa del file ‘`interr-movi-caus-02.sql`’.

79.6.3 Verifica sull’interrogazione simultanea delle relazioni «Movimenti», «Causali» e «Clienti»

«

Si prepari il file ‘`interr-movi-caus-clienti-03.sql`’, seguendo lo scheletro seguente, tenendo conto che si vuole ottenere l’elenco di tutte le tuple per le quali si possa stabilire un abbinamento in base al codice della causale e in base al codice del cliente. Precisamente, si vuole ottenere l’attributo ‘**`Articolo`**’ dalla relazione ‘**`Movimenti`**’; l’attributo ‘**`Descrizione`**’ dalla relazione ‘**`Causali`**’; l’attributo ‘**`Data`**’ dalla relazione ‘**`Movimenti`**’; l’at-

tributo '**RagioneSociale**' dalla relazione '**Clienti**'. L'attributo '**Descrizione**' della relazione '**Causali**' deve apparire con il nome '**Causale**' e l'attributo '**RagioneSociale**' della relazione '**Clienti**' deve apparire con il nome '**Cliente**'.

Figura 79.67. Scheletro del file '`interr-movi-caus-clienti-03.sql`', da completare.

```
-- Interrogazione delle relazioni "Movimenti", "Causali"
-- e "Clienti"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: interr-movi-caus-clienti-03.sql

.headers on
.mode column

SELECT …
       FROM …
       WHERE …
```

Una volta completato e salvato il file '`interr-movi-caus-clienti-03.sql`', se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < interr-movi-caus-clienti-03.sql[Invio]
```

Si dovrebbe ottenere il risultato seguente:

```
Articolo    Causale              Data        Cliente
----------  -------------------  ----------  --------------
2           Scarico per vendita  2012-01-16  Filano Filani
102         Scarico per vendita  2012-01-18  Mevio Mevi
401         Scarico per vendita  2012-01-20  Martino Martin
601         Scarico per vendita  2012-01-25  Mevio Mevi
```

Se il risultato è corretto, si proceda con la stampa:

```
$ sqlite3 mag.db < interr-movi-caus-clienti-03.sql ↩
↪         | lpr[Invio]
```

Si consegni per la valutazione, la stampa ottenuta in questo modo, assieme alla stampa del file ‘`interr-movi-caus-clienti-03.sql`’.

79.6.4 Conclusione

«

Il file ‘`prova-interr-movi-arti.sql`’ non serve più e va cancellato.

79.7 Viste

«

È possibile trasformare l’interrogazione di una o più relazioni in una ***vista***, la quale diventa in pratica una relazione virtuale.

79.7.1 Creazione della vista «Listino»

«

Con l’ausilio di un programma per la scrittura e modifica di file di testo puro, si crei il file ‘`prova-vista-listino.sql`’, contenente il testo seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Creazione della vista "Listino"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: prova-vista-listino.sql

CREATE VIEW Listino AS
            SELECT Articolo AS Codice,
                   Descrizione AS Articolo,
                   Listino AS EUR
                   FROM Articoli;
```

In questo modo, si crea la vista '**Listino**', composta dagli attributi '**Codice**', '**Articolo**' e '**EUR**', utilizzando, rispettivamente, gli attributi '**Articolo**', '**Descrizione**' e '**Listino**' dalla relazione '**Articoli**'.

Si controlli di avere scritto il file '`prova-vista-listino.sql`' in modo corretto, rispettando anche la punteggiatura; si controlli di avere salvato il file con il nome previsto, quindi si proceda con il comando seguente:

```
$ sqlite3 mag.db < prova-vista-listino.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la creazione della vista '**Listino**' ha avuto successo, altrimenti, è stato commesso un errore. Per rimediare all'errore, si deve prima cancellare la vista, quindi si può ritentare l'inserimento del comando (ammesso che il file '`prova-vista-listino.sql`' sia stato corretto di conseguenza). I passaggi per eliminare la vista, in modo interattivo, sono quelli seguenti:

```
$ sqlite3 mag.db[Invio]
```

```
  SQLite version …
  Enter ".help" for instructions
sqlite> DROP VIEW Listino;[Invio]

sqlite> .quit[Invio]
```

Quando si è consapevoli di avere creato correttamente la vista '**Listino**', la si può interrogare come se fosse una relazione normale. Si esegua il procedimento seguente, in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM Listino;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
  Codice      Articolo                        EUR
  ----------  ------------------------------  ----------
  1           Dischetti da 9 cm 1440 Kibyte   0.2
  2           Dischetti da 9 cm 1440 Kibyte   0.25
  101         CD-R 16x                        0.5
  102         CD-R 52x                        1
  201         CD-RW 4x                        1
  202         CD-RW 8x                        1.5
  301         DVD-R 8x                        1
  302         DVD-R 16x                       2
  401         DVD+R 8x                        1
  402         DVD+R 16x                       2
```

```
501          DVD-RW 8x                     2
601          DVD+RW 8x                     2
```

Come sempre, si conclude il funzionamento interattivo di '**sqlite3**' con il comando '**.quit**':

```
sqlite> .quit[Invio]
```

79.7.2 Creazione della vista «Resi»

Con l'ausilio di un programma per la scrittura e modifica di file di testo puro, si crei il file '`prova-vista-resi.sql`', contenente il testo seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Creazione della vista "Resi" (resi a fornitori)
-- Esercizio di: cognome  nome  classe
-- Data: data
-- File: prova-vista-resi.sql

CREATE VIEW Resi AS
            SELECT Articoli.Descrizione     AS Articolo,
                   Movimenti.Data           AS Data,
                   Fornitori.RagioneSociale AS Fornitore,
                   Movimenti.Quantita       AS Reso,
                   Movimenti.Valore         AS Valore
                   FROM Articoli, Movimenti, Fornitori
                   WHERE Movimenti.Causale   = 4
                     AND Movimenti.Articolo
                         = Articoli.Articolo
                     AND Movimenti.Fornitore
                         = Fornitori.Fornitore;
```

In questo modo, si crea la vista '**Resi**', utilizzando le relazioni '**Articoli**', '**Movimenti**' e '**Fornitori**', limitando la selezione

delle tuple della relazione '**Movimenti**' a quelle che riguardano un reso a fornitore, in quanto la causale corrisponde al numero quattro.

Si controlli di avere scritto il file '`prova-vista-resi.sql`' in modo corretto, rispettando anche la punteggiatura; si controlli di avere salvato il file con il nome previsto, quindi si proceda con il comando seguente:

```
$ sqlite3 mag.db < prova-vista-resi.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la creazione della vista '**Resi**' ha avuto successo, altrimenti, è stato commesso un errore. Per rimediare all'errore, si deve prima cancellare la vista, quindi si può ritentare l'inserimento del comando (ammesso che il file '`prova-vista-resi.sql`' sia stato corretto di conseguenza). I passaggi per eliminare la vista, in modo interattivo, sono quelli seguenti:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> DROP VIEW Resi;[Invio]

sqlite> .quit[Invio]
```

Quando si è consapevoli di avere creato correttamente la vista '**Resi**', la si può interrogare come se fosse una relazione normale. Si esegua il procedimento seguente, in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
```

```
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM Resi;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Articolo    Data        Fornitore   Reso        Valore
----------  ----------  ----------  ----------  ----------
CD-R 52x    2012-01-20  Caio Cai    100         20
DVD+R 8x    2012-01-20  Tizio Tizi  100         20
```

Come sempre, si conclude il funzionamento interattivo di '**sqlite3**' con il comando '**.quit**':

```
sqlite> .quit[Invio]
```

79.7.3 Verifica sulla creazione della vista «Acquisti»

Si prepari il file '`vista-acquisti.sql`', seguendo lo scheletro seguente, tenendo conto che si vuole ottenere un elenco dei movimenti di magazzino che riguardano i carichi per acquisto (causale uno). La vista deve essere composta dagli attributi seguenti:

1. '**Articolo**', corrispondente alla descrizione dell'articolo acquistato;
2. '**Data**', corrispondente alla data di acquisto;
3. '**Fornitore**', corrispondente alla ragione sociale del fornitore dal quale l'articolo è stato acquistato;
4. '**Acquistato**', corrispondente alla quantità acquistata;
5. '**Valore**', corrispondente al valore complessivo caricato (pari all'attributo con lo stesso nome della relazione '**Movimenti**').

Figura 79.77. Scheletro del file ‘`vista-acquisti.sql`’, da completare.

```
-- Creazione della vista "Acquisti"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: vista-acquisti.sql

CREATE VIEW …
       SELECT …
              FROM …
              WHERE …
```

Una volta completato e salvato il file ‘`vista-acquisti.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < vista-acquisti.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la creazione della vista ‘**`Acquisti`**’ ha avuto successo, altrimenti, è stato commesso un errore. Per rimediare all’errore, si deve prima cancellare la vista, quindi si può ritentare l’inserimento del comando (ammesso che il file ‘`vista-acquisti.sql`’ sia stato corretto di conseguenza).

Quando si è consapevoli di avere creato correttamente la vista ‘**`Acquisti`**’, la si può interrogare come se fosse una relazione normale. Si esegua il procedimento seguente, in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]
```

```
sqlite> .mode column [Invio]

sqlite> SELECT * FROM Acquisti; [Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Articolo                              Data        Fornitore           Acquistato  Valore
------------------------------------  ----------  ------------------  ----------  ----------
Dischetti da 9 cm 1440 Kibyte colorati  2012-01-15  Sempronio Semproni  10000       100
CD-R 52x                              2012-01-17  Caio Cai            1000        200
DVD+R 8x                              2012-01-19  Tizio Tizi          1000        200
DVD+RW 8x                             2012-01-21  Sempronio Semproni  2000        1000
```

Se tutto funziona regolarmente, si consegni per la valutazione la stampa del file ‘`vista-acquisti.sql`’.

79.7.4 Verifica sulla creazione della vista «Vendite»

Si prepari il file ‘`vista-vendite.sql`’, seguendo lo scheletro seguente, tenendo conto che si vuole ottenere un elenco dei movimenti di magazzino che riguardano gli scarichi per vendita (causale due). La vista deve essere composta dagli attributi seguenti:

1. ‘**`Articolo`**’, corrispondente alla descrizione dell’articolo venduto;
2. ‘**`Data`**’, corrispondente alla data di vendita;
3. ‘**`Cliente`**’, corrispondente alla ragione sociale del cliente al quale l’articolo è stato venduto;
4. ‘**`Venduto`**’, corrispondente alla quantità venduta;
5. ‘**`Valore`**’, corrispondente al valore complessivo scaricato (pari all’attributo con lo stesso nome della relazione ‘**`Movimenti`**’).

Figura 79.80. Scheletro del file ‘`vista-vendite.sql`’, da completare.

```
-- Creazione della vista "Vendite"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: vista-vendite.sql

CREATE VIEW …
       SELECT …
              FROM …
              WHERE …
```

Una volta completato e salvato il file ‘`vista-vendite.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < vista-vendite.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la creazione della vista ‘**`Vendite`**’ ha avuto successo, altrimenti, è stato commesso un errore. Per rimediare all’errore, si deve prima cancellare la vista, quindi si può ritentare l’inserimento del comando (ammesso che il file ‘`vista-vendite.sql`’ sia stato corretto di conseguenza).

Quando si è consapevoli di avere creato correttamente la vista ‘**`Vendite`**’, la si può interrogare come se fosse una relazione normale. Si esegua il procedimento seguente, in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions

sqlite> .headers on[Invio]
```

```
sqlite> .mode column [Invio]

sqlite> SELECT * FROM Vendite; [Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Articolo                              Data        Cliente         Venduto     Valore
------------------------------------  ----------  -------------   ----------  ----------
Dischetti da 9 cm 1440 Kibyte colorati  2012-01-16  Filano Filani  1000        10
CD-R 52x                              2012-01-18  Mevio Mevi      100         20
DVD+R 8x                              2012-01-20  Martino Marti   200         20
DVD+RW 8x                             2012-01-25  Mevio Mevi      1000        500
```

Se tutto funziona regolarmente, si consegni per la valutazione la stampa del file ‘`vista-vendite.sql`’.

79.7.5 Conclusione

Prima di proseguire, si deve riprendere il file ‘`magazzino.sql`’ e vi si devono aggiungere le istruzioni per la creazione delle viste ‘**Acquisti**’ e ‘**Vendite**’, come contenuto nei file ‘`vista-acquisti.sql`’ e ‘`vista-vendite.sql`’.

Una volta aggiornato il file ‘`magazzino.sql`’ come descritto, si deve cancellare il file ‘`mag.db`’ e ricreare a partire dalle istruzioni contenute nel file ‘`magazzino.sql`’:

```
$ sqlite3 mag.db < magazzino.sql [Invio]
```

Se vengono segnalati degli errori, occorre correggere il file ‘`magazzino.sql`’, cancellare nuovamente il file ‘`mag.db`’, quindi si deve ripetere l’operazione. La base di dati contenuta nel file ‘`mag.db`’, viene usata ancora e non si può proseguire se non si riesce a ricrearla correttamente.

In precedenza sono stati creati i file ‘`prova-vista-listino.sql`’ e ‘`prova-vista-resi.sql`’, che a questo punto non servono più e vanno cancellati.

79.8 Modifica del contenuto delle tuple

Una volta inserita una tupla in una relazione, si può modificare il suo contenuto con l’istruzione ‘**`UPDATE`**’, la quale si applica a tutte le tuple che soddisfano una certa condizione.

79.8.1 Modifica di una causale di magazzino

Con l’ausilio di un programma per la scrittura e modifica di file di testo puro, si crei il file ‘`prova-modifica-causali.sql`’, contenente il testo seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Modifica della relazione "Causali"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: prova-modifica-causali.sql

UPDATE Causali
       SET Descrizione = 'car x acq'
       WHERE Causale = 1;
```

In questo modo, si vuole modificare la tupla della relazione ‘**`Causali`**’, con il codice causale uno, in modo che la descrizione risulti molto più breve.

Si controlli di avere scritto il file ‘`prova-modifica-causali.sql`’ in modo corretto, rispettando anche la punteggiatura; si controlli di avere salvato il file con il nome previsto, quindi si proceda con il comando seguente:

```
$ sqlite3 mag.db < prova-modifica-causali.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la modifica della tupla dovrebbe essere avvenuta con successo, altrimenti, è stato commesso un errore. Per rimediare all'errore, dovrebbe essere sufficiente modificare il file 'prova-modifica-causali.sql' e riprovare.

Quando si è consapevoli di avere modificato correttamente la tupla in questione, si può interrogare la relazione per verificare i cambiamenti apportati. Si esegua il procedimento seguente, in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM Causali;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
  Causale     Descrizione  Variazione
  ----------  -----------  ----------
  1           car x acq    1
  2           Scarico per  -1
  3           Reso da cli  1
  4           Reso a forn  -1
  5           Rettifica a  1
  6           Rettifica a  -1
  7           Rettifica d  1
  8           Rettifica d  -1
```

```
9           Carico da p  1
10          Scarico a p  -1
11          Carico da a  1
12          Scarico ad   -1
13          Saldo inizi  1
```

Come sempre, si conclude il funzionamento interattivo di '**sqlite3**' con il comando '**.quit**':

```
sqlite> .quit[Invio]
```

Prima di passare alla sezione successiva, si deve ripristinare la base di dati al suo stato precedente. Per questo, è necessario cancellare il file 'mag.db' e poi ricrearlo con il comando seguente:

```
$ sqlite3 mag.db < magazzino.sql[Invio]
```

79.8.2 Modifica di diverse causali di magazzino

«

Si riprenda il file 'prova-modifica-causali.sql' e lo si modifichi secondo la forma seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Modifica della relazione "Causali"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: prova-modifica-causali.sql

UPDATE Causali
       SET Descrizione = UPPER (Descrizione)
       WHERE Variazione = 1;

UPDATE Causali
       SET Descrizione = LOWER (Descrizione)
       WHERE Variazione = -1;
```

In questo modo, si vuole modificare ogni tupla della relazione '**Causali**' che corrisponde a un aumento di quantità in magazzino (in quanto nell'attributo '**Variazione**' ha il valore +1), in modo da avere una descrizione con tutte lettere maiuscole. Nel contempo, si vuole che le descrizione associate a movimenti in diminuzione, siano scritte utilizzando soltanto caratteri minuscoli.

Si controlli di avere scritto il file '`prova-modifica-causali.sql`' in modo corretto, rispettando anche la punteggiatura; si controlli di avere salvato il file con il nome previsto, quindi si proceda con il comando seguente:

```
$ sqlite3 mag.db < prova-modifica-causali.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la modifica delle tuple dovrebbe essere stata eseguita con successo, altrimenti, è stato commesso un errore. Per rimediare all'errore, dovrebbe essere sufficiente modificare il file '`prova-modifica-causali.sql`' e riprovare.

Quando si è consapevoli di avere modificato correttamente le tuple, si può interrogare la relazione per verificare i cambiamenti apportati. Si esegua il procedimento seguente, in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM Causali;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Causale     Descrizione          Variazione
----------  -------------------  ----------
1           CARICO PER ACQUISTO  1
2           scarico per vendita  -1
3           RESO DA CLIENTE      1
4           reso a fornitore     -1
5           RETTIFICA AUMENTO A  1
6           rettifica aumento v  -1
7           RETTIFICA DIMINUZIO  1
8           rettifica diminuzio  -1
9           CARICO DA PRODUZION  1
10          scarico a produzion  -1
11          CARICO DA ALTRO MAG  1
12          scarico ad altro ma  -1
13          SALDO INIZIALE       1
```

Come sempre, si conclude il funzionamento interattivo di '**`sqlite3`**' con il comando '**`.quit`**':

`sqlite>` **`.quit`** [*Invio*]

Prima di passare alla sezione successiva, si deve ripristinare la base di dati al suo stato precedente. Per questo, è necessario cancellare il file '`mag.db`' e poi ricrearlo con il comando seguente:

`$` **`sqlite3 mag.db < magazzino.sql`** [*Invio*]

79.8.3 Verifica sulla modifica della relazione «Articoli»

«

Si prepari il file '`modifica-articoli.sql`', seguendo lo scheletro seguente, tenendo conto che si vuole cambiare la descrizione del primo e del secondo articolo, in modo da avere rispettivamen-

te: «Floppy 1.4» e «Floppy 1.4 C». Per ottenere questo risultato è necessario utilizzare due volte l'istruzione '**UPDATE**'.

Figura 79.89. Scheletro del file 'modifica-articoli.sql', da completare.

```
-- Modifica della relazione "Articoli"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: modifica-articoli.sql

UPDATE Articoli
       SET …
       WHERE Articolo = 1;

UPDATE Articoli
       SET …
       WHERE Articolo = 2;
```

Una volta completato e salvato il file 'modifica-articoli.sql', se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < modifica-articoli.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la modifica delle tuple dovrebbe essere avvenuta con successo, altrimenti è stato commesso un errore. Per rimediare all'errore dovrebbe essere sufficiente correggere il file 'modifica-articoli.sql' e riprovare. Quando si ritiene di avere eseguito l'operazione correttamente, si può interrogare la relazione '**Articoli**' per verificarne il risultato. Si esegua il procedimento seguente, in modo interattivo:

```
$ sqlite3 mag.db[Invio]
```

```
  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM Articoli;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
  Articolo    Descrizione  UM          Listino     ScortaMin
  ----------  -----------  ----------  ----------  ----------
  1           Floppy 1.4   pz          0.2         500
  2           Floppy 1.4   pz          0.25        500
  101         CD-R 16x     pz          0.5         500
  102         CD-R 52x     pz          1           500
  201         CD-RW 4x     pz          1           200
  202         CD-RW 8x     pz          1.5         200
  301         DVD-R 8x     pz          1           200
  302         DVD-R 16x    pz          2           200
  401         DVD+R 8x     pz          1           200
  402         DVD+R 16x    pz          2           200
  501         DVD-RW 8x    pz          2           200
  601         DVD+RW 8x    pz          2           200
```

Se tutto funziona regolarmente, si consegni per la valutazione la stampa del file ‘`modifica-articoli.sql`’.

Prima di passare alla sezione successiva, si deve ripristinare la base di dati al suo stato precedente. Per questo, è necessario cancellare il file ‘`mag.db`’ e poi ricrearlo con il comando seguente:

```
$ sqlite3 mag.db < magazzino.sql[Invio]
```

79.8.4 Verifica sulla modifica delle relazioni «Clienti» e «Fornitori»

Si prepari il file ‘`modifica-clienti-fornitori.sql`’, seguendo lo scheletro seguente, tenendo conto che si vuole cambiare la ragione sociale delle relazioni ‘**`Clienti`**’ e ‘**`Fornitori`**’, in modo che sia costituita da caratteri maiuscoli. Pertanto, la sostituzione riguarda tutte le tuple in entrambe le relazioni.

Figura 79.92. Scheletro del file ‘`modifica-clienti-fornitori.sql`’, da completare.

```
-- Modifica delle relazioni "Clienti" e "Fornitori"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: modifica-clienti-fornitori.sql

UPDATE Clienti
       SET …

UPDATE Fornitori
       SET …
```

Una volta completato e salvato il file ‘`modifica-clienti-fornitori.sql`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < modifica-clienti-fornitori.sql [Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la modifica delle tuple dovrebbe essere avvenuta con successo, altrimenti è stato commesso un errore. Per rimediare all’errore dovrebbe essere sufficiente correggere il file ‘`modifica-articoli.sql`’ e riprovare. Quando si ritiene di avere eseguito l’operazione correttamente, si

possono interrogare le due relazioni per verificarne il contenuto. Si esegua il procedimento seguente, in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM Clienti;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Cliente RagioneSociale Paese  Indirizzo    CAP   Citta     Prov Telefono    Fax          CFPI
------- -------------- ------ ------------ ----- --------- ---- ----------- ------------ -----------
1       MEVIO VEVI     ITALIA via Mare, 11 31050 Morgano   TV   0422,444444 0422,555555 45678901234
2       FILANO FILANI  ITALIA via Farfalle 31032 Feltre    BL   0439,555555 0439,666666 56789012345
3       MARTINO MARTIN ITALIA via Marte, 3 31010 Mareno di TV   0438,666666 0438,777777 67890123456
```

```
sqlite> SELECT * FROM Fornitori;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Fornitore RagioneSociale Paese  Indirizzo     CAP   Citta      Prov Telefono    Fax          CFPI
--------- -------------- ------ ------------- ----- ---------- ---- ----------- ------------ -----------
1         TIZIO TIZI     ITALIA via Tazio, 11 31100 Treviso    TV   0422,111111 0422,222222 12345678901
2         CAIO CAI       ITALIA via Caino, 22 31033 Castelfran TV   0423,222222 0423,333333 23456789012
3         SEMPRONIO SEMP ITALIA via Salina, 3 31057 Silea      TV   0422,333333 0422,444444 34567890123
```

Se tutto funziona regolarmente, si consegni per la valutazione la stampa del file ‘`modifica-clienti-fornitori.sql`’.

Prima di passare alla sezione successiva, si deve ripristinare la base di dati al suo stato precedente. Per questo, è necessario cancellare il file ‘`mag.db`’ e poi ricrearlo con il comando seguente:

```
$ sqlite3 mag.db < magazzino.sql[Invio]
```

79.8.5 Conclusione

Il file ‘`prova-modifica-causali.sql`’ non serve più e va cancellato.

79.9 Eliminazione delle tuple

La cancellazione delle tuple avviene attraverso l’istruzione ‘**`DELETE FROM`**’, con un procedimento simile a quello della modifica, in quanto va specificata la condizione di cancellazione, altrimenti si ottiene l’eliminazione di tutte le tuple della relazione.

79.9.1 Cancellazione di una causale di magazzino

Con l’ausilio di un programma per la scrittura e modifica di file di testo puro, si crei il file ‘`prova-cancella-causali.sql`’, contenente il testo seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Cancellazione nella relazione "Causali"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: prova-cancella-causali.sql

DELETE FROM Causali
       WHERE Causale = 1;
```

In questo modo, si vuole eliminare la tupla della relazione ‘**`Causali`**’, con il codice causale uno (quella che ha la descrizione «Carico per acquisto»).

Si controlli di avere scritto il file ‘`prova-cancella-causali.sql`’ in modo corretto, rispettando anche la punteggiatura; si con-

trolli di avere salvato il file con il nome previsto, quindi si proceda con il comando seguente:

```
$ sqlite3 mag.db < prova-cancella-causali.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la cancellazione della tupla dovrebbe essere avvenuta con successo, altrimenti, è stato commesso un errore. Per rimediare all'errore, dovrebbe essere sufficiente modificare il file '`prova-cancella-causali.sql`' e riprovare. Quando si ritiene di avere cancellato la tupla in questione, si può interrogare la relazione per verificarne lo stato. Si esegua il procedimento seguente, in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM Causali;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
  Causale     Descrizione          Variazione
  ----------  -------------------  ----------
  2           Scarico per vendita  -1
  3           Reso da cliente      1
  4           Reso a fornitore     -1
  5           Rettifica aumento a  1
  6           Rettifica aumento v  -1
  7           Rettifica diminuzio  1
  8           Rettifica diminuzio  -1
  9           Carico da produzion  1
```

```
10          Scarico a produzion  -1
11          Carico da altro mag  1
12          Scarico ad altro ma  -1
13          Saldo iniziale       1
```

Come sempre, si conclude il funzionamento interattivo di '**sqlite3**' con il comando '**.quit**':

```
sqlite> .quit[Invio]
```

Prima di passare alla sezione successiva, si deve ripristinare la base di dati al suo stato precedente. Per questo, è necessario cancellare il file 'mag.db' e poi ricrearlo con il comando seguente:

```
$ sqlite3 mag.db < magazzino.sql[Invio]
```

79.9.2 Cancellazione di diverse causali di magazzino

Si riprenda il file 'prova-cancella-causali.sql' e lo si modifichi secondo la forma seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Cancellazione nella relazione "Causali"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: prova-cancella-causali.sql

DELETE FROM Causali
       WHERE Variazione = -1;
```

In questo modo, si vogliono eliminare le tuple corrispondenti a una riduzione della quantità in magazzino, (in quanto nell'attributo '**Variazione**' ha il valore −1).

Si controlli di avere scritto il file ‘`prova-cancella-causali.sql`’ in modo corretto, rispettando anche la punteggiatura; si controlli di avere salvato il file con il nome previsto, quindi si proceda con il comando seguente:

```
$ sqlite3 mag.db < prova-cancella-causali.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la cancellazione dovrebbe avere avuto successo, altrimenti, è stato commesso un errore. Per rimediare all'errore, dovrebbe essere sufficiente modificare il file ‘`prova-cancella-causali.sql`’ e riprovare. Quando si ritiene di avere eseguito l'operazione con successo, si può interrogare la relazione per verificare i cambiamenti apportati. Si esegua il procedimento seguente, in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM Causali;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
  Causale     Descrizione          Variazione
  ----------  -------------------  ----------
  1           Carico per acquisto  1
  3           Reso da cliente      1
  5           Rettifica aumento a  1
  7           Rettifica diminuzio  1
  9           Carico da produzion  1
```

```
11          Carico da altro mag  1
13          Saldo iniziale       1
```

Come sempre, si conclude il funzionamento interattivo di '**sqlite3**' con il comando '**.quit**':

```
sqlite> .quit[Invio]
```

Prima di passare alla sezione successiva, si deve ripristinare la base di dati al suo stato precedente. Per questo, è necessario cancellare il file 'mag.db' e poi ricrearlo con il comando seguente:

```
$ sqlite3 mag.db < magazzino.sql[Invio]
```

79.9.3 Verifica sulla cancellazione di alcuni articoli

Si prepari il file 'cancella-articoli.sql', seguendo lo scheletro seguente, tenendo conto che si vogliono eliminare i dischetti (i primi due).

Figura 79.102. Scheletro del file 'cancella-articoli.sql', da completare.

```
-- Cancellazione di alcune tuple della relazione "Articoli"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: cancella-articoli.sql

DELETE FROM Articoli
       WHERE …

DELETE FROM Articoli
       WHERE …
```

Una volta completato e salvato il file 'cancella-articoli.sql', se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < cancella-articoli.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la cancellazione delle tuple dovrebbe essere avvenuta con successo, altrimenti è stato commesso un errore. Per rimediare all'errore dovrebbe essere sufficiente correggere il file '`modifica-articoli.sql`' e riprovare. Quando si ritiene di avere eseguito l'operazione correttamente, si può interrogare la relazione '**`Articoli`**' per verificarne il risultato. Si esegua il procedimento seguente, in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM Articoli;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
  Articolo    Descrizione  UM          Listino     ScortaMin
  ----------  -----------  ----------  ----------  ----------
  101         CD-R 16x     pz          0.5         500
  102         CD-R 52x     pz          1           500
  201         CD-RW 4x     pz          1           200
  202         CD-RW 8x     pz          1.5         200
  301         DVD-R 8x     pz          1           200
  302         DVD-R 16x    pz          2           200
  401         DVD+R 8x     pz          1           200
  402         DVD+R 16x    pz          2           200
  501         DVD-RW 8x    pz          2           200
  601         DVD+RW 8x    pz          2           200
```

Se tutto funziona regolarmente, si consegni per la valutazione la stampa del file ‘`cancella-articoli.sql`’.

Prima di passare alla sezione successiva, si deve ripristinare la base di dati al suo stato precedente. Per questo, è necessario cancellare il file ‘`mag.db`’ e poi ricrearlo con il comando seguente:

```
$ sqlite3 mag.db < magazzino.sql [Invio]
```

79.9.4 Conclusione

Il file ‘`prova-cancella-causali.sql`’ non serve più e va cancellato.

79.10 Grilletti per il controllo del dominio degli attributi

Nel momento in cui si inseriscono o si modificano i valori per una tupla di una certa relazione, può essere importante fare in modo di rifiutare i valori impossibili, in quanto non facenti parte del dominio previsto per gli attributi della stessa. Di solito, questo tipo di controllo può essere dichiarato in fase di creazione della relazione; tuttavia, un DBMS limitato potrebbe ignorare tali dichiarazioni.

I ***grilletti*** sono delle funzioni che «scattano», in quanto vengono eseguite, quando si verificano certi eventi. Attraverso i grilletti è possibile impedire l’inserimento di valori errati all’interno degli attributi e questo è l’obiettivo della sezione.

79.10.1 Creazione dei grilletti «Causali_ins» e «Causali_upd»

«

Con l'ausilio di un programma per la scrittura e modifica di file di testo puro, si crei il file '`grilletti-causali.sql`', contenente il testo seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Creazione dei grilletti "Causali_ins" e "Causali_upd"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: grilletti-causali.sql

CREATE TRIGGER Causali_ins
               BEFORE INSERT ON Causali
               FOR EACH ROW
               BEGIN
                    SELECT CASE
                    WHEN (NEW.Variazione > 1)
                    THEN
                        RAISE (ABORT, 'L''attributo "Variazione" non può essere superiore a 1!')
                    WHEN (NEW.Variazione < -1)
                    THEN
                        RAISE (ABORT, 'L''attributo "Variazione" non può essere inferiore a -1!')
                    WHEN (NEW.Variazione = 0)
                    THEN
                        RAISE (ABORT, 'L''attributo "Variazione" non può essere pari a 0!')
                    END;
                END;

CREATE TRIGGER Causali_upd
               BEFORE UPDATE ON Causali
               FOR EACH ROW
               BEGIN
                    SELECT CASE
                    WHEN (NEW.Variazione > 1)
                    THEN
                        RAISE (ABORT, 'L''attributo "Variazione" non può essere superiore a 1!')
                    WHEN (NEW.Variazione < -1)
                    THEN
                        RAISE (ABORT, 'L''attributo "Variazione" non può essere inferiore a -1!')
                    WHEN (NEW.Variazione = 0)
                    THEN
                        RAISE (ABORT, 'L''attributo "Variazione" non può essere pari a 0!')
                    END;
                END;
```

In questo modo, si creano i grilletti '**Causali_ins**' e '**Causali_upd**', con lo scopo di avvisare in caso di inserimento di un valore impossibile nell'attributo '**Variazione**' della relazione '**Causali**' (sia nel caso di inserimento di una tupla nuova, sia quando si cerca di modificare quell'attributo in una tupla già esistente). Si osservi che, all'interno dei messaggi di errore, l'apostrofo è stato raddoppiato, per evitare che possa essere interpretato come la conclusione della stringa.

Si controlli di avere scritto il file '`grilletti-causali.sql`' in modo corretto, rispettando anche la punteggiatura; si controlli di avere salvato il file con il nome previsto, quindi si proceda con il comando seguente:

```
$ sqlite3 mag.db < grilletti-causali.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la creazione dei grilletti dovrebbe essere avvenuta con successo, altrimenti, è stato commesso un errore. Per rimediare all'errore, si devono prima cancellare i grilletti, quindi si può ritentare l'inserimento del comando (ammesso che il file '`grilletti-causali.sql`' sia stato corretto di conseguenza). I passaggi per eliminare i grilletti, in modo interattivo, sono quelli seguenti:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> DROP TRIGGER Causali_ins;[Invio]

sqlite> DROP TRIGGER Causali_upd;[Invio]
```

```
sqlite> .quit[Invio]
```

Quando si ritiene di avere creato correttamente i grilletti, si può tentare l'inserimento o la modifica di tuple con valori errati nella relazione '**Causali**', per verificare se queste vengono rifiutate come dovrebbero. Si proceda con i passaggi seguenti, utilizzando '**sqlite3**' in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> INSERT INTO Causali VALUES (100, 'Doppio carico', +2);
[Invio]

  INSERT INTO Causali VALUES (100, 'Doppio carico', +2);
  SQL error: L'attributo "Variazione" non può essere ↩
 ↪superiore a 1!
sqlite> INSERT INTO Causali VALUES (101, 'Doppio scarico',
-2);[Invio]

  INSERT INTO Causali VALUES (101, 'Doppio scarico', -2);
  SQL error: L'attributo "Variazione" non può essere ↩
 ↪inferiore a -1!
sqlite> INSERT INTO Causali VALUES (102, 'Movimento nullo',
0);[Invio]

  INSERT INTO Causali VALUES (102, 'Movimento nullo', 0);
  SQL error: L'attributo "Variazione" non può essere pari a 0!
sqlite> UPDATE Causali SET Variazione = +2 WHERE Causale = 1;
[Invio]
```

```
  UPDATE Causali SET Variazione = +2 WHERE Causale = 1;
  SQL error: L'attributo "Variazione" non può essere ↩
 ↪superiore a 1!
```

```
sqlite> UPDATE Causali SET Variazione = -2 WHERE Causale = 2;
```
[*Invio*]

```
  UPDATE Causali SET Variazione = -2 WHERE Causale = 2;
  SQL error: L'attributo "Variazione" non può essere ↩
 ↪inferiore a -1!
```

```
sqlite> UPDATE Causali SET Variazione = 0 WHERE Causale = 3;
```
[*Invio*]

```
  UPDATE Causali SET Variazione = 0 WHERE Causale = 3;
  SQL error: L'attributo "Variazione" non può essere pari a 0!
```

```
sqlite> .quit
```
[*Invio*]

79.10.2 Creazione del grilletto «Articoli_ins» e «Articoli_upd»

Con l'ausilio di un programma per la scrittura e modifica di file di testo puro, si crei il file '`grilletti-articoli.sql`', contenente il testo seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Creazione dei grilletti "Articoli_ins" e "Articoli_upd"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: grilletti-articoli.sql

CREATE TRIGGER Articoli_ins
               BEFORE INSERT ON Articoli
               FOR EACH ROW
               BEGIN
                    SELECT CASE
                    WHEN (NEW.Listino <= 0)
                    THEN
                        RAISE (ABORT, 'Il prezzo non può essere inferiore o uguale a zero!')
                    WHEN (NEW.ScortaMin < 0)
                    THEN
```

```
                    RAISE (ABORT, 'La scorta minima non può essere inferiore a zero!')
                END;
            END;

CREATE TRIGGER Articoli_upd
            BEFORE UPDATE ON Articoli
            FOR EACH ROW
            BEGIN
                SELECT CASE
                WHEN (NEW.Listino <= 0)
                THEN
                    RAISE (ABORT, 'Il prezzo non può essere inferiore o uguale a zero!')
                WHEN (NEW.ScortaMin < 0)
                THEN
                    RAISE (ABORT, 'La scorta minima non può essere inferiore a zero!')
                END;
            END;
```

In questo modo, si creano i grilletti '**Articoli_ins**' e '**Articoli_upd**', con lo scopo di impedire l'inserimento di valori impossibili per il prezzo di listino e per la scorta minima (sia con le istruzioni '**INSERT**', sia con '**UPDATE**').

Si controlli di avere scritto il file '`grilletti-articoli.sql`' in modo corretto, rispettando anche la punteggiatura; si controlli di avere salvato il file con il nome previsto, quindi si proceda con il comando seguente:

`$` **`sqlite3 mag.db < grilletti-articoli.sql`** [*Invio*]

Se non si ottiene alcun messaggio da parte del programma, la creazione dei grilletti dovrebbe essere avvenuta con successo, altrimenti, è stato commesso un errore. Per rimediare all'errore, si devono prima cancellare i grilletti, quindi si può ritentare l'inserimento del comando (ammesso che il file '`grilletti-articoli.sql`' sia stato corretto di conseguenza). I passaggi per eliminare i grilletti, in modo interattivo, sono quelli seguenti:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> DROP TRIGGER Articoli_ins;[Invio]

sqlite> DROP TRIGGER Articoli_upd;[Invio]

sqlite> .quit[Invio]
```

Quando si ritiene di avere creato correttamente i grilletti in questione, si può tentare l'inserimento di tuple con valori errati nella relazione '**Articoli**', per verificare se queste vengono rifiutate come dovrebbero. Si proceda con i passaggi seguenti, utilizzando '**sqlite3**' in modo interattivo:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> INSERT INTO Articoli[Invio]

   ...>     VALUES (660, 'DVD gratis','pz', 0, 200);[Invio]

  INSERT INTO Articoli VALUES (660, 'DVD gratis','pz', 0, 200);
  SQL error: Il prezzo non può essere inferiore o uguale a zero!
sqlite> INSERT INTO Articoli[Invio]

   ...>     VALUES (661, 'DVD ti paghiamo noi','pz', -2.00,
200);[Invio]

  INSERT INTO Articoli VALUES (661, 'DVD ti paghiamo noi','pz', -2.00, 200);
  SQL error: Il prezzo non può essere inferiore o uguale a zero!
```

```
sqlite> INSERT INTO Articoli [Invio]

   ...>     VALUES (662, 'DVD virtuale','pz', 2.00, -200);
[Invio]

  INSERT INTO Articoli VALUES (662, 'DVD virtuale','pz', ↩
 ↪2.00, -200);
  SQL error: La scorta minima non può essere inferiore a zero!

sqlite> UPDATE Articoli SET Listino = 0 WHERE Articolo = 1;
[Invio]

  UPDATE Articoli SET Listino = 0 WHERE Articolo = 1;
  SQL error: Il prezzo non può essere inferiore o uguale a zero!

sqlite> UPDATE Articoli SET Listino = -2.00 WHERE Articolo =
2; [Invio]

  UPDATE Articoli SET Listino = -2.00 WHERE Articolo = 2;
  SQL error: Il prezzo non può essere inferiore o uguale a zero!

sqlite> UPDATE Articoli SET ScortaMin = -200 WHERE Articolo =
101; [Invio]

  UPDATE Articoli SET ScortaMin = -200 WHERE Articolo = 101;
  SQL error: La scorta minima non può essere inferiore a zero!

sqlite> .quit [Invio]
```

79.10.3 Verifica sulla creazione dei grilletti «Movimenti_ins» e «Movimenti_upd»

«

Si prepari il file ‘`grilletti-movimenti.sql`’, seguendo lo scheletro seguente, tenendo conto che si vuole impedire l’inserimento nella relazione ‘**Movimenti**’ di quantità inferiori o uguali a zero e di valori inferiori a zero.

Figura 79.123. Scheletro del file ‘`grilletto-movimenti.sql`’, da completare.

```
-- Creazione dei grilletti "Movimenti_ins" e "Movimenti_upd"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: grilletti-movimenti.sql

CREATE TRIGGER Movimenti_ins
               BEFORE INSERT …
               FOR EACH ROW
               BEGIN
                    …
                    …
                    …
               END;

CREATE TRIGGER Movimenti_upd
               BEFORE UPDATE …
               FOR EACH ROW
               BEGIN
                    …
                    …
                    …
               END;
```

Una volta completato e salvato il file ‘`grilletti-movimenti`’, se ne controlli il funzionamento con la base di dati:

```
$ sqlite3 mag.db < grilletti-movimenti.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la creazione dei grilletti dovrebbe essere avvenuta con successo, altrimenti, è stato commesso un errore. Per rimediare all’errore, si devono prima cancellare i grilletti, quindi si può ritentare l’inserimento del

comando (ammesso che il file ‘`grilletti-movimenti.sql`’ sia stato corretto di conseguenza).

Si consegni per la valutazione la stampa del file ‘`grilletti-movimenti.sql`’.

79.10.4 Conclusione

«

Prima di passare alla sezione successiva, si deve riprendere il file ‘`magazzino.sql`’ e vi si devono aggiungere le istruzioni per la creazione dei grilletti ‘**`Causali_ins`**’, ‘**`Causali_upd`**’, ‘**`Articoli_ins`**’, ‘**`Articoli_upd`**’, ‘**`Movimenti_ins`**’ e ‘**`Movimenti_upd`**’, come contenuto nei file ‘`grilletti-causali.sql`’, ‘`grilletti-articoli.sql`’ e ‘`grilletti-movimenti.sql`’.

Si osservi che la dichiarazione dei grilletti va collocata immediatamente dopo la creazione della relazione a cui fanno riferimento e immediatamente prima delle istruzioni che inseriscono delle tuple.

Una volta aggiornato il file ‘`magazzino.sql`’ come descritto, si deve cancellare il file ‘`mag.db`’ e ricreare a partire dalle istruzioni contenute nel file ‘`magazzino.sql`’:

```
$ sqlite3 mag.db < magazzino.sql[Invio]
```

Se vengono segnalati degli errori, occorre correggere il file ‘`magazzino.sql`’, cancellare nuovamente il file ‘`mag.db`’, quindi si deve ripetere l’operazione. La base di dati contenuta nel file ‘`mag.db`’, viene usata nella sezione successiva e non si può proseguire se non si riesce a ricrearla correttamente.

79.11 Grilletti per il controllo della validità esterna

Nel momento in cui si inseriscono, modificano o eliminano dei valori per una certa relazione, può essere importante fare in modo di rifiutare le azioni che non sono valide, in base al contenuto di altre relazioni. Di solito, questo tipo di controllo può essere dichiarato in fase di creazione della relazione; tuttavia, un DBMS limitato potrebbe ignorare tali dichiarazioni.

Qui si mostra l'uso dei grilletti per imporre dei vincoli di validità dipendenti dal contenuto di altre relazioni.

79.11.1 Controllo del codice articolo tra la relazione «Movimenti» e la relazione «Articoli»

In precedenza sono stati creati due grilletti, denominati '**`Movimenti_ins`**' e '**`Movimenti_upd`**', con lo scopo di impedire l'inserimento (o la modifica) di valori impossibili per la quantità e per il valore del movimento. Questi due grilletti vengono ripresi ed estesi, allo scopo di impedire che possano essere inseriti movimenti riferiti ad articoli inesistenti, in quanto non ancora dichiarati nella relazione '**`Articoli`**'; inoltre ne viene aggiunto un altro, per impedire che un articolo possa essere eliminato dalla relazione '**`Articoli`**', se questo risulta essere ancora utilizzato nella relazione '**`Movimenti`**'.

Pertanto, si crei il file '`grilletti-movimenti-articoli.sql`', contenente il testo seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Creazione dei grilletti "Movimenti_ins", "Movimenti_upd" e "Articoli_del"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: grilletti-movimenti-articoli.sql
```

```
CREATE TRIGGER Movimenti_ins
       BEFORE INSERT ON Movimenti
       FOR EACH ROW
       BEGIN
            SELECT CASE
            WHEN (NEW.Quantita <= 0)
            THEN
                RAISE (ABORT, 'La quantità non può essere inferiore o uguale a zero!')
            WHEN (NEW.Valore < 0)
            THEN
                RAISE (ABORT, 'Il valore caricato non può essere inferiore a zero!')
            WHEN ((SELECT Articolo FROM Articoli WHERE Articolo = NEW.Articolo) IS NULL)
            THEN
                RAISE (ABORT, 'Il codice articolo non è presente nella relazione Articoli!')
            END;
       END;

CREATE TRIGGER Movimenti_upd
       BEFORE UPDATE ON Movimenti
       FOR EACH ROW
       BEGIN
            SELECT CASE
            WHEN (NEW.Quantita <= 0)
            THEN
                RAISE (ABORT, 'La quantità non può essere inferiore o uguale a zero!')
            WHEN (NEW.Valore < 0)
            THEN
                RAISE (ABORT, 'Il valore caricato non può essere inferiore a zero!')
            WHEN ((SELECT Articolo FROM Articoli WHERE Articolo = NEW.Articolo) IS NULL)
            THEN
                RAISE (ABORT, 'Il codice articolo non è presente nella relazione Articoli!')
            END;
       END;

CREATE TRIGGER Articoli_del
       BEFORE DELETE ON Articoli
       FOR EACH ROW
       BEGIN
            SELECT CASE
            WHEN ((SELECT Articolo FROM Movimenti WHERE Articolo = OLD.Articolo) IS NOT NULL)
            THEN
                RAISE (ABORT, 'L''articolo non può essere rimosso, perché è utilizzato nella relazione Movimenti!')
            END;
       END;
```

Una volta completato e salvato il file '`grilletti-movimenti-articoli`', se ne deve controllare il funzionamento con la base di dati, ma prima vanno rimossi i grilletti '**`Movimenti_ins`**' e '**`Movimenti_upd`**', che qui vengono ricreati. Basta eseguire i passaggi seguenti:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> DROP TRIGGER Articoli_ins;[Invio]

sqlite> DROP TRIGGER Articoli_upd;[Invio]

sqlite> .quit[Invio]
```

Quando i grilletti preesistenti sono stati rimossi, si può eseguire il file ‘`grilletti-movimenti-articoli.sql`’ nella base di dati:

```
$ sqlite3 mag.db < grilletti-movimenti-articoli.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la creazione dei grilletti dovrebbe essere avvenuta con successo, altrimenti, è stato commesso un errore. Per rimediare all’errore, si devono prima cancellare i grilletti (questa volta sono tre: ‘**Movimenti_ins**’, ‘**Movimenti_upd**’ e ‘**Articoli_del**’), quindi si può ritentare l’inserimento del comando (ammesso che il file ‘`grilletti-movimenti-articoli.sql`’ sia stato corretto di conseguenza).

Per verificare che i vincoli dichiarati funzionino come previsto, si può provare a inserire un movimento che fa riferimento a un articolo inesistente; quindi, si può provare a cancellare un articolo che risulta invece movimentato:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> INSERT INTO Movimenti[Invio]
```

```
   ...>     VALUES (11, 777, 2, '2012-01-25', [Invio]

   ...>     1, NULL, 1000, 500.00); [Invio]

  INSERT INTO Movimenti VALUES (11, 777, 2, '2012-01-25', 1, NULL, 1000, 500.00);
  SQL error: Il codice articolo non è presente nella relazione Articoli!

sqlite> UPDATE Movimenti SET Articolo = 777 [Invio]

   ...>     WHERE Movimento = 2; [Invio]

  UPDATE Movimenti SET Articolo = 777 WHERE Movimento = 2;
  SQL error: Il codice articolo non è presente nella ↩
 ↪relazione Articoli!

sqlite> DELETE FROM Articoli WHERE Articolo = 2; [Invio]

  DELETE FROM Articoli WHERE Articolo = 2;
  SQL error: L'articolo non può essere rimosso, ↩
 ↪perché è utilizzato nella relazione Movimenti!

sqlite> .quit [Invio]
```

79.11.2 Controllo del codice cliente tra la relazione «Movimenti» e la relazione «Clienti»

«

Vengono qui ripresi i grilletti '**Movimenti_ins**' e '**Movimenti_upd**', aggiungendo il grilletto '**Clienti_del**', con lo scopo di impedire che possano essere inseriti movimenti riferiti a clienti inesistenti (in quanto non ancora dichiarati nella relazione '**Clienti**') e di impedire la cancellazione di un cliente quando questo risulta essere ancora utilizzato nella relazione '**Movimenti**'.

Pertanto, si crei il file '`grilletti-movimenti-clienti.sql`', contenente il testo seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Creazione dei grilletti "Movimenti_ins", "Movimenti_upd" e "Clienti_del"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: grilletti-movimenti-clienti.sql

CREATE TRIGGER Movimenti_ins
       BEFORE INSERT ON Movimenti
       FOR EACH ROW
       BEGIN
            SELECT CASE
            WHEN (NEW.Quantita <= 0)
            THEN
                RAISE (ABORT, 'La quantità non può essere inferiore o uguale a zero!')
            WHEN (NEW.Valore < 0)
            THEN
                RAISE (ABORT, 'Il valore caricato non può essere inferiore a zero!')
            WHEN ((SELECT Articolo FROM Articoli WHERE Articolo = NEW.Articolo) IS NULL)
            THEN
                RAISE (ABORT, 'Il codice articolo non è presente nella relazione Articoli!')
            WHEN ((NEW.Cliente IS NOT NULL)
                  AND ((SELECT Cliente FROM Clienti WHERE Cliente = NEW.Cliente) IS NULL))
            THEN
                RAISE (ABORT, 'Il codice cliente non è presente nella relazione Clienti!')
            END;
       END;

CREATE TRIGGER Movimenti_upd
       BEFORE UPDATE ON Movimenti
       FOR EACH ROW
       BEGIN
            SELECT CASE
            WHEN (NEW.Quantita <= 0)
            THEN
                RAISE (ABORT, 'La quantità non può essere inferiore o uguale a zero!')
            WHEN (NEW.Valore < 0)
            THEN
                RAISE (ABORT, 'Il valore caricato non può essere inferiore a zero!')
            WHEN ((SELECT Articolo FROM Articoli WHERE Articolo = NEW.Articolo) IS NULL)
            THEN
                RAISE (ABORT, 'Il codice articolo non è presente nella relazione Articoli!')
            WHEN ((NEW.Cliente IS NOT NULL)
                  AND ((SELECT Cliente FROM Clienti WHERE Cliente = NEW.Cliente) IS NULL))
            THEN
                RAISE (ABORT, 'Il codice cliente non è presente nella relazione Clienti!')
            END;
       END;

CREATE TRIGGER Clienti_del
       BEFORE DELETE ON Clienti
       FOR EACH ROW
       BEGIN
            SELECT CASE
            WHEN ((SELECT Cliente FROM Movimenti WHERE Cliente = OLD.Cliente) IS NOT NULL)
            THEN
                RAISE (ABORT, 'Il cliente non può essere rimosso, perché è utilizzato nella relazione Movimenti!')
            END;
       END;
```

A differenza dell'esempio che appare nella sezione precedente, l'attributo '**Cliente**' della relazione '**Movimenti**' può contenere il valore nullo ('**NULL**'). Per questa ragione, il grilletto verifica prima che il valore inserito non sia nullo, poi che il codice cliente esista nella relazione '**Clienti**'.

Una volta completato e salvato il file '`grilletti-movimenti-clienti`', se ne deve controllare il funzionamento con la base di dati, ma prima vanno rimossi i grilletti '**Movimenti_ins**' e '**Movimenti_upd**', che qui vengono ricreati. Basta eseguire i passaggi seguenti:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> DROP TRIGGER Articoli_ins;[Invio]

sqlite> DROP TRIGGER Articoli_upd;[Invio]

sqlite> .quit[Invio]
```

Quando i grilletti preesistenti, associati alla relazione '**Movimenti**', sono stati rimossi, si può eseguire il file '`grilletti-movimenti-clienti.sql`' nella base di dati:

```
$ sqlite3 mag.db < grilletti-movimenti-clienti.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la creazione dei grilletti dovrebbe essere avvenuta con successo, altrimenti, è stato commesso un errore. Per rimediare all'errore, si devono prima cancellare i grilletti (questa volta sono tre: '**Movimenti_ins**', '**Movimenti_upd**' e '**Clienti_del**'), quin-

di si può ritentare l'inserimento del comando (ammesso che il file ‘grilletti-movimenti-clienti.sql’ sia stato corretto di conseguenza).

Per verificare che i vincoli dichiarati funzionino come previsto, si può provare a inserire un movimento che fa riferimento a un cliente inesistente; quindi, si può provare a cancellare un articolo che risulta invece movimentato:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> INSERT INTO Movimenti[Invio]
   ...>     VALUES (11, 101, 2, '2012-01-25',[Invio]
   ...>     999, NULL, 1000, 500.00);[Invio]

  INSERT INTO Movimenti VALUES (11, 101, 2, '2012-01-25', ↩
 ↪999, NULL, 1000, 500.00);
  SQL error: Il codice cliente non è presente nella ↩
 ↪relazione Clienti!
sqlite> UPDATE Movimenti SET Cliente = 999 WHERE Movimento =
2;[Invio]

  UPDATE Movimenti SET Cliente = 999 WHERE Movimento = 2;
  SQL error: Il codice cliente non è presente nella ↩
 ↪relazione Clienti!
sqlite> DELETE FROM Clienti WHERE Cliente = 2;[Invio]

  DELETE FROM Clienti WHERE Cliente = 2;
  SQL error: Il cliente non può essere rimosso, perché ↩
 ↪è utilizzato nella relazione Movimenti!
```

`sqlite> .quit` [*Invio*]

79.11.3 Verifica sulla creazione dei grilletti «Movimenti_ins», «Movimenti_upd» e «Causali_del»

«

Si prepari il file ‘`grilletti-movimenti-causali.sql`’, modificando il file ‘`grilletti-movimenti-clienti.sql`’, in modo da riutilizzare quanto già scritto nei grilletti ‘**`Movimenti_ins`**’ e ‘**`Movimenti_upd`**’. Si segua lo scheletro seguente, tenendo conto che si vuole impedire l’inserimento nella relazione ‘**`Movimenti`**’ di causali inesistenti e che si vuole impedire la cancellazione di una causale, dalla relazione ‘**`Causali`**’, se questa risulta utilizzata nella relazione ‘**`Movimenti`**’ (in pratica, per questa funzione ulteriore, si deve aggiungere il grilletto ‘**`Causali_del`**’).

Figura 79.136. Scheletro del file ‘`grilletto-movimenti-causali.sql`’, da completare.

```
-- Creazione dei grilletti "Movimenti_ins", "Movimenti_upd" e "Causali_del"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: grilletti-movimenti-causali.sql

CREATE TRIGGER Movimenti_ins
       BEFORE INSERT ON Movimenti
       FOR EACH ROW
       BEGIN
            SELECT CASE
            WHEN (NEW.Quantita <= 0)
            THEN
                RAISE (ABORT, 'La quantità non può essere inferiore o uguale a zero!')
            WHEN (NEW.Valore < 0)
            THEN
                RAISE (ABORT, 'Il valore caricato non può essere inferiore a zero!')
            WHEN ((SELECT Articolo FROM Articoli WHERE Articolo = NEW.Articolo) IS NULL)
            THEN
                RAISE (ABORT, 'Il codice articolo non è presente nella relazione Articoli!')
            WHEN ((NEW.Cliente IS NOT NULL)
                  AND ((SELECT Cliente FROM Clienti WHERE Cliente = NEW.Cliente) IS NULL))
            THEN
```

```
                RAISE (ABORT, 'Il codice cliente non è presente nella relazione Clienti!')
            WHEN …
            THEN
                …
            END;
        END;

CREATE TRIGGER Movimenti_upd
        BEFORE UPDATE ON Movimenti
        FOR EACH ROW
        BEGIN
            SELECT CASE
            WHEN (NEW.Quantita <= 0)
            THEN
                RAISE (ABORT, 'La quantità non può essere inferiore o uguale a zero!')
            WHEN (NEW.Valore < 0)
            THEN
                RAISE (ABORT, 'Il valore caricato non può essere inferiore a zero!')
            WHEN ((SELECT Articolo FROM Articoli WHERE Articolo = NEW.Articolo) IS NULL)
            THEN
                RAISE (ABORT, 'Il codice articolo non è presente nella relazione Articoli!')
            WHEN ((NEW.Cliente IS NOT NULL)
                  AND ((SELECT Cliente FROM Clienti WHERE Cliente = NEW.Cliente) IS NULL))
            THEN
                RAISE (ABORT, 'Il codice cliente non è presente nella relazione Clienti!')
            WHEN …
            THEN
                …
            END;
        END;

CREATE TRIGGER Causali_del
        BEFORE DELETE ON Causali
        FOR EACH ROW
        BEGIN
            SELECT CASE
            WHEN …
            THEN
                …
            END;
        END;
```

Una volta completato e salvato il file '`grilletti-movimenti-causali`', se ne deve controllare il funzionamento con la base di dati, ma prima vanno rimossi i grilletti '**`Movimenti_ins`**' e '**`Movimenti_upd`**', che qui vengono ricreati. Basta eseguire i

passaggi seguenti:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> DROP TRIGGER Articoli_ins;[Invio]

sqlite> DROP TRIGGER Articoli_upd;[Invio]

sqlite> .quit[Invio]
```

Quando i grilletti preesistenti, associati alla relazione '**Movimenti**', sono stati rimossi, si può eseguire il file '`grilletti-movimenti-causali.sql`' nella base di dati:

```
$ sqlite3 mag.db < grilletti-movimenti-causali.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la creazione dei grilletti dovrebbe essere avvenuta con successo, altrimenti, è stato commesso un errore. Per rimediare all'errore, si devono prima cancellare i grilletti (tutti), quindi si può ritentare l'inserimento del comando (ammesso che il file '`grilletti-movimenti-causali.sql`' sia stato corretto di conseguenza).

Si consegni per la valutazione la stampa del file '`grilletti-movimenti-causali.sql`'.

79.11.4 Verifica sulla creazione dei grilletti «Movimenti_ins», «Movimenti_upd» e «Fornitori_del»

«

Si prepari il file '`grilletti-movimenti-fornitori.sql`', modificando il file '`grilletti-movimenti-causali.sql`', in modo da riutilizzare quanto già scritto nei grilletti '**Movimenti_ins**'

e '**Movimenti_upd**'. Si segua lo scheletro seguente, tenendo conto che si vuole impedire l'inserimento nella relazione '**Movimenti**' di fornitori inesistenti e che si vuole impedire la cancellazione di un fornitore, dalla relazione '**Fornitori**', se questo risulta utilizzato nella relazione '**Movimenti**' (in pratica, per questa funzione ulteriore, si deve aggiungere il grilletto '**Fornitori_del**').

Si osservi che nella relazione '**Movimenti**', l'attributo '**Fornitore**' può avere un valore nullo.

Figura 79.138. Scheletro del file '`grilletto-movimenti-fornitori.sql`', da completare.

```
-- Creazione dei grilletti "Movimenti_ins", "Movimenti_upd" e "Fornitori_del"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: grilletti-movimenti-fornitori.sql

CREATE TRIGGER Movimenti_ins
       BEFORE INSERT ON Movimenti
       FOR EACH ROW
       BEGIN
            SELECT CASE
            WHEN (NEW.Quantita <= 0)
            THEN
                RAISE (ABORT, 'La quantità non può essere inferiore o uguale a zero!')
            WHEN (NEW.Valore < 0)
            THEN
                RAISE (ABORT, 'Il valore caricato non può essere inferiore a zero!')
            WHEN ((SELECT Articolo FROM Articoli WHERE Articolo = NEW.Articolo) IS NULL)
            THEN
                RAISE (ABORT, 'Il codice articolo non è presente nella relazione Articoli!')
            WHEN ((NEW.Cliente IS NOT NULL)
                  AND ((SELECT Cliente FROM Clienti WHERE Cliente = NEW.Cliente) IS NULL))
            THEN
                RAISE (ABORT, 'Il codice cliente non è presente nella relazione Clienti!')
            WHEN …
            THEN
                …
            WHEN …
                  AND …
            THEN
                …
            END;
```

```
        END;

CREATE TRIGGER Movimenti_upd
        BEFORE UPDATE ON Movimenti
        FOR EACH ROW
        BEGIN
             SELECT CASE
             WHEN (NEW.Quantita <= 0)
             THEN
                 RAISE (ABORT, 'La quantità non può essere inferiore o uguale a zero!')
             WHEN (NEW.Valore < 0)
             THEN
                 RAISE (ABORT, 'Il valore caricato non può essere inferiore a zero!')
             WHEN ((SELECT Articolo FROM Articoli WHERE Articolo = NEW.Articolo) IS NULL)
             THEN
                 RAISE (ABORT, 'Il codice articolo non è presente nella relazione Articoli!')
             WHEN ((NEW.Cliente IS NOT NULL)
                   AND ((SELECT Cliente FROM Clienti WHERE Cliente = NEW.Cliente) IS NULL))
             THEN
                 RAISE (ABORT, 'Il codice cliente non è presente nella relazione Clienti!')
             WHEN …
             THEN
                 …
             WHEN …
                   AND …
             THEN
                 …
             END;
        END;

CREATE TRIGGER Fornitori_del
        BEFORE DELETE ON Fornitori
        FOR EACH ROW
        BEGIN
             SELECT CASE
             WHEN …
             THEN
                 …
             END;
        END;
```

Una volta completato e salvato il file '`grilletti-movimenti-fornitori`', se ne deve controllare il funzionamento con la base di dati, ma prima vanno rimossi i grilletti '**`Movimenti_ins`**' e '**`Movimenti_upd`**', che qui vengono ricreati. Basta eseguire i

passaggi seguenti:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> DROP TRIGGER Articoli_ins;[Invio]

sqlite> DROP TRIGGER Articoli_upd;[Invio]

sqlite> .quit[Invio]
```

Quando i grilletti preesistenti, associati alla relazione '**Movimenti**', sono stati rimossi, si può eseguire il file '`grilletti-movimenti-fornitori.sql`' nella base di dati:

```
$ sqlite3 mag.db < grilletti-movimenti-fornitori.sql[Invio]
```

Se non si ottiene alcun messaggio da parte del programma, la creazione dei grilletti dovrebbe essere avvenuta con successo, altrimenti, è stato commesso un errore. Per rimediare all'errore, si devono prima cancellare i grilletti (tutti), quindi si può ritentare l'inserimento del comando (ammesso che il file '`grilletti-movimenti-fornitori.sql`' sia stato corretto di conseguenza).

Si consegni per la valutazione la stampa del file '`grilletti-movimenti-fornitori.sql`'.

79.11.5 Conclusione

Prima di passare alla sezione successiva, si deve riprendere il file '`magazzino.sql`' e vi si devono sostituire le istruzioni per la creazione dei grilletti '**Movimenti_ins**' e '**Movimenti_upd**', come contenuto nel file '`grilletti-movimenti-fornitori.sql`';

inoltre vanno aggiunti i grilletti '**Articoli_del**', '**Causali_del**', '**Clienti_del**' e '**Fornitori_del**', come sono stati realizzati in questa sezione.

Si osservi che la dichiarazione dei grilletti va collocata dopo la creazione della relazione a cui fanno riferimento e prima delle istruzioni che inseriscono delle tuple nella stessa relazione.

Una volta aggiornato il file '`magazzino.sql`' come descritto, si deve cancellare il file '`mag.db`' e ricreare a partire dalle istruzioni contenute nel file '`magazzino.sql`':

```
$ sqlite3 mag.db < magazzino.sql [Invio]
```

Se vengono segnalati degli errori, occorre correggere il file '`magazzino.sql`', cancellare nuovamente il file '`mag.db`', quindi si deve ripetere l'operazione. La base di dati contenuta nel file '`mag.db`', viene usata ancora e non si può proseguire se non si riesce a ricrearla correttamente.

79.12 Selezione di attributi virtuali, ottenuti da un'espressione

«

Il linguaggio SQL consente di costruire delle espressioni elementari, attraverso operatori matematici e funzioni comuni; l'interrogazione di una relazione può essere realizzata anche attraverso l'uso di espressioni.

Tabella 79.140. Operatori aritmetici comuni.

Operatore e operandi	Descrizione
−op	Inverte il segno dell'operando.
op1 `+` ***op2***	Somma i due operandi.
op1 `-` ***op2***	Sottrae dal primo il secondo operando.
op1 `*` ***op2***	Moltiplica i due operandi.
op1 `/` ***op2***	Divide il primo operando per il secondo.
op1 `%` ***op2***	Modulo: il resto della divisione tra il primo e il secondo operando.

Tabella 79.141. Alcune funzioni riconosciute da SQLite.

Funzione	Descrizione
`ABS(`***n***`)`	Restituisce il valore assoluto di ***n***
`LENGTH(`***stringa***`)`	Restituisce la lunghezza in caratteri della stringa indicata come argomento.
`LOWER(`***stringa***`)` `UPPER(`***stringa***`)`	La prima funzione restituisce la stringa indicata come argomento, con lettere minuscole; la seconda con lettere maiuscole.
`MIN(`***x***`,`***y*** `[,…]` `)` `MAX(`***x***`,`***y*** `[,…]` `)`	La prima funzione restituisce il valore minimo tra quelli indicati come argomento; la seconda, invece, restituisce il valore massimo.

Funzione	Descrizione
`ROUND(`***n*** `[,` ***n*** `])`	Restituisce il valore di ***n*** arrotondato a ***m*** decimali. Se ***m*** viene omesso, si intende pari a zero.
`SUBSTR(`***stringa*** `,` ***n*** `,` ***m*** `)`	Estrae la stringa che inizia dalla posizione ***n***, lunga ***m*** caratteri.

79.12.1 Interrogazione della relazione «Movimenti» in modo da ottenere il valore unitario

«

Nella relazione '**`Movimenti`**' appare un attributo denominato '**`Valore`**'. Si tratta del valore dell'articolo, determinato in base al **costo di acquisto** (da non confondere con il prezzo di listino), con il quale si determina il valore delle merci in magazzino. Pre ogni tupla della relazione, si vuole ottenere il valore unitario, che si calcola dividendo il valore per la quantità movimentata corrispondente.

Si crei il file '`prova-interrogazione-movimenti-vu.sql`', contenente il testo seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Interrogazione della relazione "Movimenti"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: prova-interrogazione-movimenti-vu.sql

.mode columns
.headers on

SELECT Articolo,
       Causale,
       Data,
       Quantita,
```

```
        (Valore/Quantita) AS ValoreUnitario
        FROM Movimenti;
```

Si osservi che, nell'ultima colonna del listato che si vuole ottenere, viene indicata l'espressione '**`(Valore/Quantita)`**', associata a un alias, in modo da mostrare una descrizione appropriata.

Una volta completato e salvato il file '`prova-interrogazione-movimenti-vu.sql`', se ne deve controllare il funzionamento con la base di dati:

```
$ sqlite3 mag.db < prova-interrogazione-movimenti-vu.sql [Invio]
```

Si dovrebbe ottenere un listato simile a quello seguente:

```
Articolo    Causale     Data        Quantita    ValoreUnitario
----------  ----------  ----------  ----------  --------------
2           1           2012-01-15  10000       0.01
2           2           2012-01-16  1000        0.01
102         1           2012-01-17  1000        0.2
102         2           2012-01-18  100         0.2
401         1           2012-01-19  1000        0.2
401         2           2012-01-20  200         0.2
401         4           2012-01-20  100         0.2
102         4           2012-01-20  100         0.2
601         1           2012-01-21  2000        0.5
601         2           2012-01-25  1000        0.5
```

Se invece si ottengono degli errori, dovrebbe essere sufficiente correggere il file '`prova-interrogazione-movimenti-vu.sql`' e poi riprovare.

79.12.2 Vista della relazione «Movimenti» in modo da ottenere il valore unitario

«

Così come è possibile scrivere un'interrogazione a una relazione indicando delle espressioni, se ne può realizzare una vista, così da semplificare gli accessi a queste informazioni generate attraverso dei calcoli.

Si crei il file '`vista-movimenti-extra.sql`', contenente il testo seguente, sostituendo le metavariabili con informazioni appropriate e rispettando la punteggiatura:

```
-- Vista "MovimentiExtra"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: vista-movimenti-extra.sql

CREATE VIEW MovimentiExtra AS
    SELECT Movimento,
           Articolo,
           Causale,
           Data,
           Cliente,
           Fornitore,
           Quantita,
           Valore,
           (Valore/Quantita) AS ValoreUnitario
           FROM Movimenti;
```

La vista '**`MovimentiExtra`**' che si ottiene in questo modo, include tutti gli attributi della relazione '**`Movimenti`**', aggiungendo l'attributo virtuale '**`ValoreUnitario`**', ottenuto dividendo il valore complessivo per la quantità movimentata.

Una volta completato e salvato il file ‘`vista-movimenti-extra.sql`’, se ne deve controllare il funzionamento con la base di dati:

```
$ sqlite3 mag.db < vista-movimenti-extra.sql[Invio]
```

Se non vengono generati dei messaggi, l’operazione dovrebbe essere stata completata con successo, altrimenti, se la vista è stata creata, ma in modo errato, è necessario eliminarla, quindi si può correggere il file ‘`vista-movimenti-extra.sql`’ e riprovare. Per eliminare la vista creata in modo errato, si può utilizzare il programma ‘**`sqlite3`**’ in modo interattivo, come già mostrato in altri capitoli (istruzione ‘**`DROP VIEW`**’).

Quando si ritiene di avere creato la vista in modo corretto, è bene verificare di avere ottenuto il risultato desiderato:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode columns[Invio]

sqlite> SELECT * FROM MovimentiExtra[Invio]
```

```
Movimento Articolo Causale Data       Cliente Fornitore Quantita Valore ValoreUnitario
--------- -------- ------- ---------- ------- --------- -------- ------ --------------
1         2        1       2012-01-15         3         10000    100    0.01
2         2        2       2012-01-16 2                 1000     10     0.01
3         102      1       2012-01-17         2         1000     200    0.2
4         102      2       2012-01-18 1                 100      20     0.2
5         401      1       2012-01-19         1         1000     200    0.2
6         401      2       2012-01-20 3                 200      20     0.2
7         401      4       2012-01-20         1         100      20     0.2
8         102      4       2012-01-20         2         100      20     0.2
9         601      1       2012-01-21         3         2000     1000   0.5
10        601      2       2012-01-25 1                 1000     500    0.5
```

```
sqlite> .quit[Invio]
```

79.12.3 Verifica sulla creazione della vista «MovimentiExtra»

In questa verifica si deve riprendere il file ‘`vista-movimenti-extra.sql`’, per modificarlo, in modo da aggiungere un attributo virtuale ulteriore, contenente la quantità in forma algebrica: valori positivi per i carichi e valori negativi per gli scarichi. Dal momento che l’informazione se trattasi di carico o scarico è contenuta nella relazione ‘**`Causali`**’, anche questa va utilizzata nella costruzione della vista.

Si modifichi il file ‘`vista-movimenti-extra.sql`’, seguendo lo scheletro che viene proposto, per far sì che la vista ‘**`MovimentiExtra`**’ contenga gli attributi seguenti:

1. ‘**`Movimento`**’, corrispondente al numero di sequenza assegnato a ogni movimento nella relazione ‘**`Movimenti`**’;
2. ‘**`Articolo`**’, corrispondente al codice articolo della relazione ‘**`Movimenti`**’;
3. ‘**`Causale`**’, corrispondente al codice causale della relazione ‘**`Movimenti`**’;

4. '**Data**', corrispondente alla data del movimento nella relazione '**Movimenti**';
5. '**Cliente**', corrispondente al codice cliente della relazione '**Movimenti**';
6. '**Fornitore**', corrispondente al codice fornitore della relazione '**Movimenti**';
7. '**Quantità**', corrispondente alla quantità movimentata nella relazione '**Movimenti**';
8. '**Valore**', corrispondente al valore del movimento, nella relazione '**Movimenti**';
9. '**ValoreUnitario**', corrispondente al valore unitario del movimento, ottenuto dividendo il valore complessivo per la quantità (dalla relazione '**Movimenti**');
10. '**QuantitaAlgebrica**', corrispondente alla quantità movimentata, con segno, ottenuta moltiplicando l'attributo '**Variazione**' della relazione '**Causali**' all'attributo '**Quantita**' della relazione '**Movimenti**'.

Figura 79.147. Scheletro del file ‘`vista-movimenti-extra.sql`’, da completare.

```
-- Vista "MovimentiExtra"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: vista-movimenti-extra.sql

CREATE VIEW MovimentiExtra AS
    SELECT Movimenti.Movimento          AS Movimento,
           Movimenti.Articolo           AS Articolo,
           Movimenti.Causale            AS Causale,
           Movimenti.Data               AS Data,
           Movimenti.Cliente            AS Cliente,
           Movimenti.Fornitore          AS Fornitore,
           Movimenti.Quantita           AS Quantita,
           Movimenti.Valore             AS Valore,
           (Movimenti.Valore/Movimenti.Quantita)
                                        AS ValoreUnitario,
           …
           FROM …
           WHERE Movimenti.Causale = Causali.causale;
```

Prima di poter eseguire questo file con la base di dati, occorre eliminare la vista ‘**MovimentiExtra**’, che già dovrebbe esistere. Si ricorda che per eliminare una vista si utilizza l’istruzione ‘**DROP VIEW**’ e che conviene intervenire con il programma ‘**sqlite3**’ in modo interattivo.

Per eseguire il file ‘`vista-movimenti-extra.sql`’, si agisce come sempre:

```
$ sqlite3 mag.db < vista-movimenti-extra.sql [Invio]
```

Se la creazione della vista produce degli errori, occorre elimi-

nare nuovamente la vista e, dopo la correzione del file ‘`vista-movimenti-extra.sql`’, si può ritentare.

Quando si è consapevoli di avere creato correttamente la vista ‘**`MovimentiExtra`**’, la si può interrogare come se fosse una relazione normale:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM MovimentiExtra;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Movimento Articolo Causale Data        Cliente Fornitore Quantita Valore ValoreUnitario QuantitaAlgebrica
--------- -------- ------- ----------  ------- --------- -------- ------ -------------- -----------------
1         2        1       2012-01-15          3         10000    100    0.01           10000
2         2        2       2012-01-16  2                 1000     10     0.01           -1000
3         102      1       2012-01-17          2         1000     200    0.2            1000
4         102      2       2012-01-18  1                 100      20     0.2            -100
5         401      1       2012-01-19          1         1000     200    0.2            1000
6         401      2       2012-01-20  3                 200      20     0.2            -200
7         401      4       2012-01-20          1         100      20     0.2            -100
8         102      4       2012-01-20          2         100      20     0.2            -100
9         601      1       2012-01-21          3         2000     1000   0.5            2000
10        601      2       2012-01-25  1                 1000     500    0.5            -1000
```

Se tutto funziona regolarmente, si consegni per la valutazione la stampa del file ‘`vista-movimenti-extra.sql`’.

79.12.4 Conclusione

«

Prima di passare alla sezione successiva, si deve riprendere il file ‘`magazzino.sql`’ e vi si deve aggiungere l’istruzione per la creazione della vista ‘**`MovimentiExtra`**’, come realizzato nella verifica appena conclusa.

Una volta aggiornato il file ‘`magazzino.sql`’ come descritto, si deve cancellare il file ‘`mag.db`’ e ricreare a partire dalle istruzioni contenute nel file ‘`magazzino.sql`’:

```
$ sqlite3 mag.db < magazzino.sql[Invio]
```

Se vengono segnalati degli errori, occorre correggere il file ‘`magazzino.sql`’, cancellare nuovamente il file ‘`mag.db`’, quindi si deve ripetere l’operazione. La base di dati contenuta nel file ‘`mag.db`’, viene usata nella sezione successiva e non si può proseguire se non si riesce a ricrearla correttamente.

79.13 Aggregazioni

«

L’aggregazione è una forma di interrogazione attraverso cui si ottengono risultati riepilogativi del contenuto di una relazione, nel suo complesso o a gruppi di tuple. Per questo si utilizzano delle funzioni speciali al posto dell’espressione che esprime gli attributi del risultato. Queste funzioni restituiscono un solo valore e come tali concorrono a creare un’unica tupla.

Tabella 79.150. Alcune funzioni aggreganti riconosciute da SQLite.

Funzione	Descrizione
`COUNT(`***x***`)`	Restituisce il numero di tuple, nel gruppo, per le quali l'espressione ***x*** restituisce un valore diverso da '**`NULL`**'.
`COUNT(*)`	Restituisce il numero di tuple esistenti nel gruppo.
`AVG(`***x***`)`	Restituisce la media, nel gruppo di tuple, dei valori che ottiene l'espressione ***x***, escludendo '**`NULL`**' e considerando i valori non numerici pari a zero.
`MIN(`***x***`)` `MAX(`***x***`)`	Restituisce il valore minimo o massimo, nel gruppo di tuple, dei valori che ottiene l'espressione ***x***, escludendo '**`NULL`**' e considerando i valori non numerici pari a zero.
`SUM(`***x***`)`	Restituisce la somma, nel gruppo di tuple, dei valori che ottiene l'espressione ***x***, escludendo '**`NULL`**' e considerando i valori non numerici pari a zero.

La forma che assume l'istruzione '**`SELECT`**' quando si usano le aggregazioni e tipicamente quella seguente:

```
SELECT specificazione_dell'attributo_1 [ , ... specificazione_dell'attributo_n ]
    FROM specificazione_della_relazione_1 [ , ... specificazione_della_relazione_n ]
    [ WHERE condizione ]
    [ GROUP BY attributo_1 [ , ... ] ]
```

In pratica, le funzioni aggreganti vanno usate nell'elenco che descri-

ve gli attributi. Se non si usa l'opzione '**GROUP BY**', il gruppo di tuple di riferimento comprende tutte le tuple della relazione o della congiunzione (di relazioni). Se si specifica l'opzione '**GROUP BY**', le tuple vengono raggruppate in base all'uguaglianza degli attributi indicati come argomento di tale opzione. In pratica:

1. la relazione ottenuta dall'istruzione '**SELECT**...**FROM**' viene filtrata dalla condizione '**WHERE**';
2. la relazione risultante viene riordinata in modo da raggruppare le tuple in cui i contenuti degli attributi elencati dopo l'opzione '**GROUP BY**' sono uguali;
3. su questi gruppi di tuple vengono valutate le funzioni di aggregazione.

79.13.1 Aggregazioni banali

«

Per prendere un po' di dimestichezza con le aggregazioni, conviene usare il programma '**sqlite3**' in modo interattivo e fare qualche piccolo esperimento:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]
```

Si vogliono contare le tuple della relazione '**Movimenti**':

```
sqlite> SELECT COUNT(*) FROM Movimenti;[Invio]
```

```
10
```

Si vogliono contare i movimenti per ogni tipo di articolo:

```
sqlite> SELECT Articolo, COUNT(*) FROM Movimenti GROUP BY
Articolo; [Invio]

Articolo    COUNT(*)
----------  ----------
2           2
102         3
401         3
601         2
```

Si vuole conoscere la quantità esistente di ogni articolo (si usa la vista ***MovimentiExtra***, che offre l'attributo '**`QuantitaAlgebrica`**'):

```
sqlite> SELECT Articolo, SUM(QuantitaAlgebrica) [Invio]
   ...>     FROM MovimentiExtra GROUP BY Articolo; [Invio]

Articolo    SUM(QuantitaAlgebrica)
----------  ----------------------
2           9000
102         800
401         700
601         1000
```

Si vuole conoscere la quantità esistente di ogni articolo in magazzino e il valore (il valore viene calcolato a partire da quello medio, moltiplicato per la quantità algebrica):

```
sqlite> SELECT Articolo, SUM(QuantitaAlgebrica), [Invio]
    ...>       SUM(QuantitaAlgebrica*ValoreUnitario) [Invio]
    ...>     FROM MovimentiExtra GROUP BY Articolo; [Invio]
```

```
Articolo    SUM(QuantitaAlgebrica)  SUM(QuantitaAlgebrica*ValoreUnitario)
----------  ----------------------  -------------------------------------
2           9000                    90
102         800                     160
401         700                     140
601         1000                    500
```

Si vuole conoscere la quantità esistente di ogni articolo in magazzino e il costo medio, determinato dividendo il valore complessivo per la quantità esistente:

```
sqlite> SELECT Articolo, [Invio]

sqlite>        SUM(QuantitaAlgebrica), [Invio]

sqlite>        SUM(QuantitaAlgebrica*ValoreUnitario)/↩
↪SUM(QuantitaAlgebrica) [Invio]

sqlite>        FROM MovimentiExtra GROUP BY Articolo; [Invio]
```

```
Articolo    SUM(QuantitaAlgebrica)  SUM(QuantitaAlgebrica*ValoreUnitario)/SUM(QuantitaAlgebrica)
----------  ----------------------  ------------------------------------------------------------
2           9000                    0.01
102         800                     0.2
401         700                     0.2
601         1000                    0.5
```

```
sqlite> .quit [Invio]
```

79.13.2 Verifica sulla creazione della vista «SituazioneMagazzino»

«

Si vuole realizzare la vista '**SituazioneMagazzino**' che, in questa verifica, si limiti a mostrare poche informazioni riepilogative sullo stato del magazzino.

Si realizzi il file '`vista-situazione-magazzino-1.sql`', seguendo lo scheletro che viene proposto, per far sì che la vista

‘**SituazioneMagazzino**’ contenga gli attributi seguenti:

1. ‘**Codice**’, corrispondente al codice articolo della relazione ‘**Movimenti**’ o della vista ‘**MovimentiExtra**’;
2. ‘**Articolo**’, corrispondente alla descrizione dell’articolo, come indicato nella relazione ‘**Articoli**’;
3. ‘**Esistenza**’, corrispondente alla somma algebrica dei carichi, come si ottiene dalla vista ‘**MovimentiExtra**’.

Figura 79.157. Scheletro del file ‘`vista-situazione-magazzino-1.sql`’, da completare.

```
-- Vista "SituazioneMagazzino"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: vista-situazione-magazzino-1.sql

CREATE VIEW SituazioneMagazzino AS
    SELECT MovimentiExtra.Articolo        AS Codice,
           …
           …
           FROM …
           WHERE MovimentiExtra.Articolo = Articoli.Articolo
           GROUP BY MovimentiExtra.Articolo;
```

Per eseguire il file ‘`vista-situazione-magazzino-1.sql`’, si agisce come sempre:

```
$ sqlite3 mag.db < vista-situazione-magazzino-1.sql[Invio]
```

Se la creazione della vista produce degli errori, occorre eliminare la vista e, dopo la correzione del file ‘`vista-situazione-magazzino-1.sql`’, si può ritentare.

Quando si è consapevoli di avere creato correttamente la vista '**SituazioneMagazzino**', la si può interrogare come se fosse una relazione normale:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM SituazioneMagazzino;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Codice      Articolo                                 Esistenza
----------  ---------------------------------------  ----------
2           Dischetti da 9 cm 1440 Kibyte colorati   9000
102         CD-R 52x                                 800
401         DVD+R 8x                                 700
601         DVD+RW 8x                                1000
```

Se tutto funziona regolarmente, si consegni per la valutazione la stampa del file '`vista-situazione-magazzino-1.sql`'.

79.13.3 Verifica sulla creazione della vista «SituazioneMagazzino»

«

Si vuole estendere la vista '**SituazioneMagazzino**', già realizzata in parte nella verifica precedente; pertanto, in questa verifica si modifica il file '`vista-situazione-magazzino-1.sql`' Salvandolo con il nome '`vista-situazione-magazzino-2`'. Si vogliono ottenere gli attributi seguenti:

1. '**Codice**', corrispondente al codice articolo della relazione '**Movimenti**' o della vista '**MovimentiExtra**';
2. '**Articolo**', corrispondente alla descrizione dell'articolo, come indicato nella relazione '**Articoli**';
3. '**ScortaMin**', corrispondente alla scorta minima, come contenuto nella relazione '**Articoli**';
4. '**Esistenza**', corrispondente alla somma algebrica dei carichi, come si ottiene dalla vista '**MovimentiExtra**';
5. '**Valore**', corrispondente al valore complessivo di ogni articolo (come mostrato negli esempi prima di queste verifiche).

Figura 79.160. Scheletro del file '`vista-situazione-magazzino-2.sql`', da completare.

```
-- Vista "SituazioneMagazzino"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: vista-situazione-magazzino-2.sql

CREATE VIEW SituazioneMagazzino AS
    SELECT MovimentiExtra.Articolo            AS Codice,
           …
           …
           FROM …
           WHERE MovimentiExtra.Articolo = Articoli.Articolo
           GROUP BY MovimentiExtra.Articolo;
```

Prima di poter eseguire questo file con la base di dati, occorre eliminare la vista '**SituazioneMagazzino**', che già dovrebbe esistere. Si ricorda che per eliminare una vista si utilizza l'istruzione '**DROP VIEW**' e che conviene intervenire con il programma '**sqlite3**' in modo interattivo.

Per eseguire il file ‘`vista-situazione-magazzino-2.sql`’, si agisce come sempre:

```
$ sqlite3 mag.db < vista-situazione-magazzino-2.sql[Invio]
```

Se la creazione della vista produce degli errori, occorre eliminare la vista e, dopo la correzione del file ‘`vista-situazione-magazzino-2.sql`’, si può ritentare.

Quando si è consapevoli di avere creato correttamente la vista ‘**`SituazioneMagazzino`**’, la si può interrogare come se fosse una relazione normale:

```
$ sqlite3 mag.db[Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on[Invio]

sqlite> .mode column[Invio]

sqlite> SELECT * FROM SituazioneMagazzino;[Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Codice      Articolo                                ScortaMin   Esistenza   Valore
----------  --------------------------------------  ----------  ----------  ----------
2           Dischetti da 9 cm 1440 Kibyte colorati  500         9000        90
102         CD-R 52x                                500         800         160
401         DVD+R 8x                                200         700         140
601         DVD+RW 8x                               200         1000        500
```

Se tutto funziona regolarmente, si consegni per la valutazione la stampa del file ‘`vista-situazione-magazzino-2.sql`’.

79.13.4 Verifica sulla creazione della vista «SituazioneMagazzino»

Si vuole estendere la vista '**SituazioneMagazzino**', già realizzata in parte nella verifica precedente, in modo ottenere anche il costo medio; pertanto, in questa verifica si modifica il file '`vista-situazione-magazzino-2.sql`' salvandolo con il nome '`vista-situazione-magazzino-3.sql`'. Si vogliono ottenere gli attributi seguenti:

1. '**Codice**', corrispondente al codice articolo della relazione '**Movimenti**' o della vista '**MovimentiExtra**';
2. '**Articolo**', corrispondente alla descrizione dell'articolo, come indicato nella relazione '**Articoli**';
3. '**ScortaMin**', corrispondente alla scorta minima, come contenuto nella relazione '**Articoli**';
4. '**Esistenza**', corrispondente alla somma algebrica dei carichi, come si ottiene dalla vista '**MovimentiExtra**';
5. '**Valore**', corrispondente al valore complessivo di ogni articolo (come mostrato negli esempi prima di queste verifiche);
6. '**CostoMedio**', corrispondente al valore complessivo di ogni articolo, diviso la quantità esistente (come mostrato negli esempi prima di queste verifiche).

Figura 79.163. Scheletro del file ‘`vista-situazione-magazzino-3.sql`’, da completare.

```
-- Vista "SituazioneMagazzino"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: vista-situazione-magazzino-3.sql

CREATE VIEW SituazioneMagazzino AS
    SELECT MovimentiExtra.Articolo         AS Codice,
           …
           …
           FROM …
           WHERE MovimentiExtra.Articolo = Articoli.Articolo
           GROUP BY MovimentiExtra.Articolo;
```

Prima di poter eseguire questo file con la base di dati, occorre eliminare la vista ‘**SituazioneMagazzino**’, che già dovrebbe esistere. Si ricorda che per eliminare una vista si utilizza l’istruzione ‘**DROP VIEW**’ e che conviene intervenire con il programma ‘**sqlite3**’ in modo interattivo.

Per eseguire il file ‘`vista-situazione-magazzino-3.sql`’, si agisce come sempre:

```
$ sqlite3 mag.db < vista-situazione-magazzino-3.sql[Invio]
```

Se la creazione della vista produce degli errori, occorre eliminare la vista e, dopo la correzione del file ‘`vista-situazione-magazzino-3.sql`’, si può ritentare.

Quando si è consapevoli di avere creato correttamente la vista ‘**SituazioneMagazzino**’, la si può interrogare come se fosse una relazione normale:

```
$ sqlite3 mag.db [Invio]

  SQLite version …
  Enter ".help" for instructions
sqlite> .headers on [Invio]

sqlite> .mode column [Invio]

sqlite> SELECT * FROM SituazioneMagazzino; [Invio]
```

Si dovrebbe ottenere il listato seguente:

```
Codice Articolo                                ScortaMin Esistenza Valore CostoMedio
------ --------------------------------------  --------- --------- ------ ----------
2      Dischetti da 9 cm 1440 Kibyte colorati  500       9000      90     0.01
102    CD-R 52x                                500       800       160    0.2
401    DVD+R 8x                                200       700       140    0.2
601    DVD+RW 8x                               200       1000      500    0.5
```

Se tutto funziona regolarmente, si consegni per la valutazione la stampa del file ‘`vista-situazione-magazzino-3.sql`’.

79.13.5 Conclusione

Prima di passare alla sezione successiva, si deve riprendere il file ‘`magazzino.sql`’ e vi si deve aggiungere l’istruzione per la creazione della vista ‘**`SituazioneMagazzino`**’, come realizzato nel’ultima verifica appena conclusa.

Una volta aggiornato il file ‘`magazzino.sql`’ come descritto, si deve cancellare il file ‘`mag.db`’ e ricreare a partire dalle istruzioni contenute nel file ‘`magazzino.sql`’:

```
$ sqlite3 mag.db < magazzino.sql [Invio]
```

Se vengono segnalati degli errori, occorre correggere il file ‘`magazzino.sql`’, cancellare nuovamente il file ‘`mag.db`’, quindi

si deve ripetere l'operazione. La base di dati contenuta nel file '`mag.db`', viene usata ancora e non si può proseguire se non si riesce a ricrearla correttamente.

79.14 Inserimento automatico del costo medio

A conclusione di queste lezioni sul linguaggio SQL, viene mostrata la soluzione di un problema, senza richiedere altre verifiche.

Quando si inserisce un movimento nella relazione '**`Movimenti`**', l'utente deve indicare il valore del movimento. Si determina facilmente questo valore quando il bene viene acquistato, in quanto corrisponde al costo complessivo (IVA esclusa). Quando l'articolo viene scaricato per perché reso al fornitore, il valore deve essere lo stesso della fattura a cui si riferisce (in proporzione alla quantità resa), ma quando viene scaricato per la vendita, occorre decidere come attribuire questo valore.

Il modo più semplice per definire il valore del bene che viene scaricato per la vendita, o comunque per scopi diversi dal reso, è quello di calcolare il costo medio ponderato per movimento. In pratica, si tratterebbe di consultare la vista '**`SituazioneMagazzino`**', prima di procedere all'inserimento, in modo da conoscere il costo medio unitario, ottenuto in base ai movimenti esistenti.

Quello che si vuol fare qui è di costruire un grilletto che inserisca automaticamente il valore, determinandolo in base al costo medio ponderato per movimento, quando si inserisce un movimento e si omette l'indicazione del valore stesso.

79.14.1 Vista «CostoMedioValido»

«

La vista '**SituazioneMagazzino**' calcola il costo medio tenendo conto di tutte le tuple, anche quelle che contengono un valore indeterminato del movimento ('**NULL**'). Per lo scopo che si vuole raggiungere, è necessario calcolare il costo medio escludendo i valori indeterminati; pertanto, si realizza una vista apposita:

```
-- Vista "CostoMedioValido"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: vista-costo-medio-valido.sql

CREATE VIEW CostoMedioValido AS
    SELECT Articolo,
           SUM(QuantitaAlgebrica*ValoreUnitario)        AS Valore,
           (SUM(QuantitaAlgebrica*ValoreUnitario)/SUM(QuantitaAlgebrica))
                                                        AS CostoMedio
           FROM MovimentiExtra
           WHERE Valore NOT NULL
           GROUP BY Articolo;
```

79.14.2 Grilletto «ValorizzazioneScarichi»

«

Si può creare un grilletto, che aggiorni automaticamente tutte le tuple della relazione '**Movimenti**', che hanno un valore movimentato indeterminato, traendo il costo medio dalla vista '**CostoMedioValido**':

```
-- Grilletto "ValorizzazioneScarichi"
-- Esercizio di: cognome nome classe
-- Data: data
-- File: grilletto-valorizzazione-scarichi.sql

CREATE TRIGGER ValorizzazioneScarichi
       AFTER INSERT ON Movimenti
       BEGIN
```

```
            UPDATE Movimenti
                   SET Valore =
                       (SELECT CostoMedio * NEW.Quantita FROM CostoMedioValido
                        WHERE Articolo = NEW.articolo)
                   WHERE Valore IS NULL;
        END;
```

Naturalmente, i movimenti che vengono presi in considerazione dal grilletto, sono solo quelli che vengono inseriti **dopo** la sua creazione.

Si osservi, comunque, che occorre anche impedire la sostituzione del valore con qualcosa di indeterminato. In pratica, occorre estendere il grilletto associato alla modifica delle tuple della relazione '**`Movimenti`**', in modo da non accettare valori indeterminati per l'attributo del valore:

```
CREATE TRIGGER Movimenti_upd
       BEFORE UPDATE ON Movimenti
       FOR EACH ROW
       BEGIN
            SELECT CASE
            …
            …
            …
            WHEN (NEW.Valore IS NULL)
            THEN
                RAISE (ABORT, 'In fase di variazione, il valore non può essere indeterminato!')
            END;
       END;
```

Parte vi

Corso basilare di programmazione

«

Introduzione 2267

Programma didattico 2267

Strumenti per la compilazione 2271

80 Dai sistemi di numerazione all'organizzazione della memoria 2279

80.1 Sistemi di numerazione 2281

80.2 Conversioni numeriche di valori interi 2287

80.3 Conversioni numeriche di valori non interi 2299

80.4 Operazioni elementari e sistema di rappresentazione binaria dei valori 2308

80.5 Calcoli con i valori binari rappresentati nella forma usata negli elaboratori 2323

80.6 Scorrimenti, rotazioni, operazioni logiche 2339

80.7 Organizzazione della memoria 2347

80.8 Riferimenti 2365

80.9 Soluzioni agli esercizi proposti 2365

81 Nozioni minime sul linguaggio C 2373

81.1 Primo approccio al linguaggio C 2375

81.2 Variabili e tipi del linguaggio C 2383

81.3 Operatori ed espressioni del linguaggio C 2402

81.4 Strutture di controllo di flusso del linguaggio C 2423
81.5 Funzioni del linguaggio C 2441
81.6 Riferimenti 2453
81.7 Soluzioni agli esercizi proposti 2453
82 Puntatori, array e stringhe in C 2475
82.1 Espressioni a cui si assegnano dei valori 2477
82.2 Puntatori 2478
82.3 Dichiarazione di una variabile puntatore 2478
82.4 Dereferenziazione 2479
82.5 «Little endian» e «big endian» 2482
82.6 Chiamata di funzione con puntatori 2487
82.7 Array 2490
82.8 Array a una dimensione 2490
82.9 Array multidimensionali 2495
82.10 Natura dell'array 2500
82.11 Array e funzioni 2506
82.12 Aritmetica dei puntatori 2507
82.13 Stringhe 2511
82.14 Puntatori a puntatori 2517
82.15 Puntatori a più dimensioni 2522
82.16 Parametri della funzione main() 2530
82.17 Puntatori a variabili distrutte 2533
82.18 Soluzioni agli esercizi proposti 2534
Indice analitico del volume 2541

Introduzione

Il corso contenuto in questa parte riguarda i concetti elementari della programmazione, al livello minimo di astrazione possibile, utilizzando il linguaggio C per la messa in pratica degli algoritmi. Il corso è «basilare», ma gli argomenti trattati non sono così semplici come il termine potrebbe fare supporre.

Gli argomenti del corso sono già trattati in altri capitoli dell'opera, ma qui, in più, si inseriscono degli esercizi corretti.[1]

Per svolgere il corso correttamente è indispensabile fare tutti gli esercizi, verificando le soluzioni. Se il corso è guidato da un tutore, è bene presentarsi sempre alle lezioni avendo già studiato gli argomenti che devono essere trattati e avendo fatto gli esercizi indicati.

Programma didattico

Il corso, se assistito da un tutore, prevede l'impiego di circa 45 ore, di cui, almeno otto da dedicare alle verifiche (due ore di verifica per modulo, più due ore aggiuntive per una verifica di recupero complessiva).

Modulo 1

- **sistemi di numerazione**
  - decimale
  - binario
  - ottale
  - esadecimale
  - conversioni numeriche intere

  - conversioni numeriche intere tra binario, ottale e esadecimale
- **operazioni aritmetiche elementari in binario**
  - complemento a uno
  - complemento a due
  - somma binaria
  - sottrazione binaria
  - rappresentazione dei numeri interi con segno
- **operazioni elementari all'interno della CPU**
  - aumento e riduzione delle cifre binarie di un numero intero senza segno
  - aumento e riduzione delle cifre binarie di un numero intero con segno
  - somme con i numeri interi con segno
  - somme e sottrazioni con i numeri interi senza segno
  - scorrimento logico (senza segno)
  - scorrimento aritmetico (con segno)
  - rotazione
  - AND
  - OR
  - XOR
  - NOT
- **organizzazione della memoria**
  - pila dei dati o *stack* (cenni)
  - chiamata di funzioni e passaggio degli argomenti attraverso la pila (cenni)
  - variabili scalari

  - array
  - stringhe
  - puntatori
  - ordine dei byte

Modulo 2

- **primo approccio al linguaggio C**
  - commenti, istruzioni, raggruppamenti
  - compilazione
  - emissione di messaggi testuali
  - sospensione dell'esecuzione del programma in attesa della pressione di [*Invio*]
  - costruzione del primo programma che emette un messaggio e attende la pressione di [*Invio*] per terminare
- **tipi principali del linguaggio C**
  - tipi scalari primitivi: char, short int, int, long int, float, double
  - tipi scalari primitivi: distinzione tra presenza e assenza del segno
  - costanti letterali
  - dichiarazione di variabili scalari
  - il tipo void
- **operatori ed espressioni del linguaggio C**
  - operatori aritmetici
  - operatori di confronto
  - operatori logici
  - operatori binari

  - cast (conversione di tipo)
  - espressioni multiple
- **strutture di controllo di flusso del linguaggio C**
  - if
  - switch
  - while
  - for
- **funzioni del linguaggio C**
  - funzione '`main (void)`'
  - prototipo
  - descrizione della funzione
  - valore restituito dalla funzione
  - valore restituito dal programma

Modulo 3

- **puntatori in C**
  - espressioni a cui si assegnano dei valori (*lvalue*)
  - dichiarazione di una variabile puntatore
  - dereferenziazione di un puntatore
  - *big endian*, *little endian* e puntatori
  - puntatori come parametri di una funzione
- **array in C**
  - dichiarazione di un array
  - selezione di un elemento all'interno di un array all'interno delle espressioni
  - array a più dimensioni
  - uso del ciclo '`for`' per la scansione di un array

  - relazione tra array e puntatori
  - dereferenziazione di un puntatore come se fosse un array
  - array come parametri di una funzione
  - aritmetica dei puntatori
  - stringhe
- **puntatori di puntatori**
  - dichiarazione e dereferenziazione
  - puntatori a più dimensioni, ovvero: array di puntatori
  - parametri della funzione ***main()***

Strumenti per la compilazione

«

Per potersi esercitare nell'uso del linguaggio C, è possibile avvalersi di un servizio *pastebin* completo, come *http://codepad.org* e *http://ideone.com* . A questi servizi ci si deve iscrivere, in modo da poter salvare i propri esercizi.

Se si dispone di un elaboratore completo, si può utilizzare un compilatore vero e proprio. I sistemi GNU e derivati, dispongono di norma del compilatore GNU C, ma in generale ogni sistema Unix dovrebbe consentire di compilare un programma utilizzando semplicemente il comando '`cc`', a cui si fa riferimento inizialmente nel capitolo del corso che introduce alla compilazione stessa.

Per compilare un programma C in un sistema operativo come MS-Windows, occorre uno strumento apposito. Nel caso di MS-Windows si suggerisce l'uso di Dev-C++ che è molto facile da installare e da usare, pur non offrendo il classico '`cc`' da riga di comando. Nelle figure successive viene mostrato, intuitivamente, il procedimento per creare un file, compilarlo ed eseguirlo.

Figura u5.1. Aspetto di Dev-C++ dopo l’avvio.

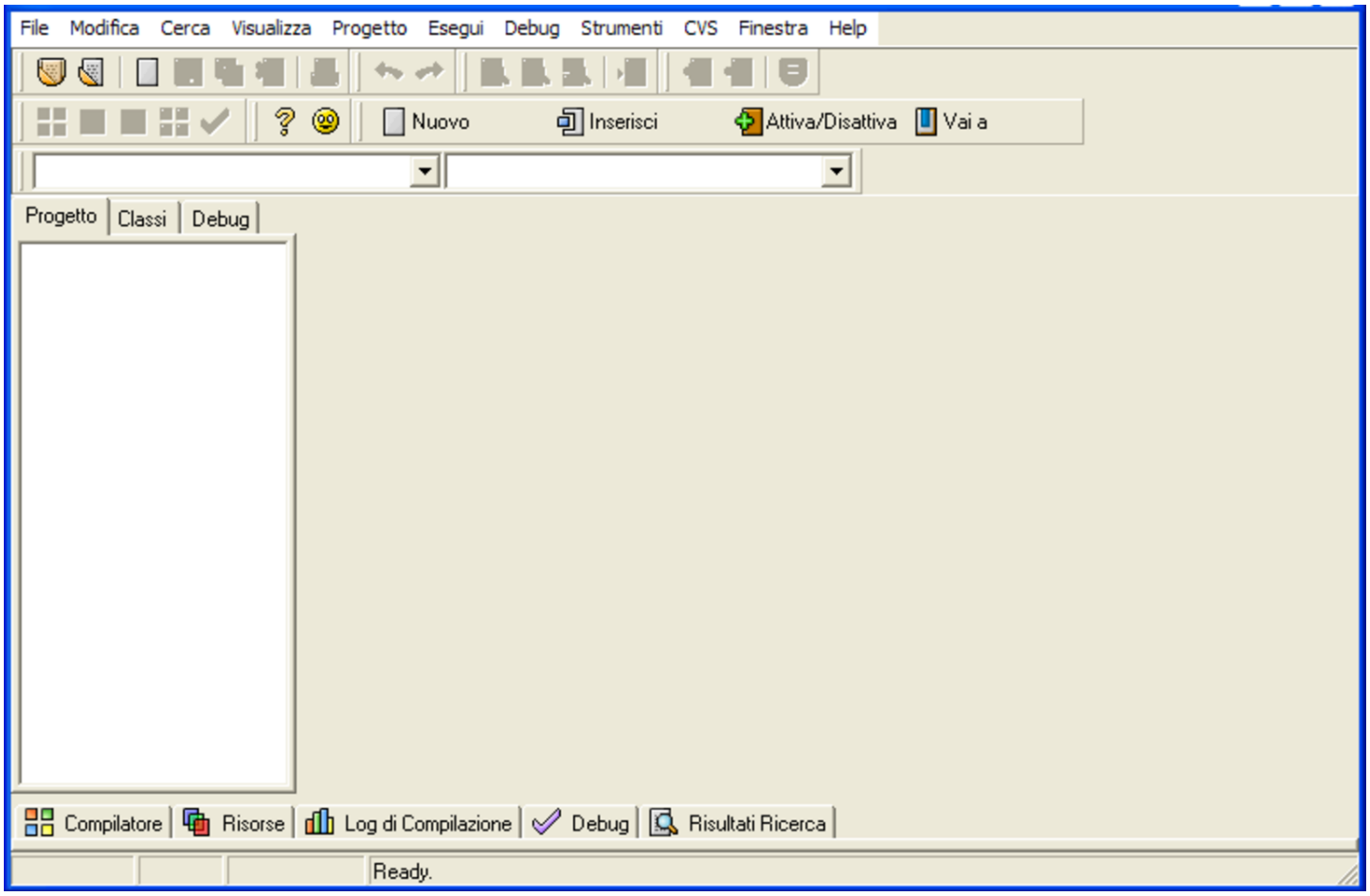


Figura u5.2. Creazione di un file sorgente nuovo.

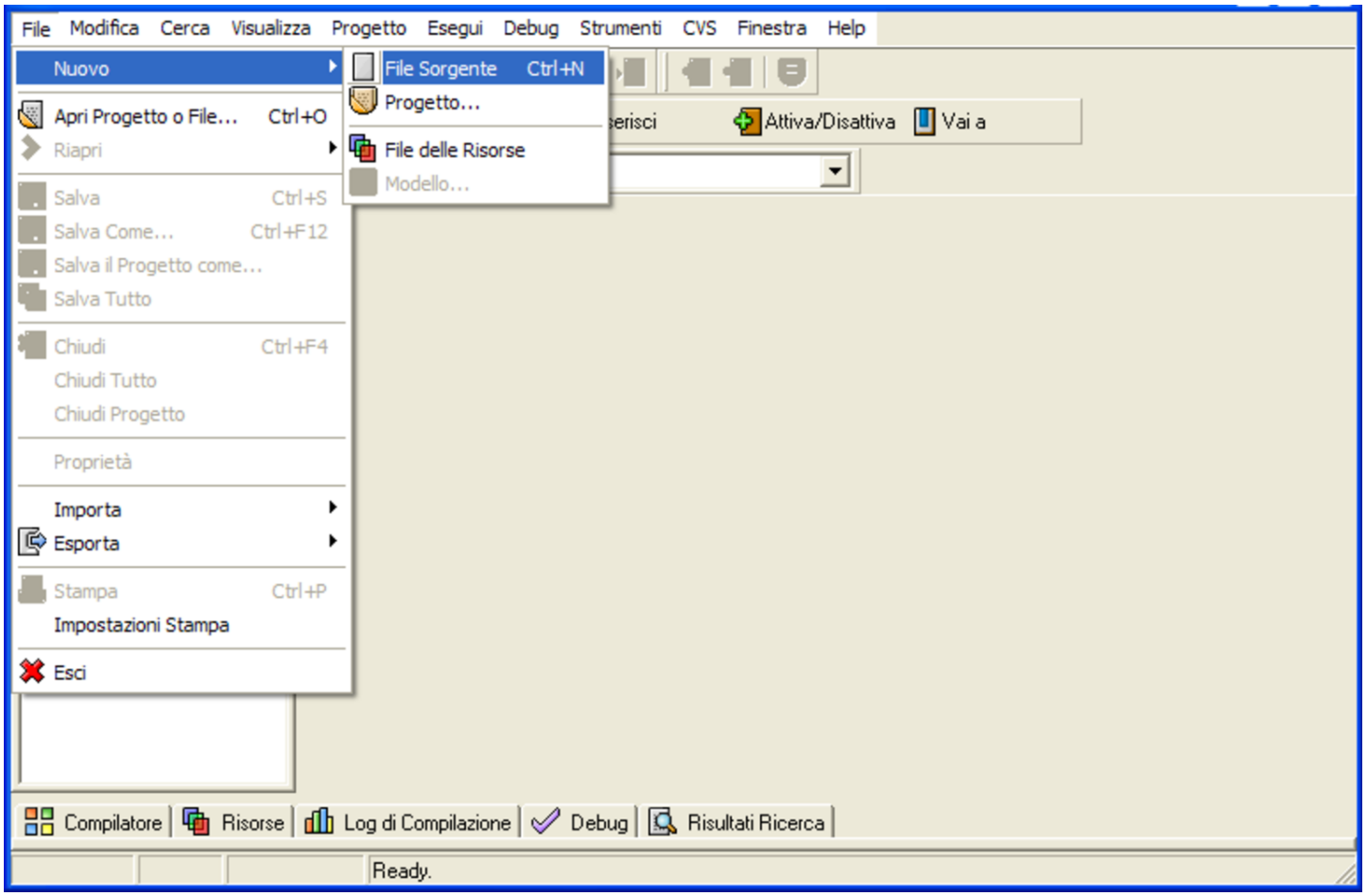


Figura u5.3. Un file che mostra un messaggio, attende la pressione di [*Invio*] e termina di funzionare.

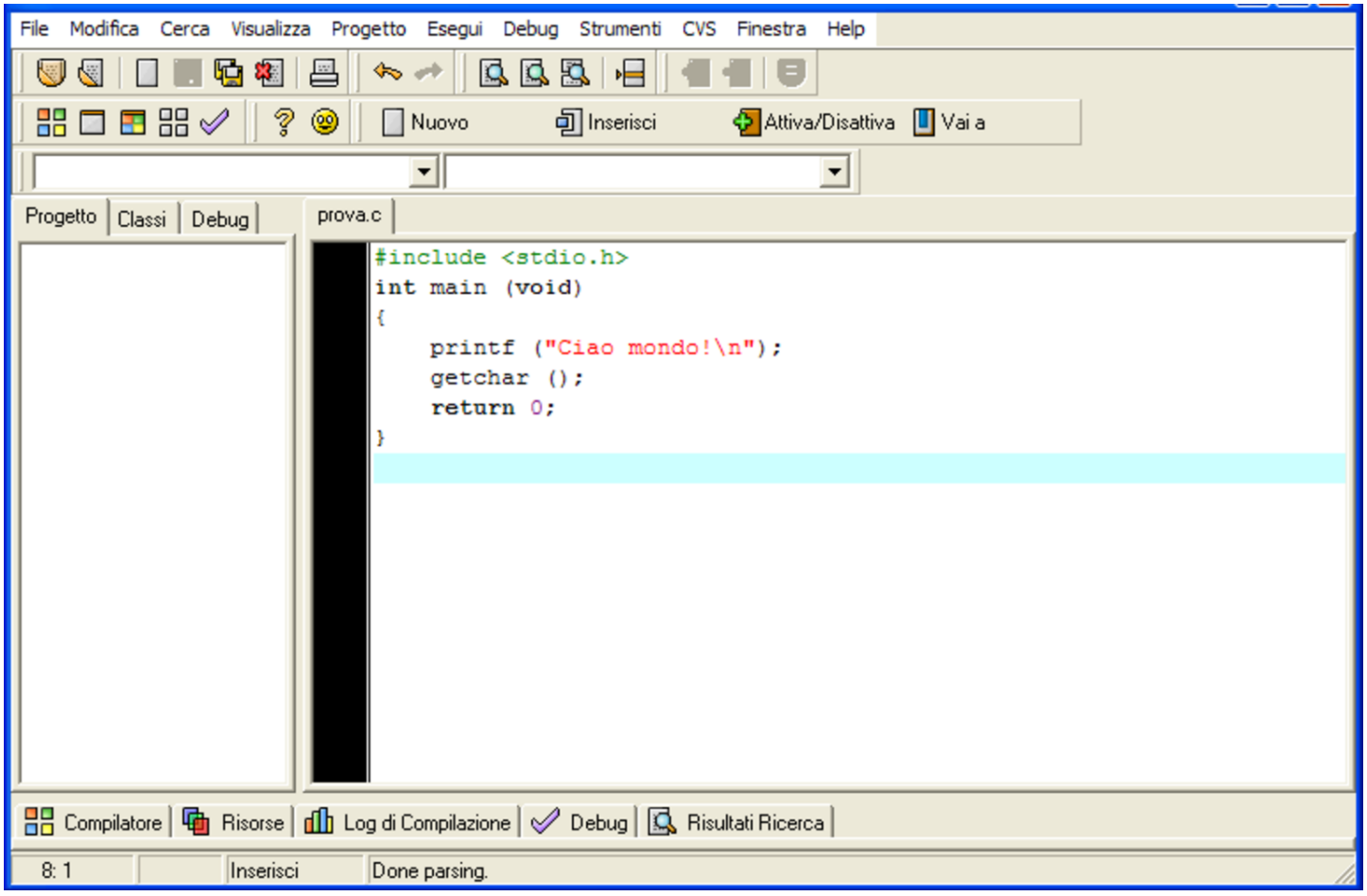


Figura u5.4. Compilazione.

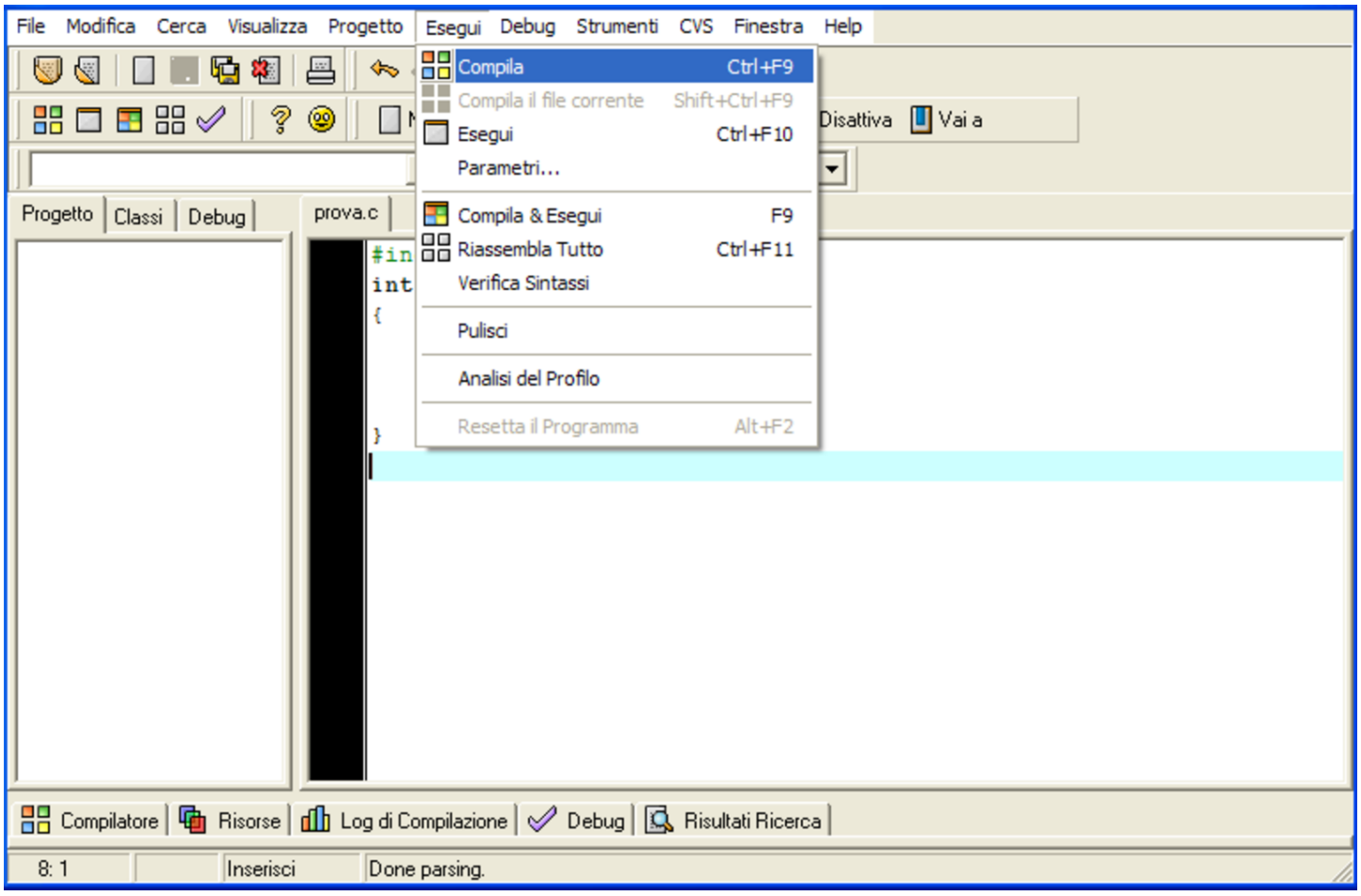


Figura u5.5. Esecuzione.

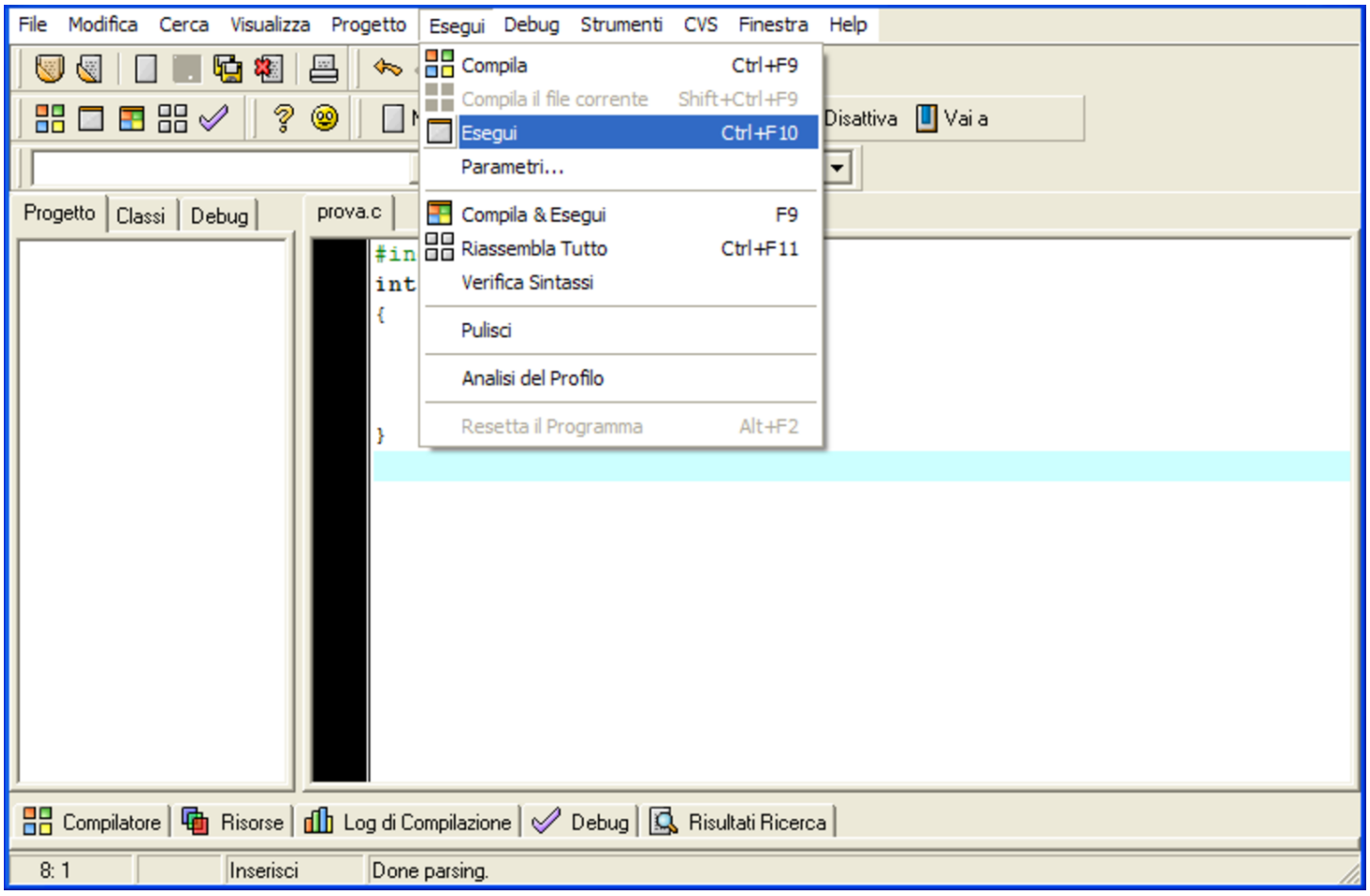


Figura u5.6. Finestra testuale da dove si vede l'emissione del messaggio del programma. Basta premere [*Invio*] per fare terminare il funzionamento del programma e lasciare così che la finestra si chiuda.

Riferimenti:

- *Codepad*, *http://codepad.org*
- *Ideone.com*, *http://ideone.com*
- BloodshedSoftware, *Dev-C++*, *http://www.bloodshed.net/devcpp.html* , *http://www.bloodshed.net/dev/* , *http://sourceforge.net/projects/dev-cpp/*

[1] Va tenuta sempre in considerazione la possibilità che alcune soluzioni o correzioni non siano esatte, pertanto, in caso di dubbio, va consultato un docente o comunque una persona competente.

Capitolo 80

Dai sistemi di numerazione all'organizzazione della memoria

80.1 Sistemi di numerazione 2281
80.1.1 Sistema decimale 2281
80.1.2 Sistema binario 2282
80.1.3 Sistema ottale 2284
80.1.4 Sistema esadecimale 2285
80.2 Conversioni numeriche di valori interi 2287
80.2.1 Numerazione ottale 2288
80.2.2 Numerazione esadecimale 2290
80.2.3 Numerazione binaria 2292
80.2.4 Conversione tra ottale, esadecimale e binario 2297
80.3 Conversioni numeriche di valori non interi 2299
80.3.1 Conversione da base 10 ad altre basi 2299
80.3.2 Conversione a base 10 da altre basi 2303
80.3.3 Conversione tra ottale, esadecimale e binario 2306
80.4 Operazioni elementari e sistema di rappresentazione binaria dei valori 2308
80.4.1 Complemento alla base di numerazione 2308
80.4.2 Complemento a uno e complemento a due 2311
80.4.3 Addizione binaria 2312
80.4.4 Sottrazione binaria 2313
80.4.5 Moltiplicazione binaria 2314

80.4.6 Divisione binaria 2315
80.4.7 Rappresentazione binaria di numeri interi senza segno 2315
80.4.8 Rappresentazione binaria di numeri interi con segno 2316
80.4.9 Cenni alla rappresentazione binaria di numeri in virgola mobile 2321
80.5 Calcoli con i valori binari rappresentati nella forma usata negli elaboratori 2323
80.5.1 Modifica della quantità di cifre di un numero binario intero 2323
80.5.2 Sommatorie con i valori interi con segno 2326
80.5.3 Somme e sottrazioni con i valori interi senza segno 2330
80.5.4 Somme e sottrazioni in fasi successive 2335
80.6 Scorrimenti, rotazioni, operazioni logiche 2339
80.6.1 Scorrimento logico 2340
80.6.2 Scorrimento aritmetico 2341
80.6.3 Moltiplicazione 2342
80.6.4 Divisione 2343
80.6.5 Rotazione 2344
80.6.6 Operatori logici 2345
80.7 Organizzazione della memoria 2347
80.7.1 Pila per salvare i dati 2347
80.7.2 Chiamate di funzioni 2348

80.7.3 Variabili e array 2351
80.7.4 Ordine dei byte 2358
80.7.5 Stringhe, array e puntatori 2360
80.7.6 Utilizzo della memoria 2362
80.8 Riferimenti 2365
80.9 Soluzioni agli esercizi proposti 2365

80.1 Sistemi di numerazione

I sistemi di numerazione più comuni sono di tipo posizionale, definiti in tal modo perché la posizione in cui appaiono le cifre ha significato. I sistemi di numerazione posizionali si distinguono per la ***base di numerazione***.

80.1.1 Sistema decimale

Il sistema di numerazione decimale è tale perché utilizza dieci simboli, pertanto è un sistema ***in base dieci***. Trattandosi di un sistema di numerazione posizionale, le cifre numeriche, da «0» a «9», vanno considerate secondo la collocazione relativa tra di loro.

A titolo di esempio si può prendere il numero 745 che, eventualmente, va rappresentato in modo preciso come 745_{10}: secondo l'esperienza comune si comprende che si tratta di settecento, più quaranta, più cinque, ovvero, settecentoquarantacinque. Si arriva a questo valore sapendo che la prima cifra a destra rappresenta delle unità (cinque unità), la seconda cifra a partire da destra rappresenta delle decine

(quattro decine), la terza cifra a partire da destra rappresenta delle centinaia (sette centinaia).

Figura 80.1. Esempio di scomposizione di un numero in base dieci.

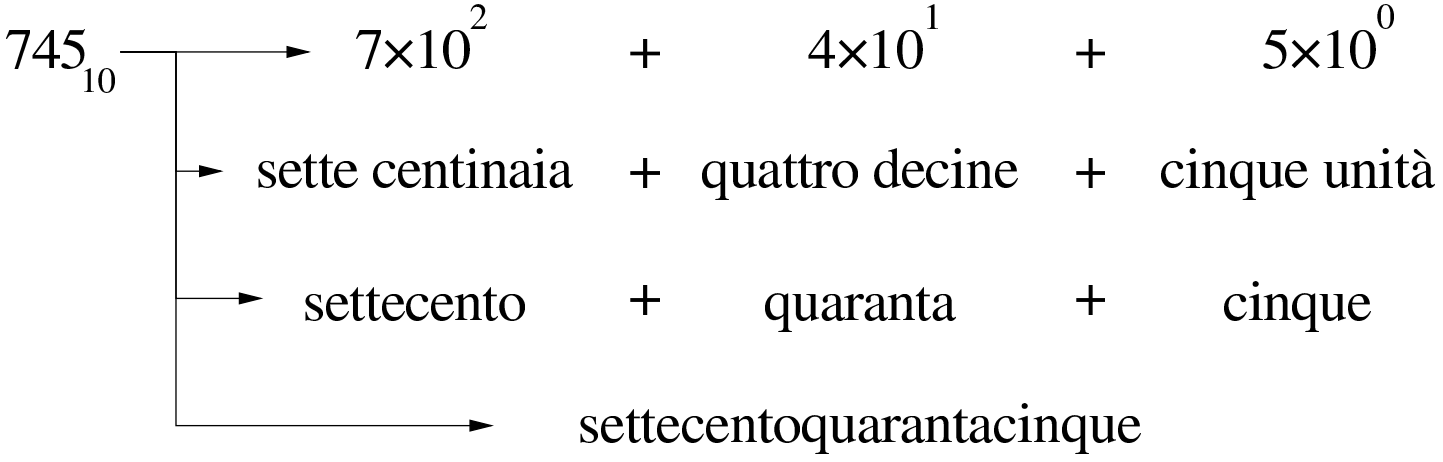
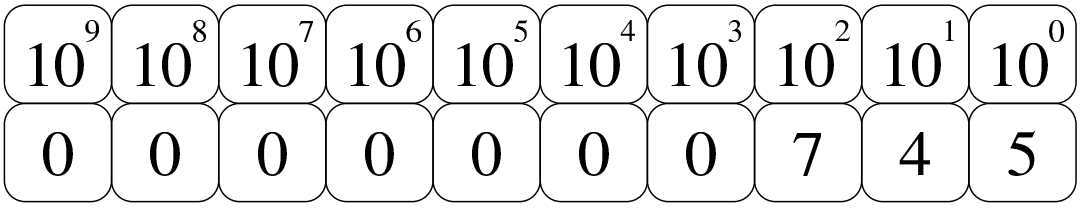


Figura 80.2. Scomposizione di un numero in base dieci.



80.1.2 Sistema binario

«

Il sistema di numerazione binario (in base due), utilizza due simboli: «0» e «1».

Figura 80.3. Esempio di scomposizione di un numero in base due.

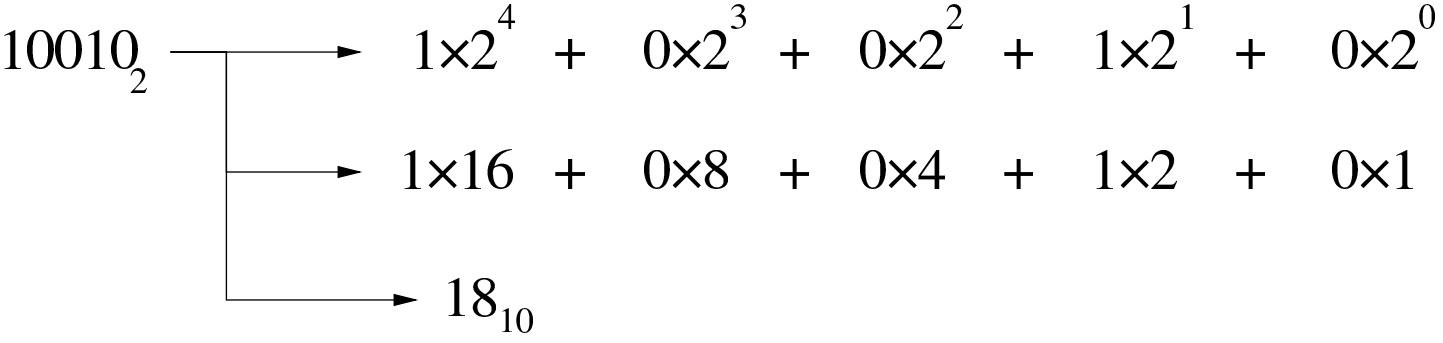


Figura 80.4. Scomposizione di un numero in base due.

2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	0	0	0	1	0	0	1	0

80.1.2.1 Esercizio

«

Si traduca il valore 11110011_2 in base dieci, con l'aiuto dello schema successivo, completandolo con una matita o con una penna, eventualmente con l'uso di una calcolatrice comune:

2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Pertanto, il risultato in base dieci è:

10^9	10^8	10^7	10^6	10^5	10^4	10^3	10^2	10^1	10^0

80.1.2.2 Esercizio

«

Si traduca il valore 01100110_2 in base dieci, con l'aiuto dello schema successivo, completandolo con una matita o con una penna, eventualmente con l'uso di una calcolatrice comune:

2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Pertanto, il risultato in base dieci è:

10^9	10^8	10^7	10^6	10^5	10^4	10^3	10^2	10^1	10^0

80.1.3 Sistema ottale

Il sistema di numerazione ottale (in base otto), utilizza otto simboli: da «0» a «7».

Figura 80.9. Esempio di scomposizione di un numero in base otto.

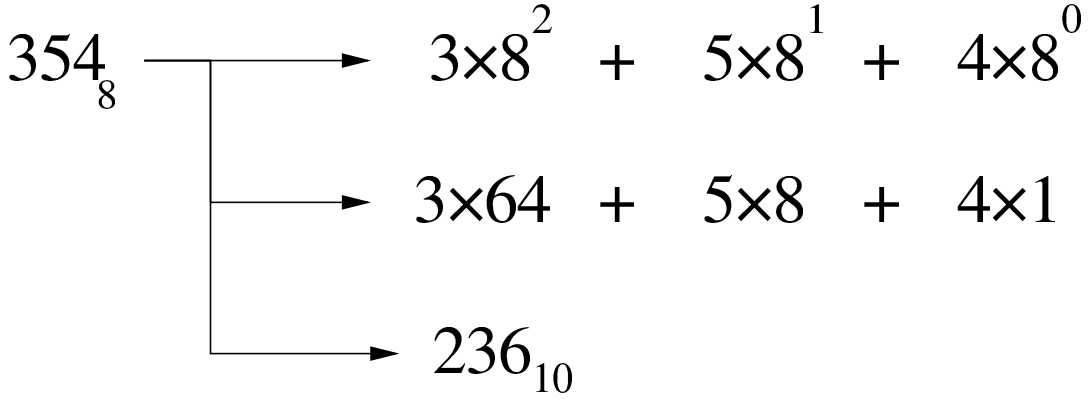


Figura 80.10. Scomposizione di un numero in base otto.

8^9	8^8	8^7	8^6	8^5	8^4	8^3	8^2	8^1	8^0
0	0	0	0	0	0	0	3	5	4

80.1.3.1 Esercizio

Si traduca il valore 1357_8 in base dieci, con l'aiuto dello schema successivo, completandolo con una matita o con una penna, eventualmente con l'uso di una calcolatrice comune:

8^9	8^8	8^7	8^6	8^5	8^4	8^3	8^2	8^1	8^0

Pertanto, il risultato in base dieci è:

10^9	10^8	10^7	10^6	10^5	10^4	10^3	10^2	10^1	10^0

80.1.3.2 Esercizio

«

Si traduca il valore 7531_8 in base dieci, con l'aiuto dello schema successivo, completandolo con una matita o con una penna, eventualmente con l'uso di una calcolatrice comune:

8^9	8^8	8^7	8^6	8^5	8^4	8^3	8^2	8^1	8^0

Pertanto, il risultato in base dieci è:

10^9	10^8	10^7	10^6	10^5	10^4	10^3	10^2	10^1	10^0

80.1.4 Sistema esadecimale

«

Il sistema di numerazione esadecimale (in base sedici), utilizza sedici simboli: le cifre numeriche da «0» a «9» e le lettere (maiuscole) dalla «A» alla «F».

Figura 80.15. Esempio di scomposizione di un numero in base sedici.

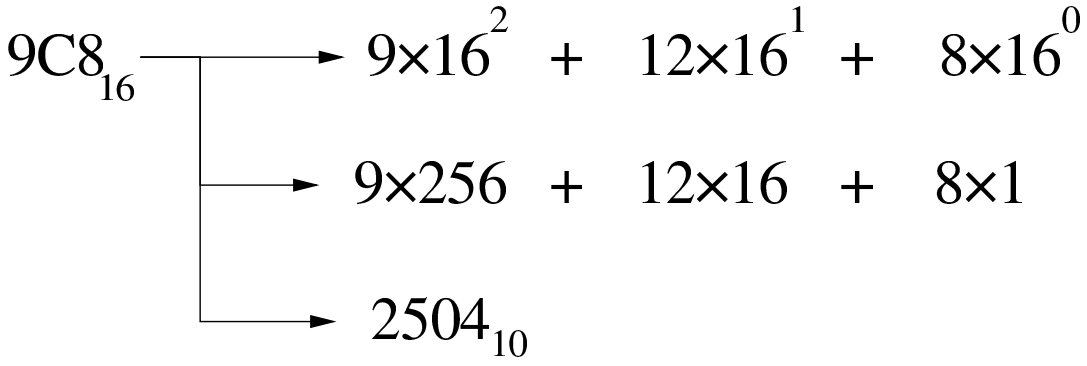


Figura 80.16. Scomposizione di un numero in base sedici.

16^9	16^8	16^7	16^6	16^5	16^4	16^3	16^2	16^1	16^0
0	0	0	0	0	0	0	9	C	8

80.1.4.1 Esercizio

«

Si traduca il valore $15AC_{16}$ in base dieci, con l'aiuto dello schema successivo, completandolo con una matita o con una penna, eventualmente con l'uso di una calcolatrice comune:

16^9	16^8	16^7	16^6	16^5	16^4	16^3	16^2	16^1	16^0

Pertanto, il risultato in base dieci è:

10^9	10^8	10^7	10^6	10^5	10^4	10^3	10^2	10^1	10^0

80.1.4.2 Esercizio

«

Si traduca il valore $CF58_{16}$ in base dieci, con l'aiuto dello schema successivo, completandolo con una matita o con una penna, eventualmente con l'uso di una calcolatrice comune:

16^9	16^8	16^7	16^6	16^5	16^4	16^3	16^2	16^1	16^0

Pertanto, il risultato in base dieci è:

10^9	10^8	10^7	10^6	10^5	10^4	10^3	10^2	10^1	10^0

80.2 Conversioni numeriche di valori interi

«

Un numero intero espresso in base dieci, viene interpretato sommando il valore di ogni singola cifra moltiplicando per 10^n (***n*** rappresenta la cifra ***n***-esima, a partire da zero). Per esempio, 12345 si può esprimere come $5\times10^0 + 4\times10^1 + 3\times10^2 + 2\times10^3 + 1\times10^4$. Nello stesso modo, si può scomporre un numero per esprimerlo in base dieci dividendo ripetutamente il numero per la base, recuperando ogni volta il resto della divisione. Per esempio, il valore 12345 (che ovviamente è già espresso in base dieci), si scompone nel modo seguente: 12345/10=1234 con il resto di cinque; 1234/10=123 con il resto di quattro; 123/10=12 con il resto di tre; 12/10=1 con il resto di due; 1/10=0 con il resto di uno (quando si ottiene un quoziente nullo, la conversione è terminata). Ecco che la sequenza dei resti dà il numero espresso in base dieci: 12345.

Riquadro 80.21. Il resto della divisione.

Per riuscire a convertire un numero intero da una base di numerazione a un'altra, occorre sapere calcolare il resto della divisione.
Si immagini di avere un sacchetto di nove palline uguali, da dividere equamente fra quattro amici. Per calcolare quante palline spettano a ognuno, si esegue la divisione seguente:
9/4 = 2,25
Il risultato intero della divisione è due, pertanto ognuno dei quattro amici può avere due palline e il resto della divisione è costituito dalle palline che non possono essere suddivise. Come si comprende facilmente, il resto è di una pallina:
9 – (2×4) = 1

80.2.1 Numerazione ottale

«

La numerazione ottale, ovvero in base otto, si avvale di otto cifre per rappresentare i valori: da zero a sette. La tecnica di conversione di un numero ottale in un numero decimale è la stessa mostrata a titolo esemplificativo per il sistema decimale, con la differenza che la base di numerazione è otto. Per esempio, per interpretare il numero ottale 12345_8, si procede come segue: $5\times8^0 + 4\times8^1 + 3\times8^2 + 2\times8^3 + 1\times8^4$. Pertanto, lo stesso numero si potrebbe rappresentare in base dieci come 5 349. Al contrario, per convertire il numero 5 349 (qui espresso in base 10), si può procedere nel modo seguente: 5 349/8=668 con il resto di cinque; 668/8=83 con il resto di quattro; 83/8=10 con il resto di tre; 10/8=1 con il resto di due; 1/8=0 con il resto di uno. Ecco che così si riottiene il numero ottale 12345_8.

Figura 80.22. Conversione in base otto.

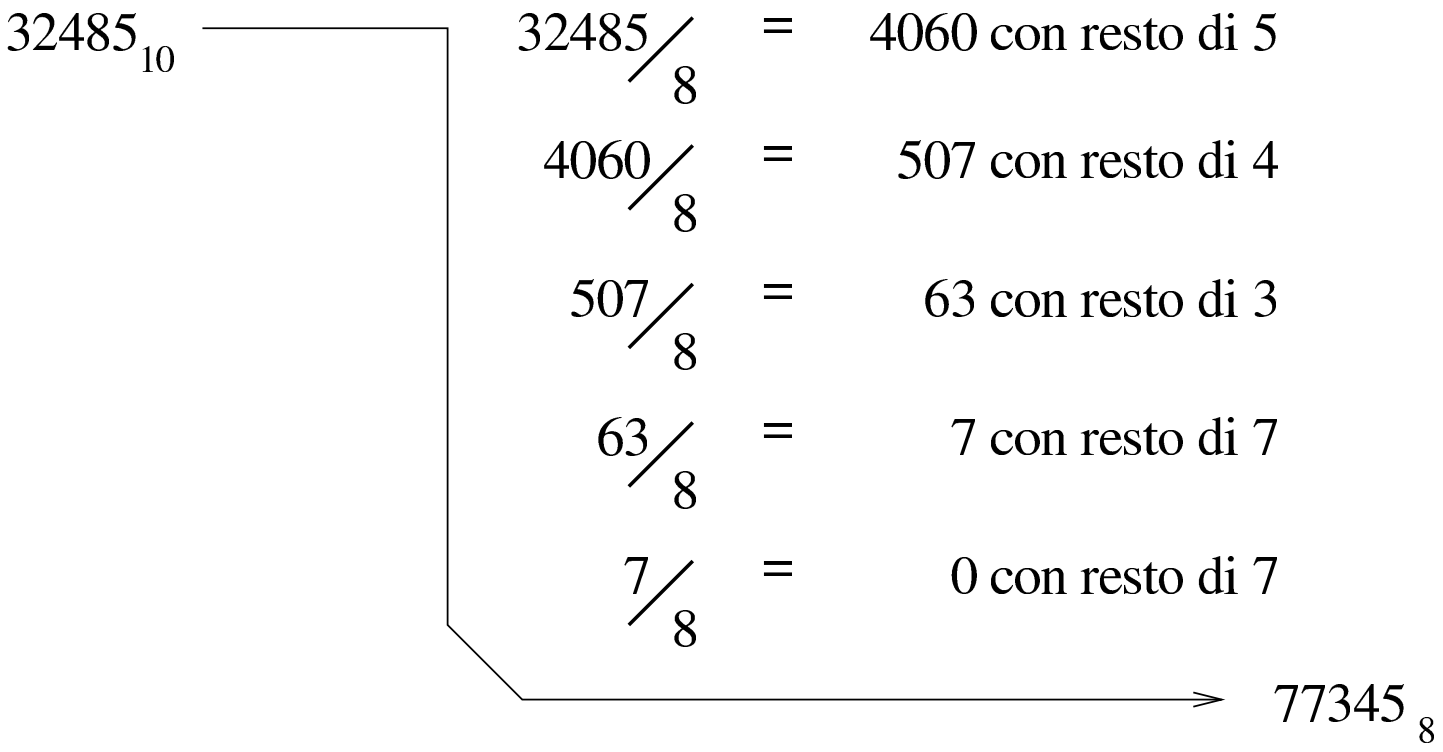
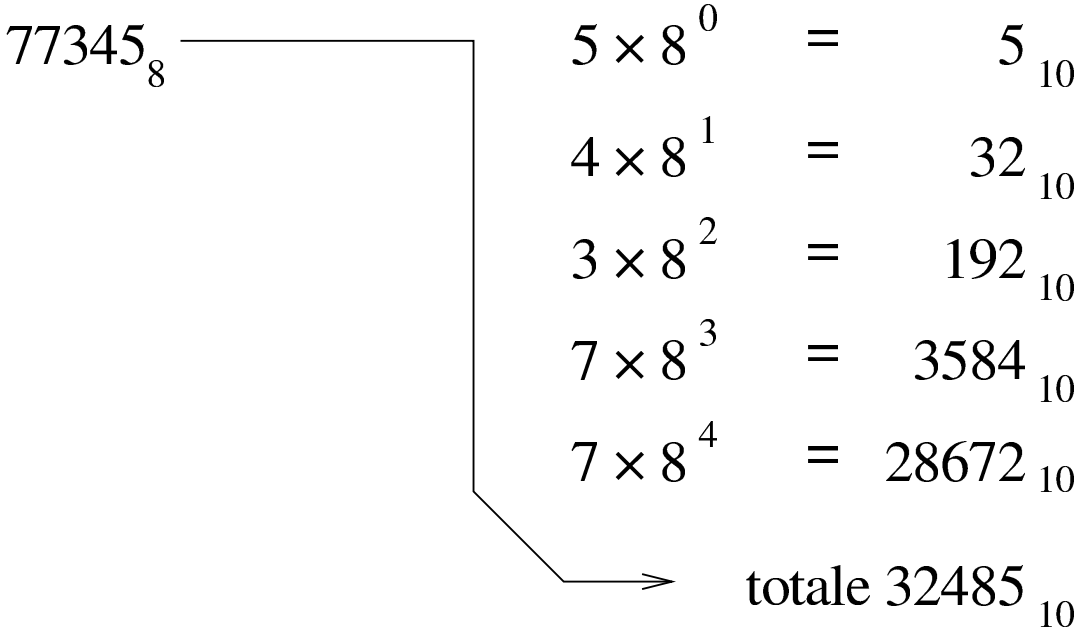


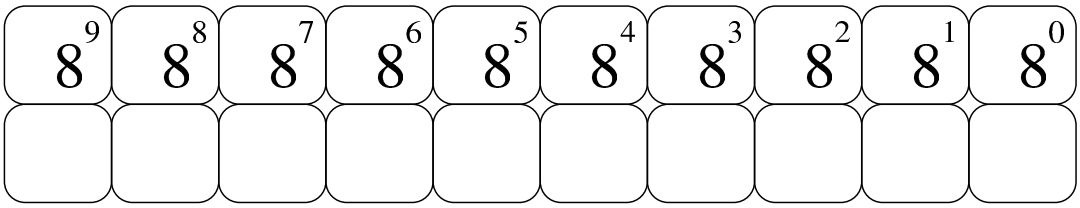
Figura 80.23. Calcolo del valore corrispondente di un numero espresso in base otto.



80.2.1.1 Esercizio

«

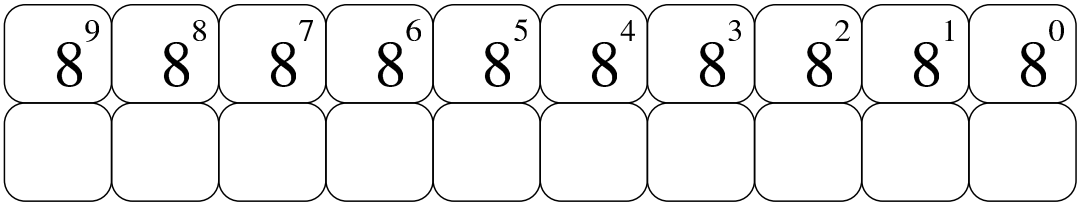
Si traduca il valore 1234_{10} in base otto, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.2.1.2 Esercizio

«

Si traduca il valore 4321_{10} in base otto, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.2.2 Numerazione esadecimale

«

La numerazione esadecimale, ovvero in base sedici, funziona in modo analogo a quella ottale, con la differenza che si avvale di 16 cifre per rappresentare i valori, per cui si usano le cifre numeriche da zero a nove, più le lettere da «A» a «F» per i valori successivi. In pratica, la lettera «A» nelle unità corrisponde al numero 10 e la lettera «F» nelle unità corrisponde al numero 15.

La tecnica di conversione è la stessa già vista per il sistema ottale, tenendo conto della difficoltà ulteriore introdotta dalle lettere aggiuntive. Per esempio, per interpretare il numero esadecimale $19ADF_{16}$, si procede come segue: $15 \times 16^0 + 13 \times 16^1 + 10 \times 16^2 +$

$9\times16^3 + 1\times16^4$. Pertanto, lo stesso numero si potrebbe rappresentare in base dieci come 105183. Al contrario, per convertire il numero 105183 (qui espresso in base 10), si può procedere nel modo seguente: 105183/16=6573 con il resto di 15, ovvero F_{16}; 6573/16=410 con il resto di 13, ovvero D_{16}; 410/16=25 con il resto di 10, ovvero A_{16}; 25/16=1 con il resto di nove; 1/16=0 con il resto di uno. Ecco che così si riottiene il numero esadecimale $19ADF_{16}$.

Figura 80.26. Conversione in base sedici.

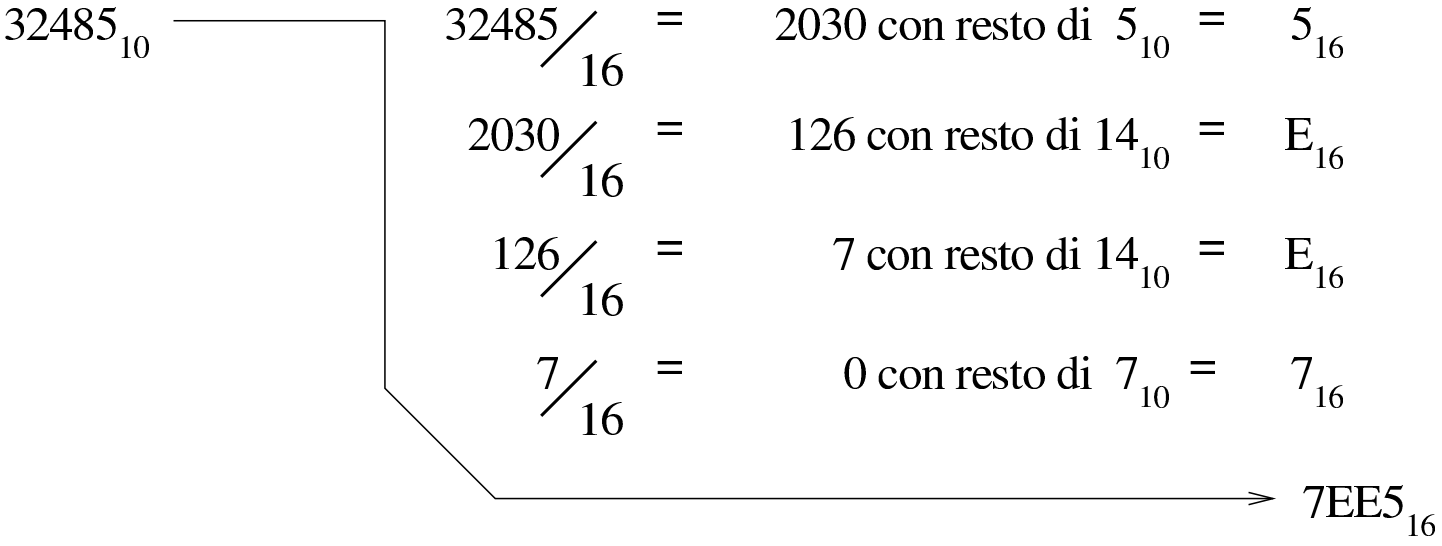
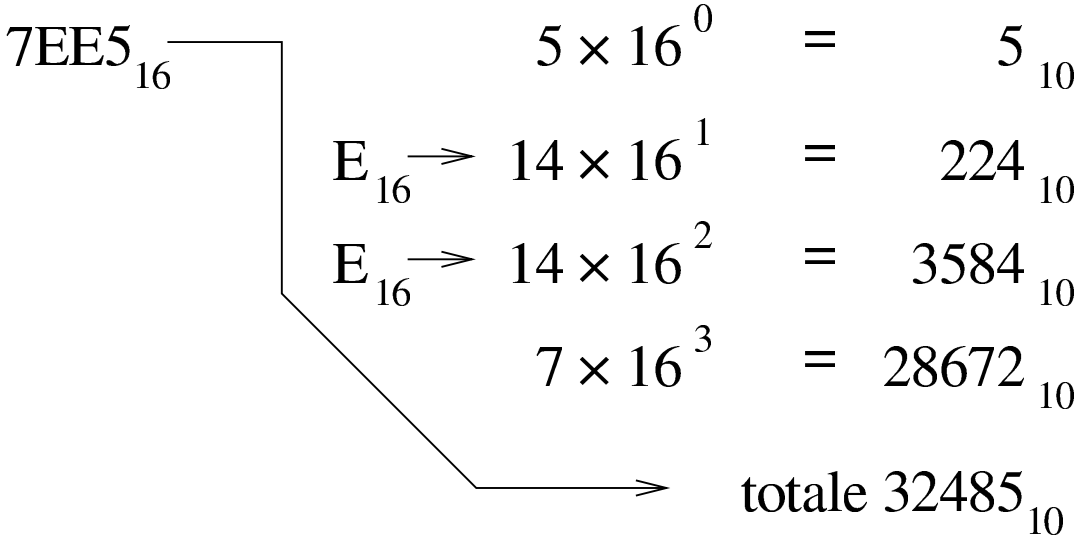
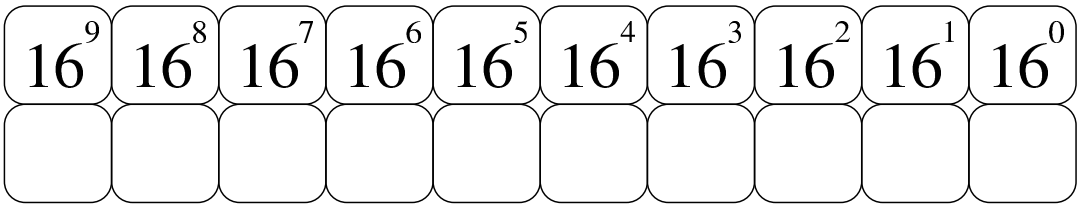


Figura 80.27. Calcolo del valore corrispondente di un numero espresso in base sedici.



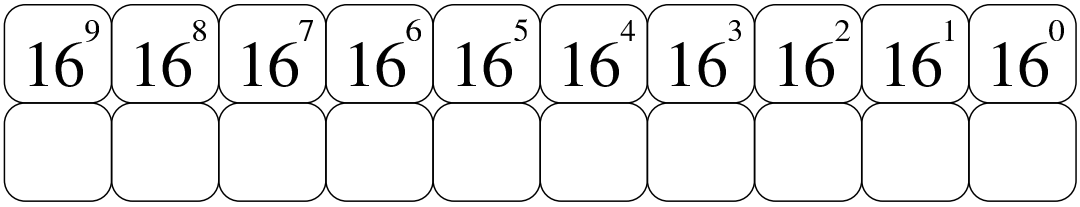
80.2.2.1 Esercizio

Si traduca il valore 44221_{10} in base sedici, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.2.2.2 Esercizio

Si traduca il valore 12244_{10} in base sedici, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.2.3 Numerazione binaria

La numerazione binaria, ovvero in base due, si avvale di sole due cifre per rappresentare i valori: zero e uno. Si tratta evidentemente di un esempio limite di rappresentazione di valori, dal momento che utilizza il minor numero di cifre. Questo fatto semplifica in pratica la conversione.

Seguendo la logica degli esempi già mostrati, si analizza brevemente la conversione del numero binario 1100_2: $0\times2^0 + 0\times2^1 + 1\times2^2 + 1\times2^3$. Pertanto, lo stesso numero si potrebbe rappresentare come 12 secondo il sistema standard. Al contrario, per convertire il numero 12, si può procedere nel modo seguente: 12/2=6 con il resto di zero; 6/2=3

con il resto di zero; 3/2=1 con il resto di uno; 1/2=0 con il resto di uno. Ecco che così si riottiene il numero binario 1100_2.

Figura 80.30. Conversione in base due.

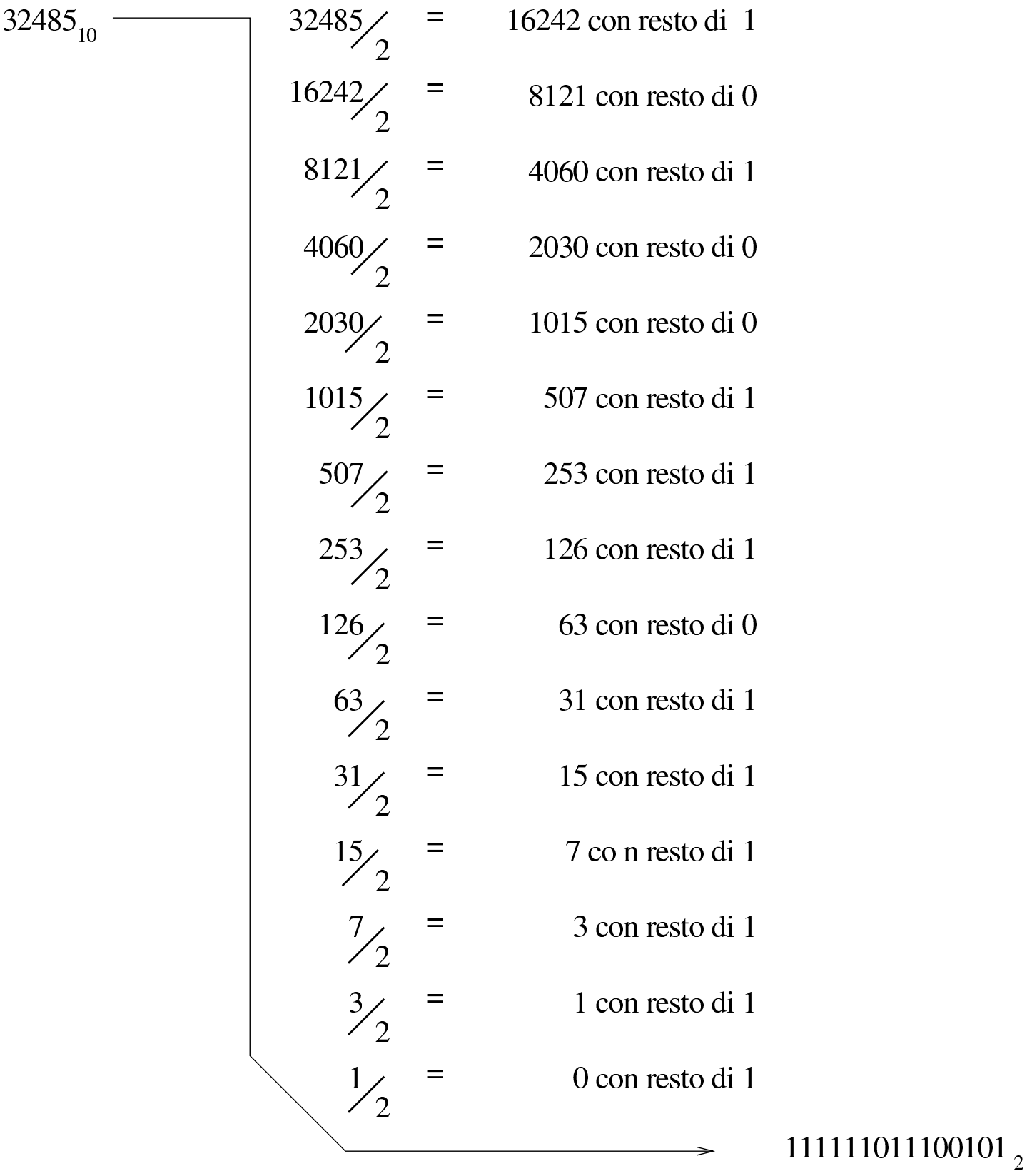
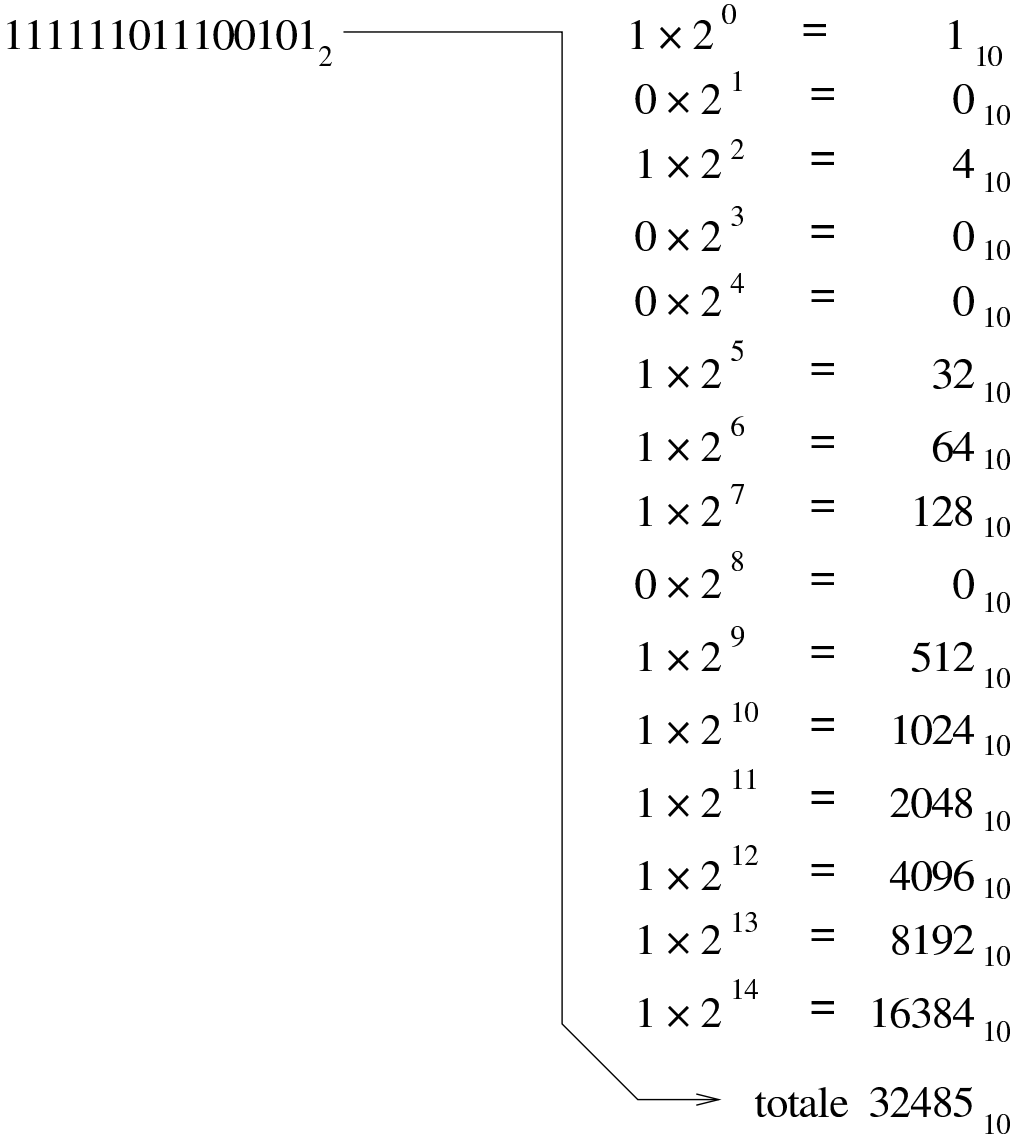
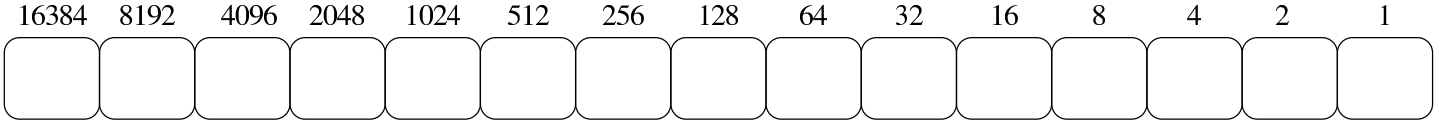


Figura 80.31. Calcolo del valore corrispondente di un numero espresso in base due.



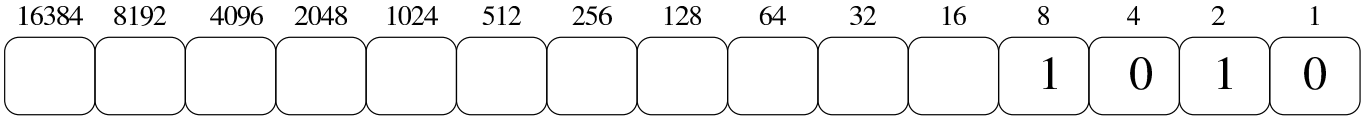
Si può convertire un numero in binario, in modo più semplice, se si costruisce una tabellina simile a quella seguente:



I valori indicati sopra ogni casellina sono la sequenza delle potenze di due: 2^0, 2^1, 2^2,... 2^n.

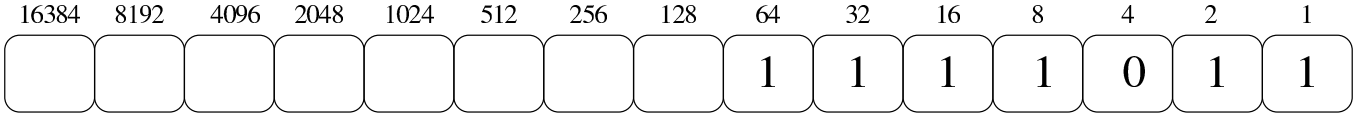
Se si vuole convertire un numero binario in base dieci, basta disporre

le sue cifre dentro le caselline, allineato a destra, moltiplicando ogni singola cifra per il valore che gli appare sopra, sommando poi ciò che si ottiene. Per esempio:



$8 + 0 + 2 + 0 = 10$

Per trovare il corrispondente binario di un numero in base 10, basta sottrarre sempre il valore più grande possibile. Supponendo di voler convertire il numero 123 in binario, si possono sottrarre i valori: 64, 32, 16, 8, 2 e 1:

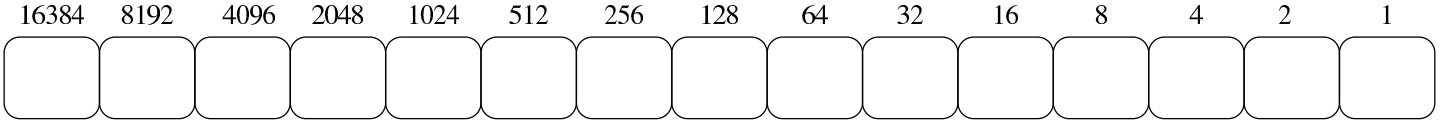


$64 + 32 + 16 + 8 + 0 + 2 + 1 = 123$

80.2.3.1 Esercizio

«

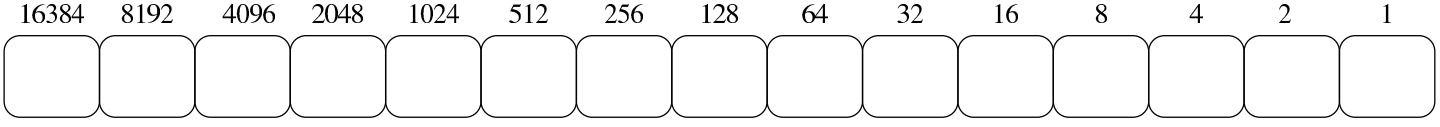
Si traduca il valore 1234_{10} in base due, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.2.3.2 Esercizio

«

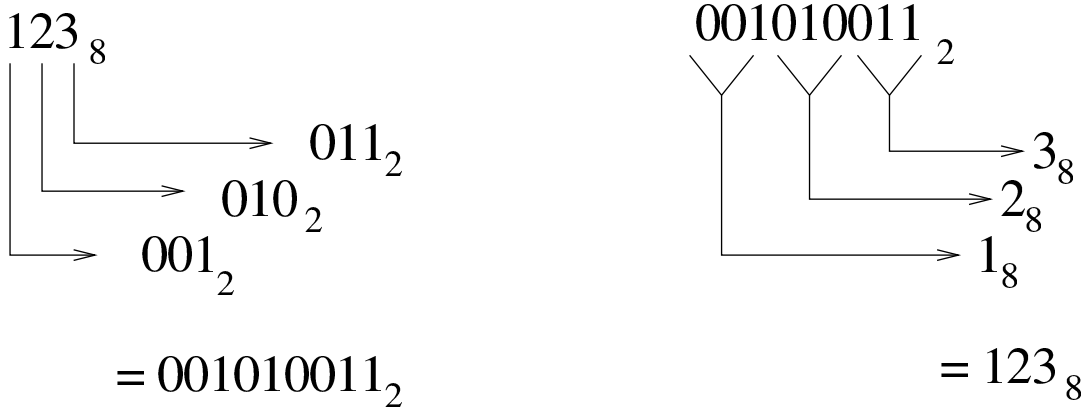
Si traduca il valore 4321_{10} in base due, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.2.4 Conversione tra ottale, esadecimale e binario

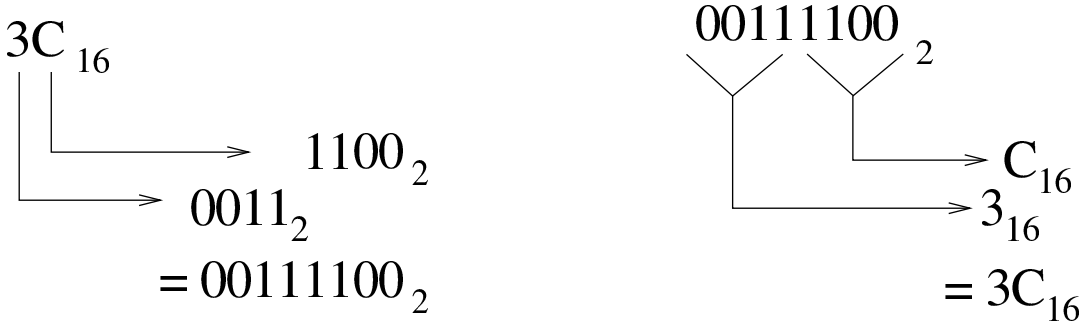
I sistemi di numerazione ottale ed esadecimale hanno la proprietà di convertirsi in modo facile in binario e viceversa. Infatti, una cifra ottale richiede esattamente tre cifre binarie per la sua rappresentazione, mentre una cifra esadecimale richiede quattro cifre binarie per la sua rappresentazione. Per esempio, il numero ottale 123_8 si converte facilmente in 001010011_2; inoltre, il numero esadecimale $3C_{16}$ si converte facilmente in 00111100_2.

Figura 80.37. Conversione tra la numerazione ottale e numerazione binaria.



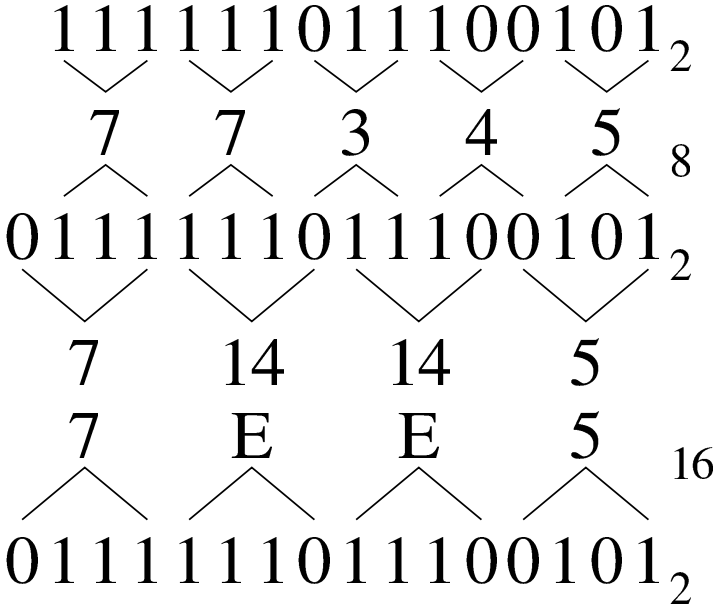
In pratica, è sufficiente convertire ogni cifra ottale o esadecimale nel valore corrispondente in binario. Quindi, sempre nel caso di 123_8, si ottengono 001_2, 010_2 e 011_2, che basta attaccare come già è stato mostrato. Nello stesso modo si procede nel caso di $3C_{16}$, che forma rispettivamente 0011_2 e 1100_2.

Figura 80.38. Conversione tra la numerazione esadecimale e numerazione binaria.



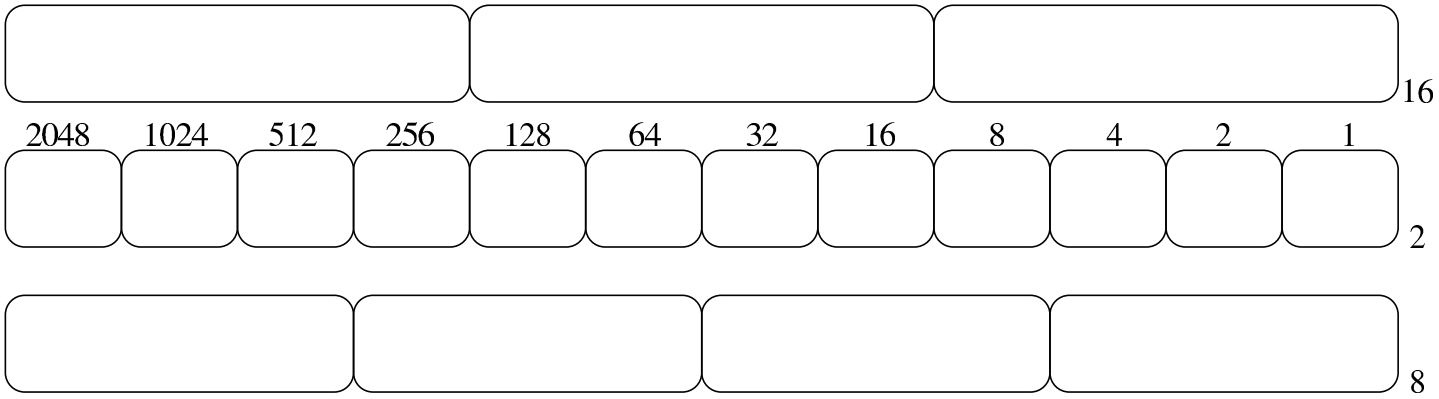
È evidente che risulta facilitata ugualmente la conversione da binario a ottale o da binario a esadecimale.

Figura 80.39. Riassunto della conversione tra binario-ottale e binario-esadecimale.



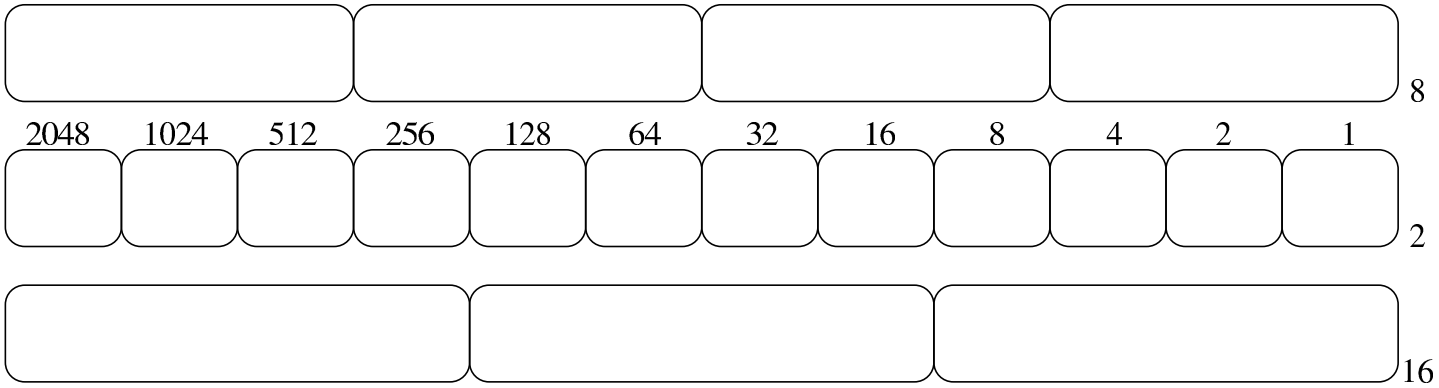
80.2.4.1 Esercizio

Si traduca il valore ABC_{16} in base due e quindi in base otto, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.2.4.2 Esercizio

Si traduca il valore 7655_8 in base due e quindi in base sedici, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.3 Conversioni numeriche di valori non interi

La conversione di valori non interi in basi di numerazione differenti, richiede un procedimento più complesso, dove si convertono, separatamente, la parte intera e la parte restante.

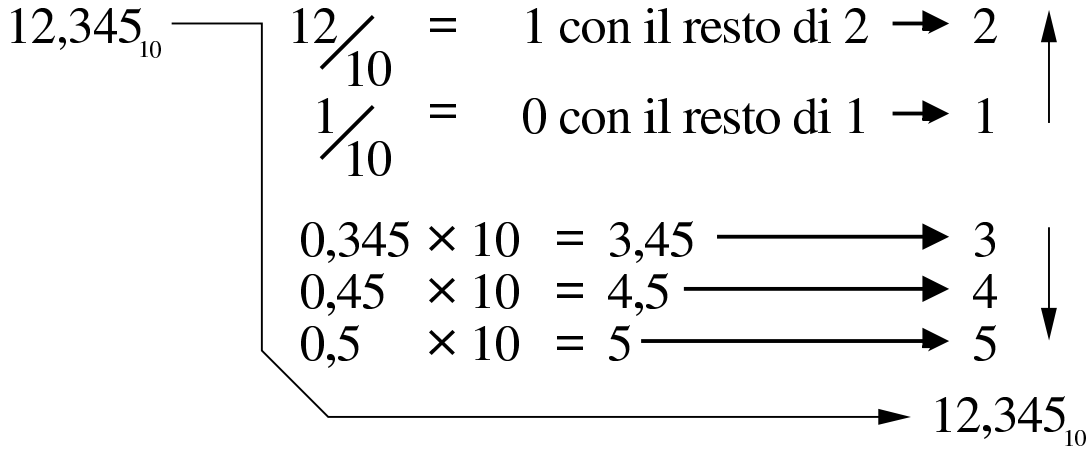
Il procedimento di scomposizione di un numero che contenga delle cifre dopo la parte intera, si svolge in modo simile a quello di un numero intero, con la differenza che le cifre dopo la parte intera vanno moltiplicate per la base elevata a una potenza negativa. Per esempio, il numero $12{,}345_{10}$ si può esprimere come $1\times10^1 + 2\times10^0 + 3\times10^{-1} + 4\times10^{-2} + 5\times10^{-3}$.

80.3.1 Conversione da base 10 ad altre basi

Come accennato nella premessa del capitolo, la conversione di un numero in un'altra base procede in due fasi: una per la parte intera, l'altra per la parte restante, unendo poi i due valori trovati. Per comprendere il meccanismo conviene simulare una conversione dalla base 10 alla stessa base 10, con un esempio: 12,345.

Per la parte intera, si procede come al solito, dividendo per la base di numerazione del numero da trovare e raccogliendo i resti; per la parte rimanente, il procedimento richiede invece di moltiplicare il valore per la base di destinazione e raccogliere le cifre intere trovate. Si osservi la figura successiva che rappresenta il procedimento.

Figura 80.42. Conversione da base 10 a base 10.



Quello che si deve osservare dalla figura è che l'ordine delle cifre cambia nelle due fasi del calcolo. Nelle figure successive si vedono altri esempi di conversione nelle altre basi di numerazione comuni.

Figura 80.43. Conversione da base 10 a base 16.

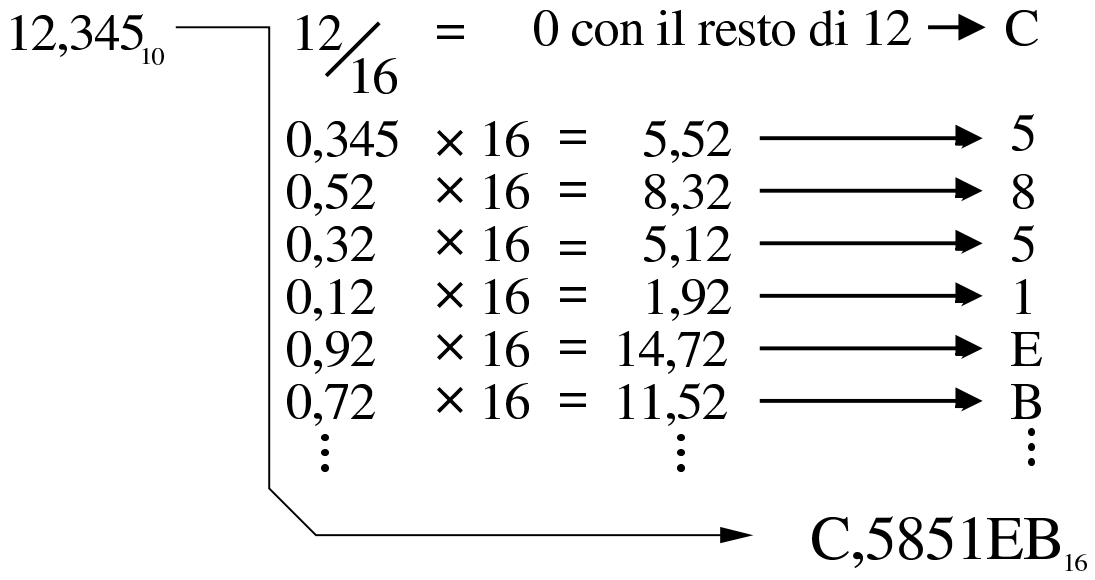


Figura 80.44. Conversione da base 10 a base 8.

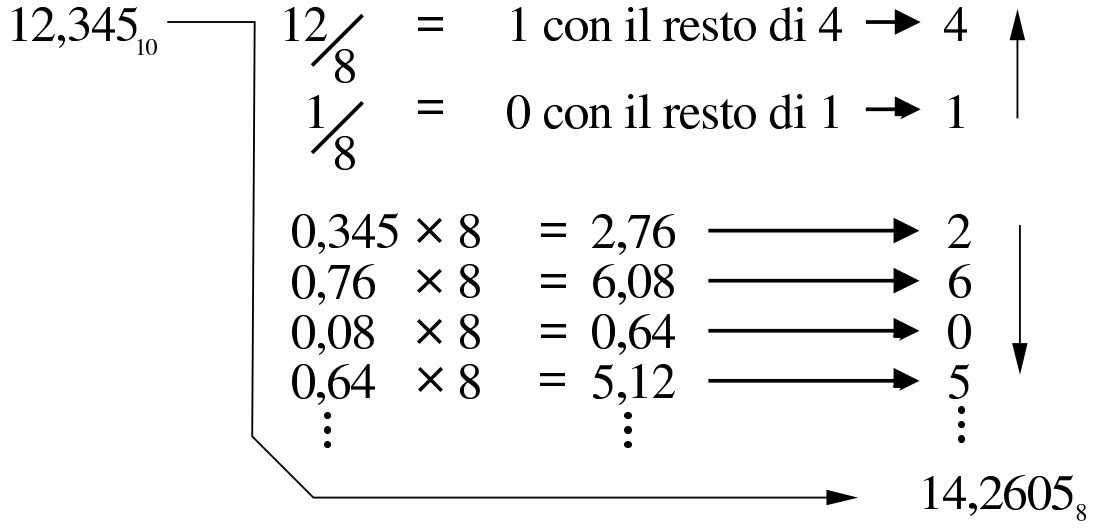
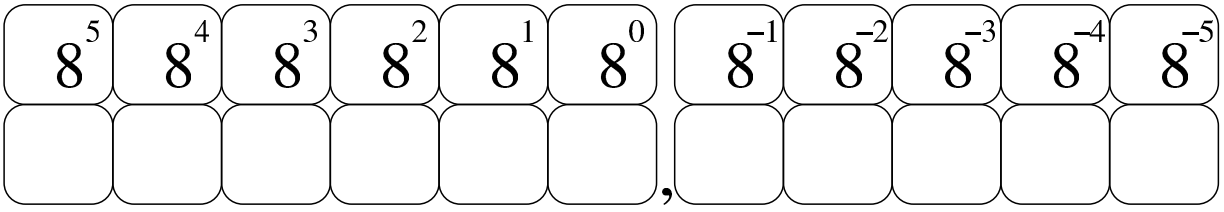


Figura 80.45. Conversione da base 10 a base 2.

80.3.1.1 Esercizio

«

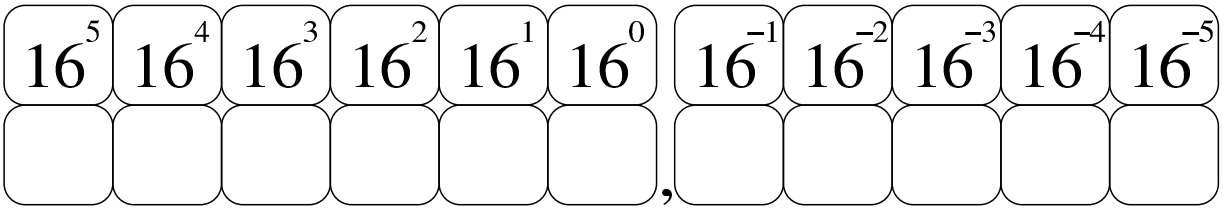
Si traduca il valore $43{,}21_{10}$ in base otto, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.3.1.2 Esercizio

«

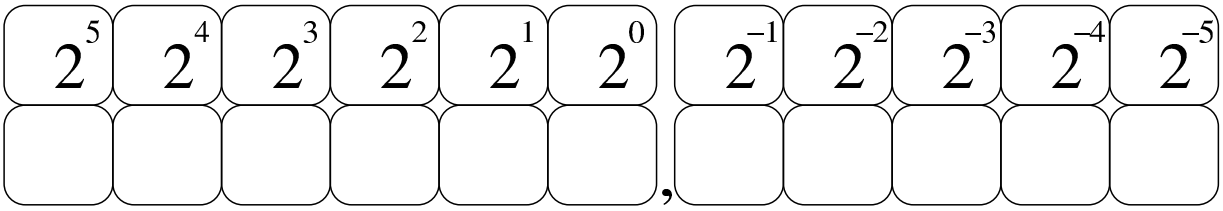
Si traduca il valore $765{,}4321_{10}$ in base sedici, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.3.1.3 Esercizio

«

Si traduca il valore $21{,}11_{10}$ in base due, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.3.2 Conversione a base 10 da altre basi

«

Per convertire un numero da una base di numerazione qualunque alla base 10, è necessario attribuire a ogni cifra il valore corrispondente, da sommare poi per ottenere il valore complessivo. Nelle figure successive si vedono gli esempi relativi alle basi di numerazione più comuni.

Figura 80.49. Conversione da base 16 a base 10.

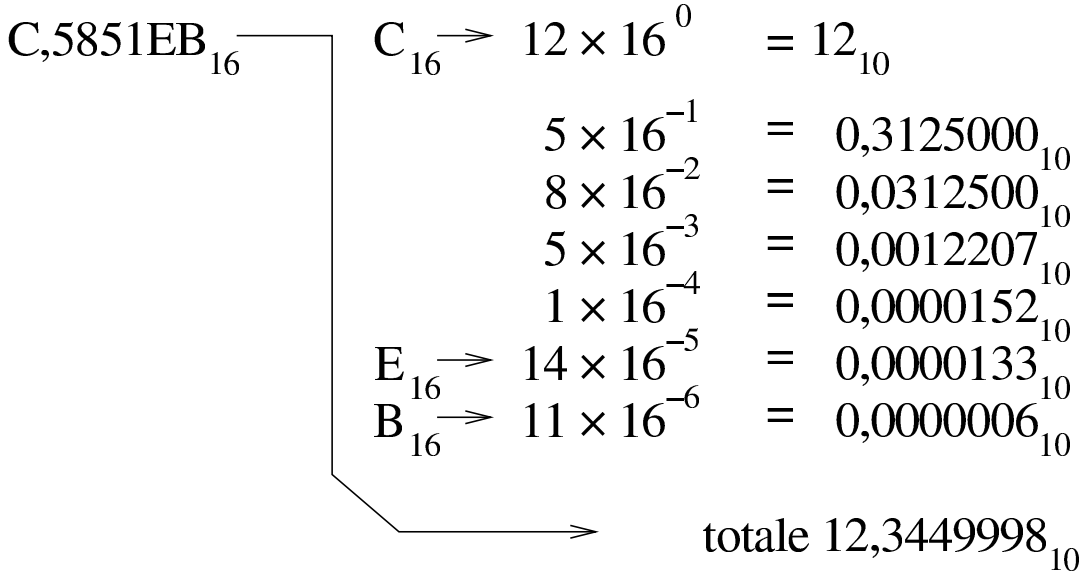


Figura 80.50. Conversione da base 8 a base 10.

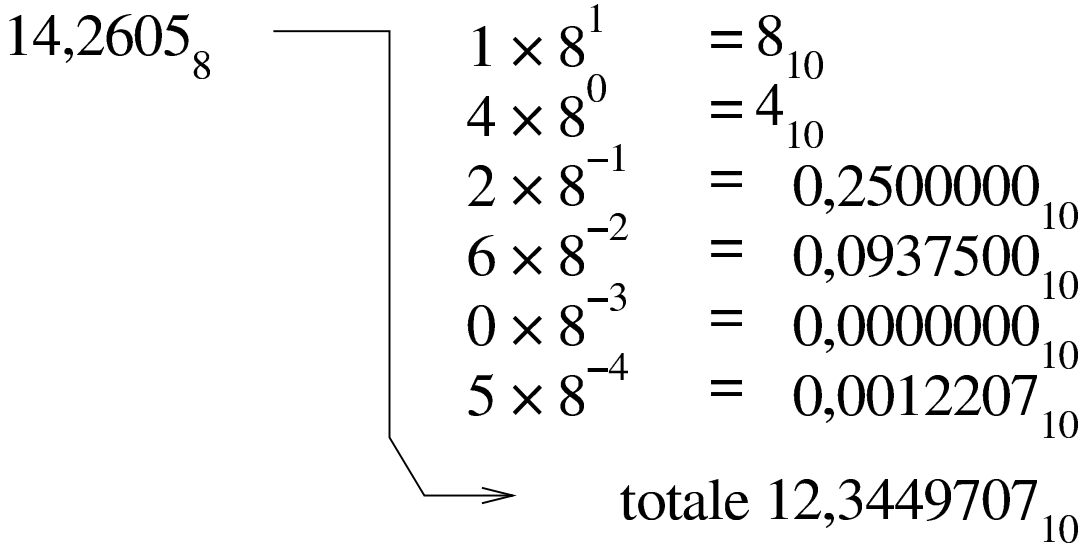
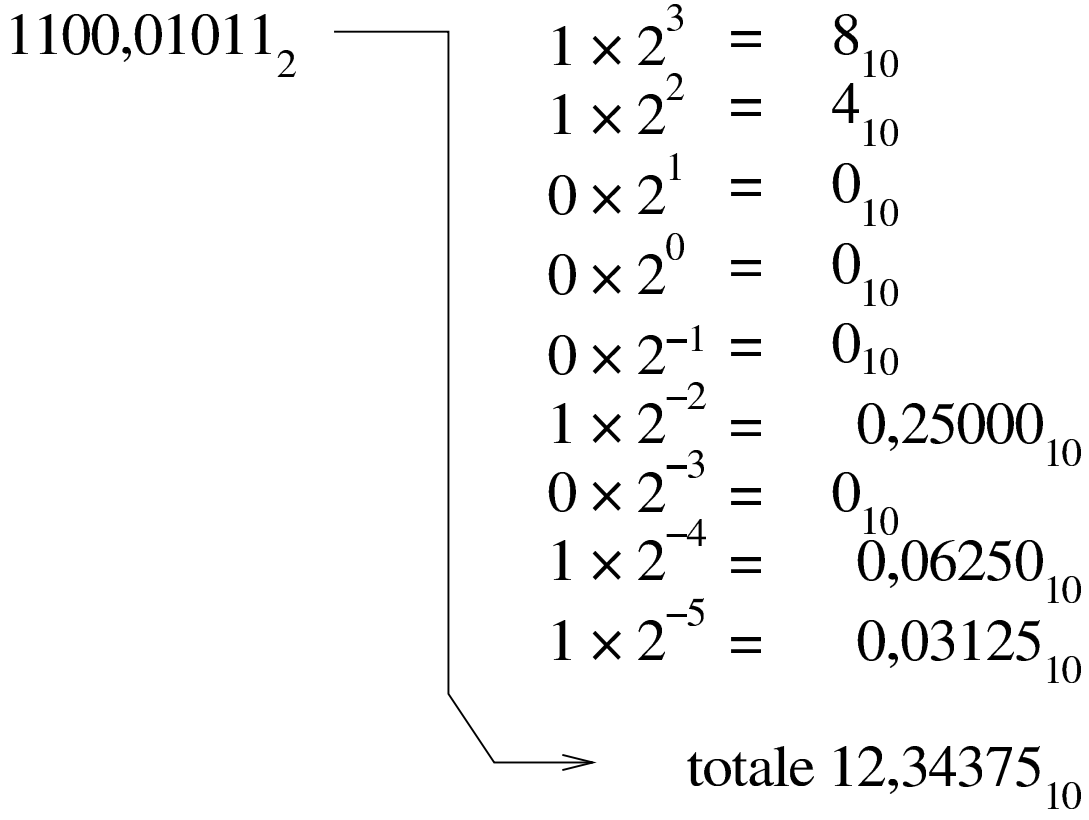
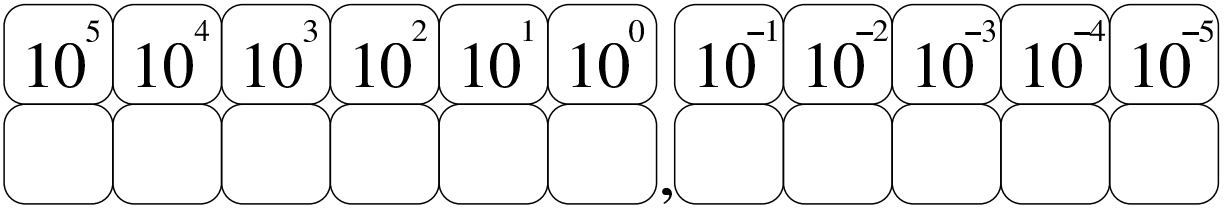


Figura 80.51. Conversione da base 2 a base 10.



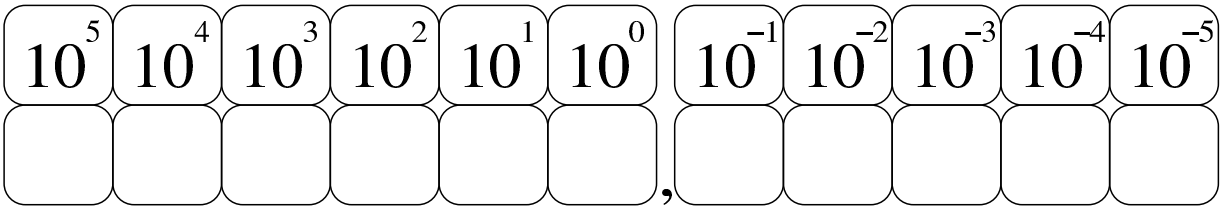
80.3.2.1 Esercizio

Si traduca il valore $765{,}432_8$ in base dieci, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.3.2.2 Esercizio

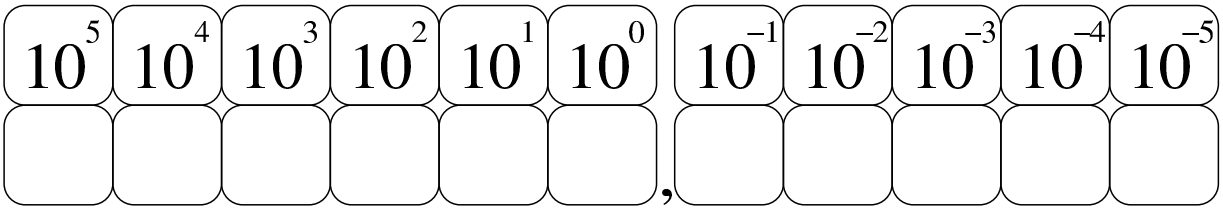
Si traduca il valore $AB{,}CD_{16}$ in base dieci, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.3.2.3 Esercizio

«

Si traduca il valore $101010{,}110011_2$ in base dieci, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:

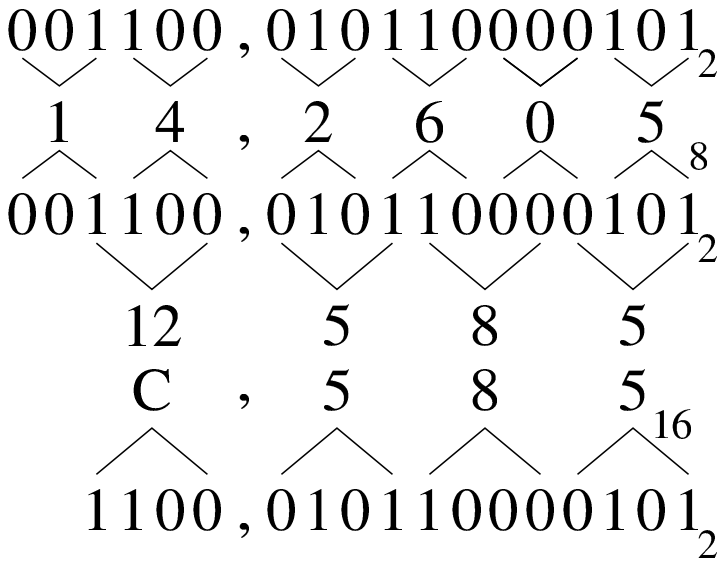


80.3.3 Conversione tra ottale, esadecimale e binario

«

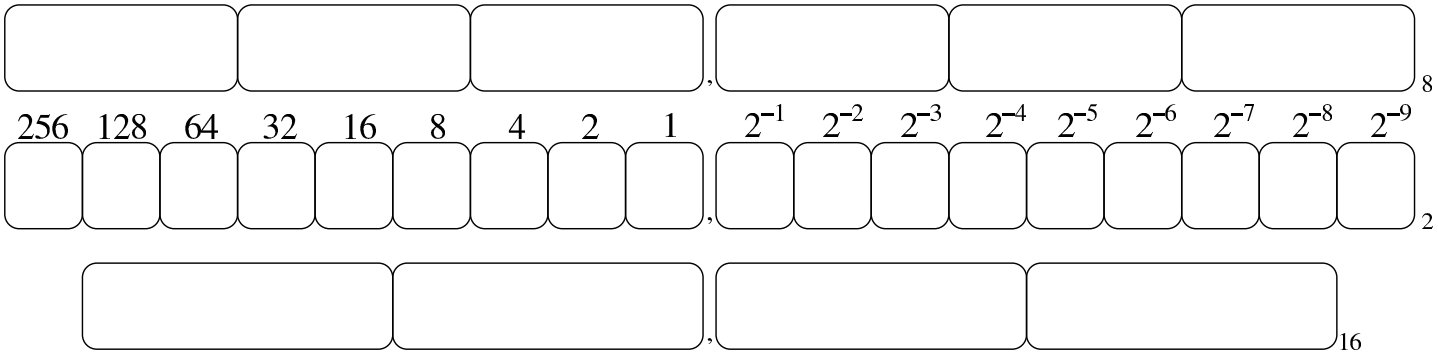
Per quanto riguarda la conversione tra sistemi di numerazione ottale, esadecimale e binario, vale lo stesso principio dei numeri interi, con la differenza che occorre rispettare la separazione della parte intera da quella decimale. L'esempio della figura successiva dovrebbe essere abbastanza chiaro.

Figura 80.55. Conversione tra binario-ottale e binario-esadecimale.



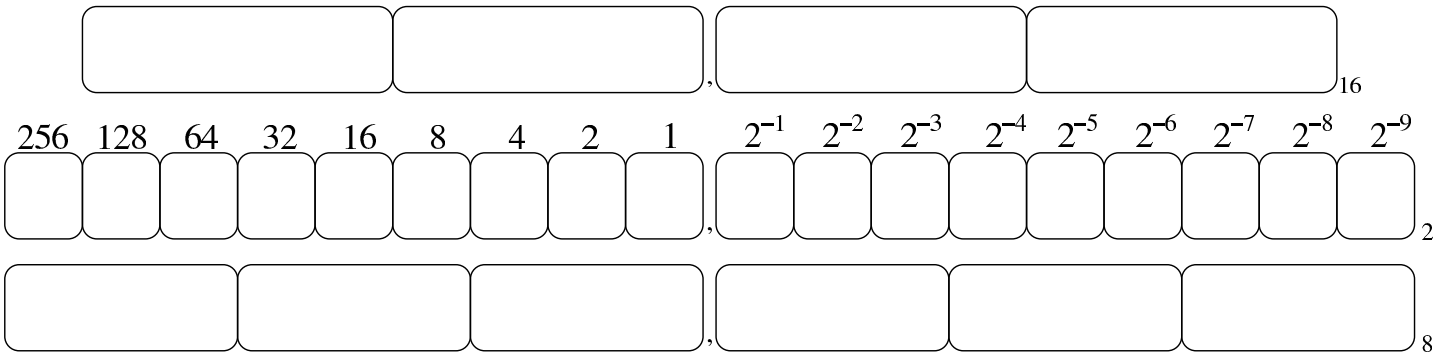
80.3.3.1 Esercizio

Si traduca il valore $76{,}55_8$ in base due e quindi in base sedici, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.3.3.2 Esercizio

Si traduca il valore $A7{,}C1_{16}$ in base due e quindi in base otto, con l'uso di una calcolatrice comune e di un foglio di carta per annotare i calcoli intermedi, compilando poi lo schema successivo:



80.4 Operazioni elementari e sistema di rappresentazione binaria dei valori

«

È importante conoscere alcuni concetti legati ai calcoli più semplici, applicati al sistema binario; soprattutto il modo in cui si utilizza il complemento a due. Infatti, la memoria di un elaboratore consente di annotare esclusivamente delle cifre binarie, in uno spazio di dimensione prestabilita e fissa; pertanto, attraverso il complemento a due si ha la possibilità di gestire in modo «semplice» la rappresentazione dei numeri interi negativi.

80.4.1 Complemento alla base di numerazione

«

Dato un numero n, espresso in base b, con k cifre, il ***complemento alla base*** è costituito da $b^k - n$.

Per esempio, il complemento alla base del numero 00123456789 (espresso in base dieci utilizzando 11 cifre) è 99876543211:

$$\begin{array}{r} 100000000000_{10} - \\ 00123456789_{10} = \\ \hline 99876543211_{10} \end{array}$$

Dall'esempio si deve osservare che la quantità di cifre utilizzata è determinante nel calcolo del complemento, infatti, il complemento alla

base dello stesso numero, usando però solo nove cifre (123456789) è invece 876543211:

$$\begin{array}{r} 1000000000_{10} - \\ 123456789_{10} = \\ \hline 876543211_{10} \end{array}$$

In modo analogo si procede con i valori aventi una base diversa; per esempio, il complemento alla base del numero binario 00110011_2, composto da otto cifre, è pari a 11001101_2:

$$\begin{array}{r} 100000000_2 - \\ 00110011_2 = \\ \hline 11001101_2 \end{array}$$

Il calcolo del complemento alla base, nel sistema binario, avviene in modo molto semplice, se si trasforma in questo modo:

$$\begin{array}{r} 11111111_2 - \\ 00110011_2 = \\ \hline 11001100_2 + \\ 1_2 = \\ \hline 11001101_2 \end{array}$$

In pratica, si prende un numero composto da una quantità di cifre a uno, pari alla stessa quantità di cifre del numero di partenza; quindi si esegue la sottrazione, poi si aggiunge il valore uno al risultato finale. Si osservi però cosa accade con una situazione leggermente differente, per il calcolo del complemento alla base di 0011001100_2:

$$
\begin{array}{r}
1111111111_2 - \\
0011001100_2 = \\
\hline
1100110011_2 + \\
1_2 = \\
\hline
1100110100_2
\end{array}
$$

Per eseguire una sottrazione, si può calcolare il complemento alla base del sottraendo (il valore da sottrarre), sommandolo poi al valore di partenza, trascurando il riporto eventuale. Per esempio, volendo sottrarre da 1757 il valore 758, si può calcolare il complemento alla base di 0758 (usando la stessa quantità di cifre dell'altro valore), per poi sommarla. Il complemento alla base di 0758 è 9242:

$$
\begin{array}{r}
10000_{10} - \\
0758_{10} = \\
\hline
9242_{10}
\end{array}
$$

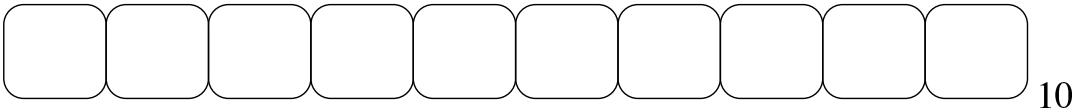
Invece di eseguire la sottrazione, si somma il valore ottenuto a quello di partenza, ignorando il riporto:

$$
\begin{array}{r}
1757_{10} + \\
9242_{10} = \\
\hline
10999_{10} - \\
10000_{10} = \\
\hline
999_{10}
\end{array}
$$

Infatti: 1757−758=999.

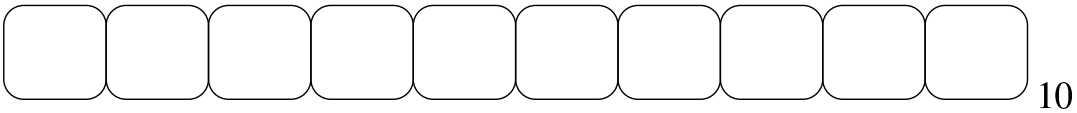
80.4.1.1 Esercizio

Si determini il complemento alla base del valore 0000123456_{10} (a dieci cifre), compilando lo schema successivo:



80.4.1.2 Esercizio

Si determini il complemento alla base del valore 9999123456_{10} (a dieci cifre), compilando lo schema successivo:



80.4.2 Complemento a uno e complemento a due

Quando si fa riferimento a numeri in base due, il complemento alla base è più noto come «complemento a due» (che evidentemente è la stessa cosa). D'altro canto, il complemento a uno è ciò che è già stato descritto con l'esempio seguente, dove si ottiene a partire dal numero 0011001100_2:

$$\begin{array}{r} 1111111111_2 - \\ 0011001100_2 = \\ \hline 1100110011_2 \end{array}$$

Si comprende intuitivamente che il complemento a uno si ottiene semplicemente invertendo le cifre binarie:

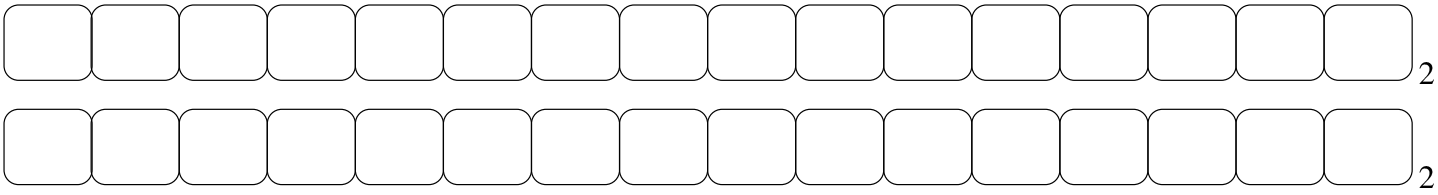
$$0011001100_2 \downarrow 1100110011_2$$

Pertanto, il complemento a due di un numero binario si ottiene facilmente invertendo le cifre del numero di partenza e aggiungendo una unità al risultato.

80.4.2.1 Esercizio

«

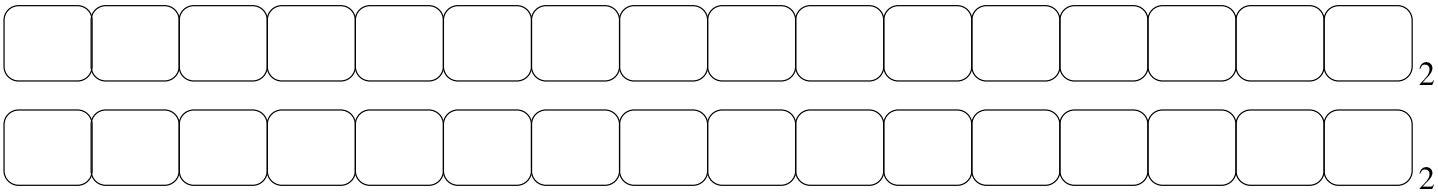
Si determini il complemento a uno e il complemento a due del valore 0011001001000101_2, compilando gli schemi successivi:



80.4.2.2 Esercizio

«

Si determini il complemento a uno e il complemento a due del valore 1111001100010101_2, compilando gli schemi successivi:



80.4.3 Addizione binaria

«

L'addizione binaria avviene in modo analogo a quella del sistema decimale, con la differenza che si utilizzano soltanto due cifre numeriche: 0 e 1. Pertanto, si possono presentare solo i casi seguenti:

$$0_2 + 0_2 = 0_2$$
$$0_2 + 1_2 = 1_2$$
$$1_2 + 0_2 = 1_2$$
$$1_2 + 1_2 = 10_2$$

ovvero: zero con riporto di uno

Segue l'esempio di una somma tra due numeri in base due:

$$\begin{array}{rl} 10011001_2 + & (153_{10}) \\ 00110011_2 = & (51_{10}) \\ \hline 11001100_2 & (204_{10}) \end{array}$$

80.4.4 Sottrazione binaria

La sottrazione binaria può essere eseguita nello stesso modo di quella che si utilizza nel sistema decimale. Come avviene nel sistema decimale, quando una cifra del minuendo (il numero di partenza) è minore della cifra corrispondente nel sottraendo (il numero da sottrarre), si prende a prestito una unità dalla cifra precedente (a sinistra), che così si somma al minuendo con il valore della base di numerazione. L'esempio seguente mostra una sottrazione con due numeri binari:

$$\begin{array}{rl} 10011001_2 - & (153_{10}) \\ 00110011_2 = & (51_{10}) \\ \hline 01100110_2 & (102_{10}) \end{array}$$

Generalmente, la sottrazione binaria viene eseguita sommando il complemento alla base del sottraendo. Il complemento alla base di 00110011_2 con otto cifre è 11001101_2:

$$\begin{array}{rl} 100000000_2 - & \\ 00110011_2 = & \\ \hline 11001101_2 & \end{array}$$

Pertanto, la sottrazione originale diventa una somma, dove si trascura il riporto:

$$\begin{array}{rll} 10011001_2 & + & (153_{10}) \\ 11001101_2 & = & \\ \hline 101100110_2 & - & \\ 100000000_2 & = & \\ \hline 01100110_2 & & (102_{10}) \end{array}$$

80.4.5 Moltiplicazione binaria

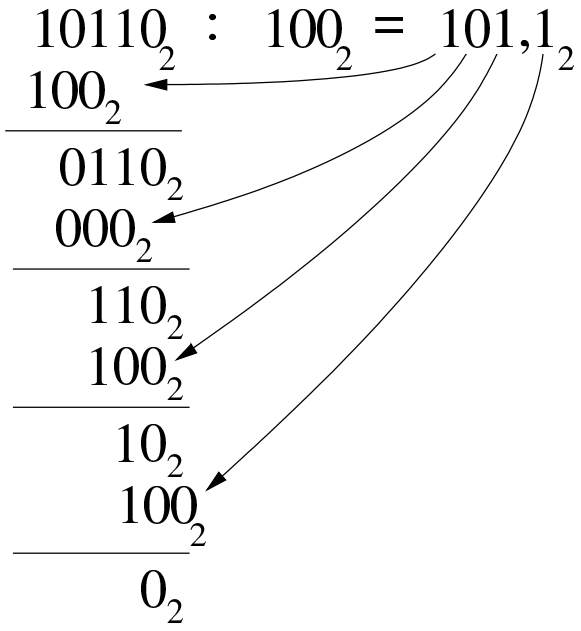
«

La moltiplicazione binaria si esegue in modo analogo a quella per il sistema decimale, con il vantaggio che è sufficiente sommare il moltiplicando, facendolo scorrere verso sinistra, in base al valore del moltiplicatore. Naturalmente, lo spostamento di un valore binario verso sinistra di ***n*** posizioni, corrisponde a moltiplicarlo per 2^n. Si osservi l'esempio seguente dove si moltiplica 10011001_2 per 1011_2:

$$\begin{array}{rll} 10011001_2 & \times & (153_{10}) \\ 1011_2 & = & (11_{10}) \\ \hline 10011001_2 & + & \\ 10011001_2 & + & \\ 00000000_2 & + & \\ 10011001_2 & = & \\ \hline 11010010011_2 & & (1683_{10}) \end{array}$$

80.4.6 Divisione binaria

La divisione binaria si esegue in modo analogo al procedimento per i valori in base dieci. Si osservi l'esempio seguente, dove si divide il numero 10110_2 (22_{10}) per 100_2 (4_{10}):



In questo caso il risultato è 101_2 (5_{10}), con il resto di 10_2 (2_{10}); ovvero $101,1_2$ ($5,5_{10}$).

Intuitivamente si comprende che: si prende il divisore, senza zeri anteriori, lo si fa scorrere a sinistra in modo da trovarsi allineato inizialmente con il dividendo; se la sottrazione può avere luogo, si scrive la cifra 1_2 nel risultato; si continua con gli scorrimenti e le sottrazioni; al termine, il valore residuo è il resto della divisione intera.

80.4.7 Rappresentazione binaria di numeri interi senza segno

La rappresentazione di un valore intero senza segno coincide normalmente con il valore binario contenuto nella variabile gestita dal-

l'elaboratore. Pertanto, una variabile della dimensione di 8 bit, può rappresentare valori da zero a 2^8-1:

00000000_2 (0_{10})
00000001_2 (1_{10})
00000010_2 (2_{10})
...
11111110_2 (254_{10})
11111111_2 (255_{10})

80.4.8 Rappresentazione binaria di numeri interi con segno

«

Attualmente, per rappresentare valori interi con segno (positivo o negativo), si utilizza il metodo del complemento alla base, ovvero del complemento a due, dove il primo bit indica sempre il segno. Attraverso questo metodo, per cambiare di segno a un valore è sufficiente calcolarne il complemento a due.

Per esempio, se si prende un valore positivo rappresentato in otto cifre binarie come 00010100_2, pari a $+20_{10}$, il complemento a due è: 11101100_2, pari a -20_{10} secondo questa convenzione. Per trasformare il valore negativo nel valore positivo corrispondente, basta calcolare nuovamente il complemento a due: da 11101100_2 si ottiene ancora 00010100_2 che è il valore positivo originario.

Con il complemento a due, disponendo di n cifre binarie, si possono rappresentare valori da $-2^{(n-1)}$ a $+2^{(n-1)}-1$ ed esiste un solo modo per rappresentare lo zero: quando tutte le cifre binarie sono pari a zero. Infatti, rimanendo nell'ipotesi di otto cifre binarie, il complemento a uno di 00000000_2 è 11111111_2, ma aggiungendo una unità per otte-

nere il complemento a due si ottiene di nuovo 00000000_2, perdendo il riporto.

Si osservi che il valore negativo più grande rappresentabile non può essere trasformato in un valore positivo corrispondente, perché si creerebbe un traboccamento. Per esempio, utilizzando sempre otto bit (segno incluso), il valore minimo che possa essere rappresentato è 1000000_2, pari a -128_{10}, ma se si calcola il complemento a due, si ottiene di nuovo lo stesso valore binario, che però non è valido. Infatti, il valore positivo massimo che si possa rappresentare in questo caso è solo $+127_{10}$.

Figura 80.80. Confronto tra due valori interi con segno.

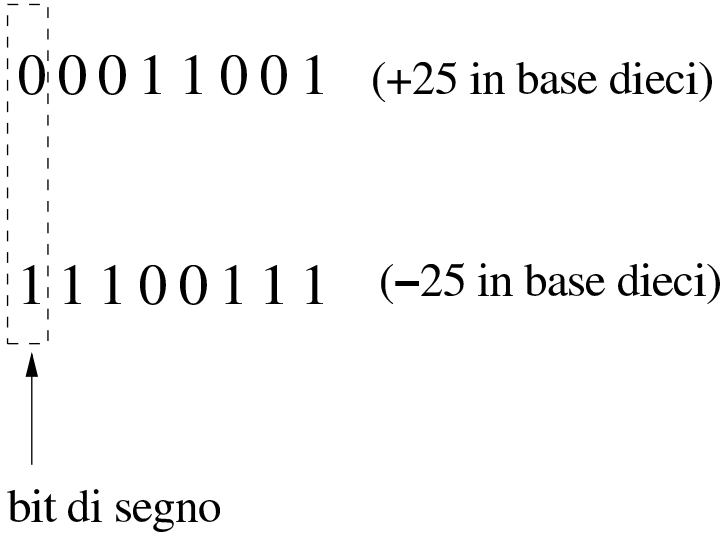
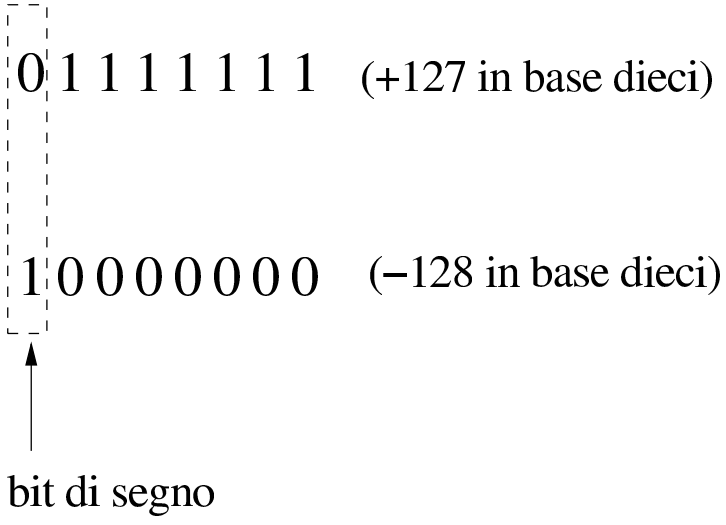


Figura 80.81. Valori massimi rappresentabili con soli otto bit.

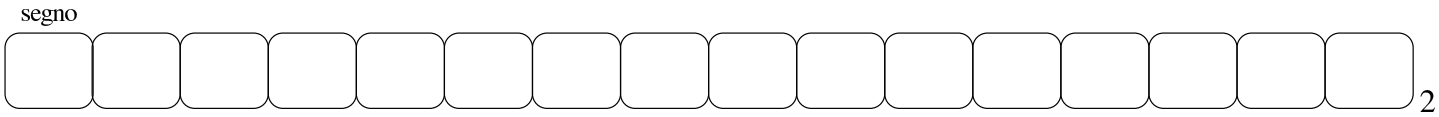


Il meccanismo del complemento a due ha il vantaggio di trasformare la sottrazione in una semplice somma algebrica.

80.4.8.1 Esercizio

«

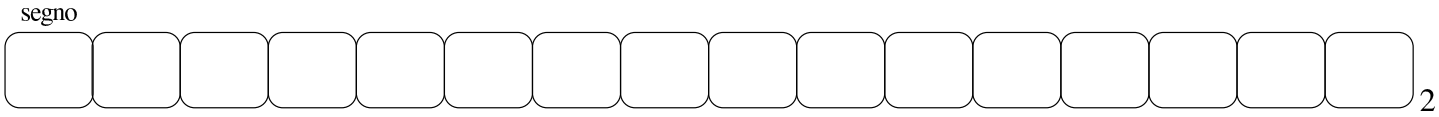
Come si rappresenta il numero $+103_{10}$, in una variabile binaria, a sedici cifre con segno?



80.4.8.2 Esercizio

«

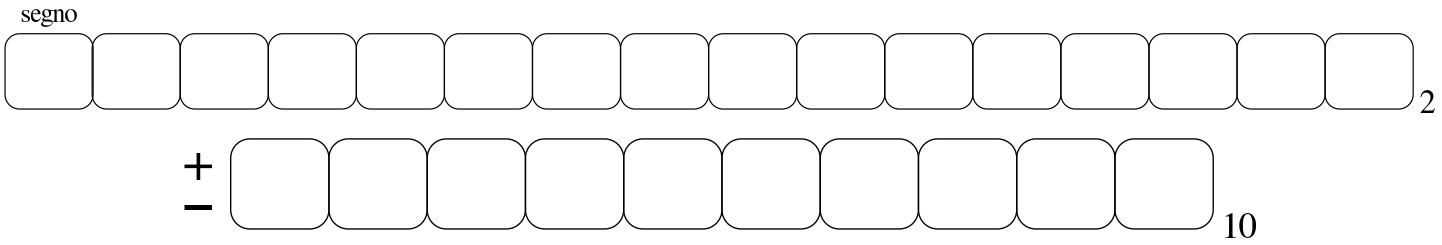
Come si rappresenta il numero -103_{10}, in una variabile binaria, a sedici cifre con segno?



80.4.8.3 Esercizio

«

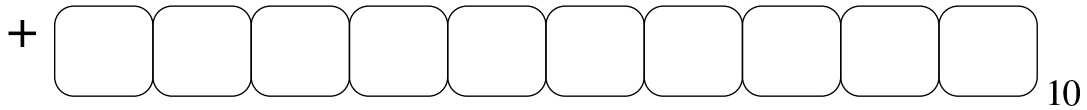
Data una variabile a sedici cifre che rappresenta un numero con segno, contenente il valore 1111111111110001_2, si calcoli il complemento a due e poi il valore corrispondente in base dieci, specificando il segno:



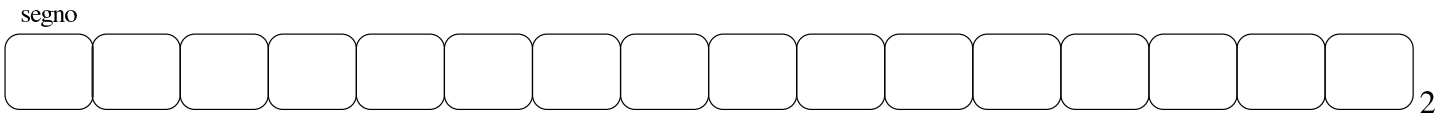
80.4.8.4 Esercizio

«

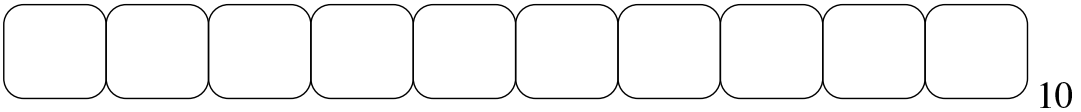
Data una variabile a sedici cifre che rappresenta un numero con segno, contenente il valore 0000000000110001_2, si calcoli il valore corrispondente in base dieci:



Si calcoli quindi il complemento a due:



Supponendo di interpretare il valore binario ottenuto dal complemento a due, come se si trattasse di un'informazione priva di segno, si calcoli nuovamente il valore corrispondente in base dieci:



80.4.8.5 Esercizio

«

Data una variabile a dodici cifre binarie che rappresenta un numero con segno, leggendo il suo contenuto come se fosse una variabile priva di segno, si potrebbe determinare quel segno originale in base al valore che si ottiene. Si scrivano gli intervalli che riguardano valori positivi e valori negativi:

Intervallo che rappresenta valori positivi	Intervallo che rappresenta valori negativi

80.4.8.6 Esercizio

«

Data una variabile a sedici cifre binarie che rappresenta un numero con segno, leggendo il suo contenuto come se fosse una variabile priva di segno, si potrebbe determinare quel segno originale in base al valore che si ottiene. Si scrivano gli intervalli che riguardano valori positivi e valori negativi:

Intervallo che rappresenta valori positivi	Intervallo che rappresenta valori negativi

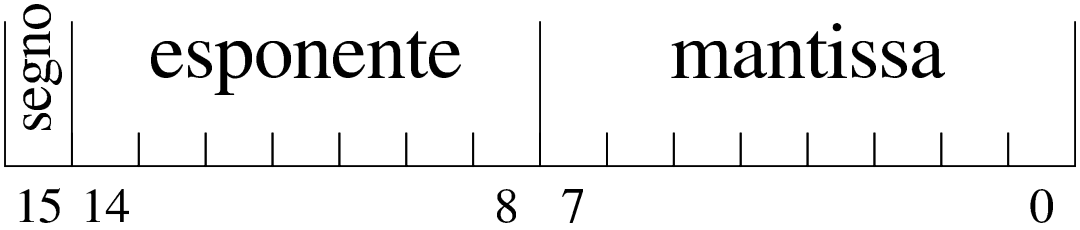
80.4.9 Cenni alla rappresentazione binaria di numeri in virgola mobile

Una forma diffusa per rappresentare dei valori molto grandi, consiste nell'indicare un numero con dei decimali moltiplicato per un valore costante elevato a un esponente intero. Per esempio, per rappresentare il numero 123000000 si potrebbe scrivere $123 \cdot 10^6$, oppure anche $0{,}123 \cdot 10^9$. Lo stesso ragionamento vale anche per valori molti piccoli; per esempio 0,000000123 che si potrebbe esprimere come $0{,}123 \cdot 10^{-6}$.

Per usare una notazione uniforme, si può convenire di indicare il numero che appare prima della moltiplicazione per la costante elevata a una certa potenza come un valore che più si avvicina all'unità, essendo minore o al massimo uguale a uno. Pertanto, per gli esempi già mostrati, si tratterebbe sempre di $0{,}123 \cdot 10^n$.

Per rappresentare valori a ***virgola mobile*** in modo binario, si usa un sistema simile, dove i bit a disposizione della variabile vengono suddivisi in tre parti: segno, esponente (di una base prestabilita) e mantissa, come nell'esempio che appare nella figura successiva.[1]

Figura 80.91. Ipotesi di una variabile a 16 bit per rappresentare dei numeri a virgola mobile.

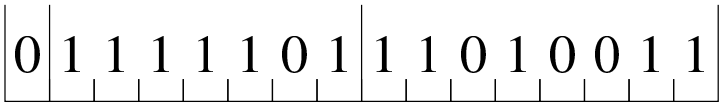


Nella figura si ipotizza la gestione di una variabile a 16 bit per la rappresentazione di valori a virgola mobile. Come si vede dallo schema, il bit più significativo della variabile viene utilizzato per rappresen-

tare il segno del numero; i sette bit successivi si usano per indicare l'esponente (con segno) e gli otto bit finali per la mantissa (senza segno perché indicato nel primo bit), ovvero il valore da moltiplicare per una certa costante elevata all'esponente.

Quello che manca da decidere è come deve essere interpretato il numero della mantissa e qual è il valore della costante da elevare all'esponente indicato. Sempre a titolo di esempio, si conviene che il valore indicato nella mantissa esprima precisamente «0,***mantissa***» e che la costante da elevare all'esponente indicato sia 16 (ovvero 2^4), che si traduce in pratica nello spostamento della virgola di quattro cifre binarie alla volta.[2]

Figura 80.92. Esempio di rappresentazione del numero 0,051513671875 ($211 \cdot 16^{-3}$), secondo le convenzioni stabilite. Si osservi che il valore dell'esponente è negativo ed è così rappresentato come complemento alla base (due) del valore assoluto relativo.



$$+211 \cdot 16^{-3}$$

0,00000000000011010011

Naturalmente, le convenzioni possono essere cambiate: per esempio il segno lo si può incorporare nella mantissa; si può rappresentare l'esponente attraverso un numero al quale deve essere sottratta una costante fissa; si può stabilire un valore diverso della costante da elevare all'esponente; si possono distribuire diversamente gli spazi assegnati all'esponente e alla mantissa.

80.5 Calcoli con i valori binari rappresentati nella forma usata negli elaboratori

Una volta chiarito il modo in cui si rappresentano comunemente i valori numerici elaborati da un microprocessore, in particolare per ciò che riguarda i valori negativi con il complemento a due, occorre conoscere in che modo si trattano o si possono trattare questi dati (indipendentemente dall'ordine dei byte usato).

80.5.1 Modifica della quantità di cifre di un numero binario intero

Un numero intero senza segno, espresso con una certa quantità di cifre, può essere trasformato in una quantità di cifre maggiore, aggiungendo degli zeri nella parte più significativa. Per esempio, il numero 0101_2 può essere trasformato in 00000101_2 senza cambiarne il valore. Nello stesso modo, si può fare una copia di un valore in un contenitore più piccolo, perdendo le cifre più significative, purché queste siano a zero, altrimenti il valore risultante sarebbe alterato.

Quando si ha a che fare con valori interi con segno, nel caso di valori positivi, l'estensione e la riduzione funzionano come per i valori senza segno, con la differenza che nella riduzione di cifre, la prima deve ancora rappresentare un segno positivo. Se invece si ha a che fare con valori negativi, l'aumento di cifre richiede l'aggiunta di cifre a uno nella parte più significativa, mentre la riduzione comporta l'eliminazione di cifre a uno nella parte più significativa, con il vincolo di mantenere inalterato il segno.

Figura 80.93. Aumento e riduzione delle cifre di un numero intero senza segno.

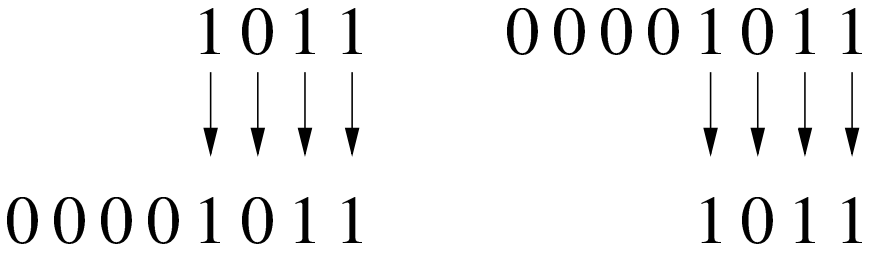


Figura 80.94. Aumento e riduzione delle cifre di un numero intero positivo.

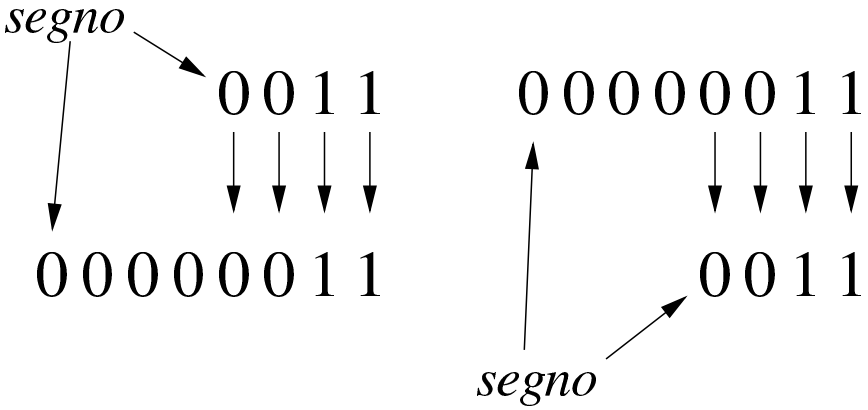
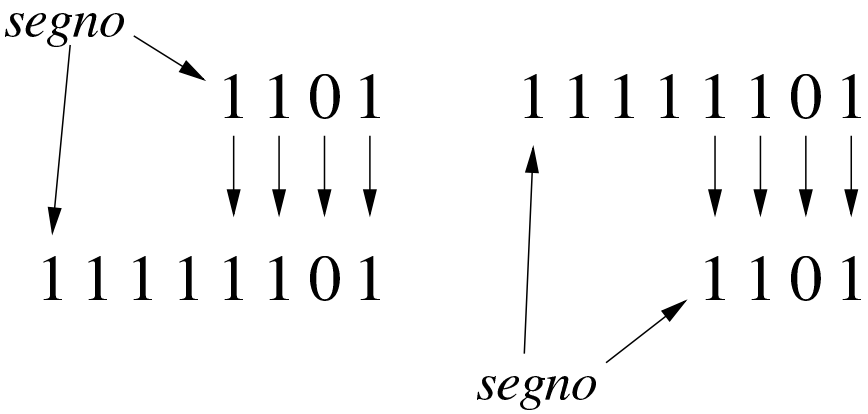


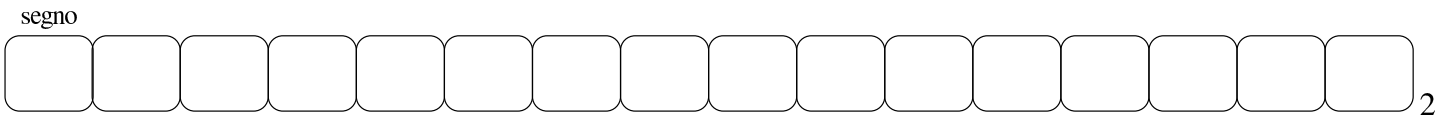
Figura 80.95. Aumento e riduzione delle cifre di un numero intero negativo.



80.5.1.1 Esercizio

«

Una variabile a otto cifre binarie, conviene un valore con segno, pari a 11100011_2. Questo valore viene copiato in una variabile a sedici cifre con segno. Indicare il valore che deve apparire nella variabile di destinazione.

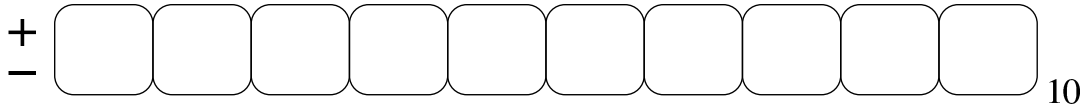


80.5.1.2 Esercizio

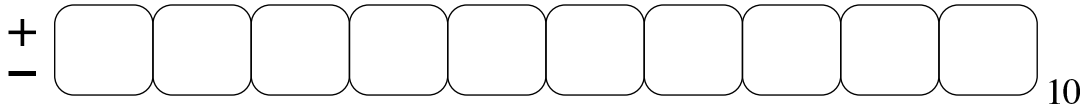
«

Una variabile a sedici cifre binarie, contiene un valore con segno, pari a 0000111110001111_2. Questo valore viene copiato in una variabile a otto cifre con segno. Il risultato della copia è valido? Perché?

Il valore originale della variabile a sedici cifre con segno è pari a:



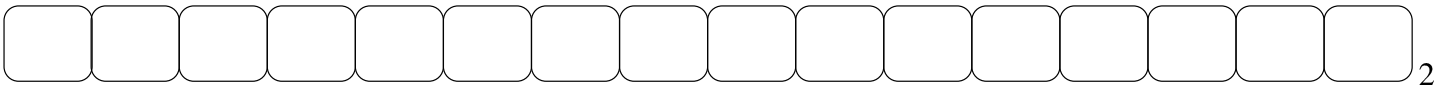
Il valore contenuto nella variabile a otto cifre con segno, potrebbe essere pari a:



80.5.1.3 Esercizio

«

Una variabile a otto cifre binarie, contiene un valore con segno, pari a 11100011_2. Questo valore viene copiato in una variabile a sedici cifre senza segno, ignorando il fatto che la variabile originale abbia un segno. Indicare il valore che appare nella variabile di destinazione dopo la copia.



Se successivamente si volesse considerare la variabile a sedici cifre usata per la destinazione della copia, come se fosse una variabile con segno, il valore che vi si potrebbe leggere al suo interno risulterebbe uguale a quello della variabile di origine?

80.5.2 Sommatorie con i valori interi con segno

Vengono proposti alcuni esempi che servono a dimostrare le situazioni che si presentano quando si sommano valori con segno, ricordando che i valori negativi sono rappresentati come complemento alla base del valore assoluto corrispondente.

Figura 80.100. Somma di due valori positivi che genera un risultato valido.

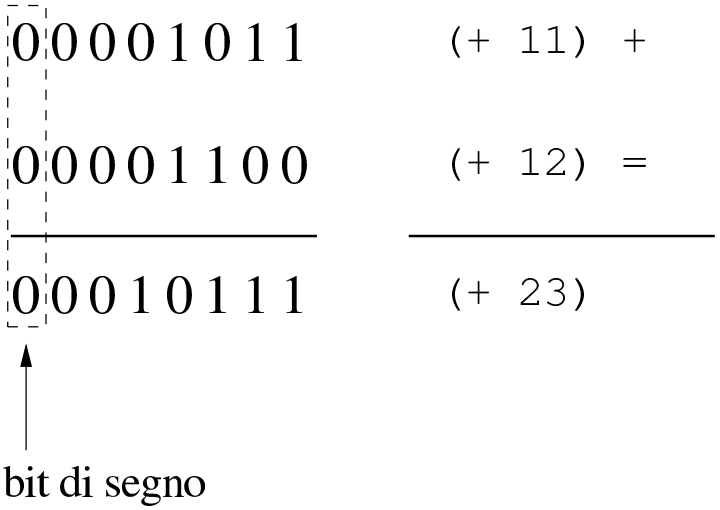


Figura 80.101. Somma di due valori positivi, dove il risultato apparentemente negativo indica la presenza di un traboccamento.

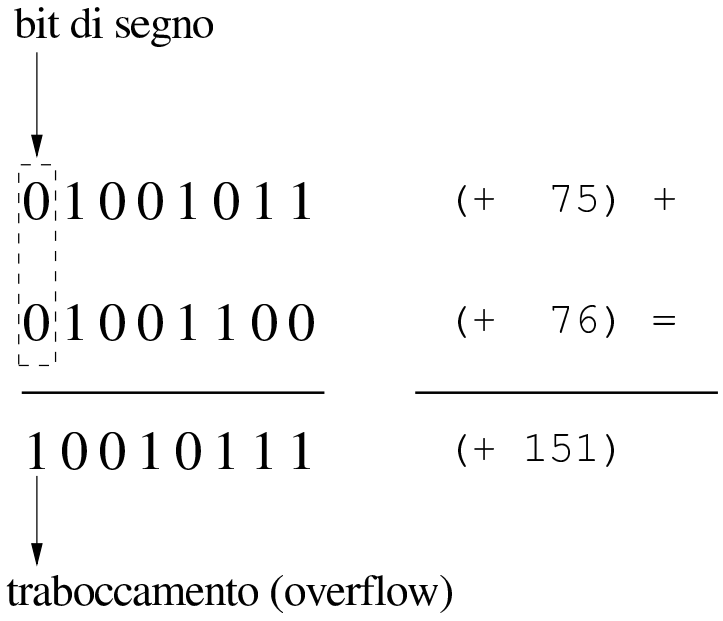


Figura 80.102. Somma di un valore positivo e di un valore negativo: il risultato è sempre valido.

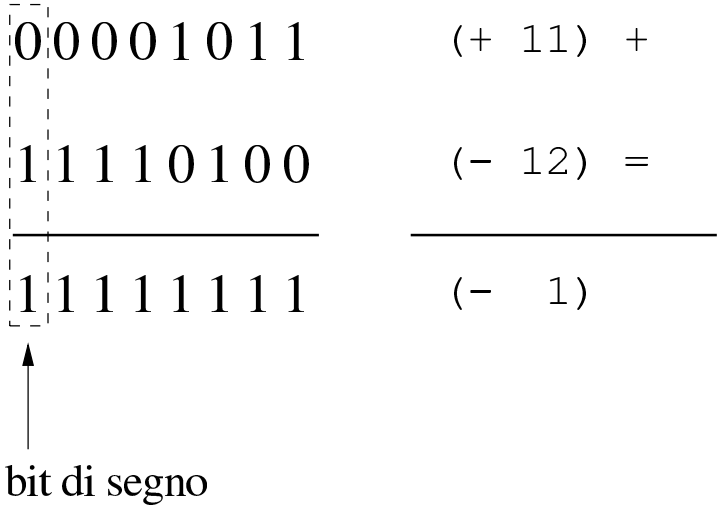


Figura 80.103. Somma di un valore positivo e di un valore negativo: in tal caso il risultato è sempre valido e se si manifesta un riporto, come in questo caso, va ignorato semplicemente.

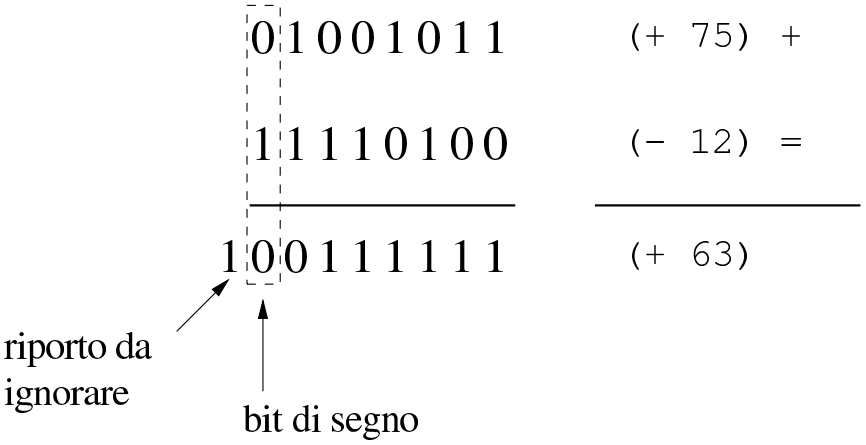


Figura 80.104. Somma di due valori negativi che produce un segno coerente e un riporto da ignorare.

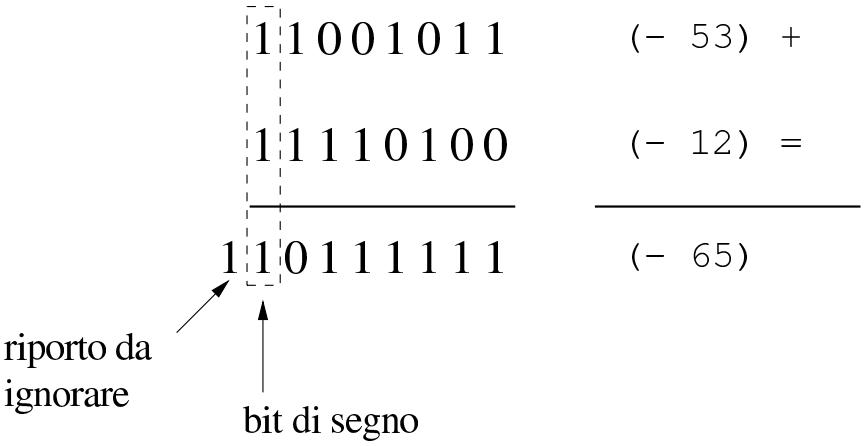
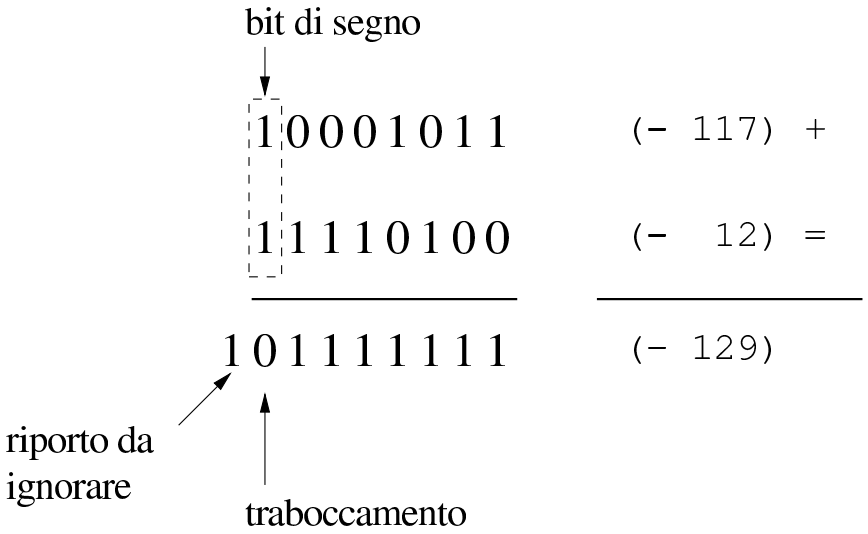


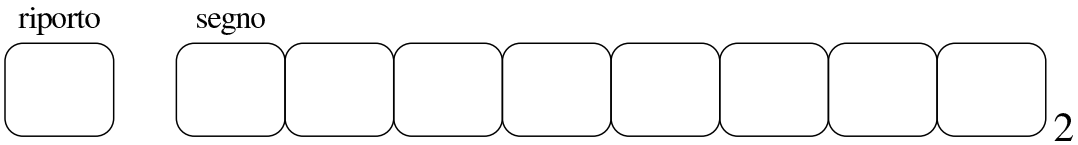
Figura 80.105. Somma di due valori negativi che genera un traboccamento, evidenziato da un risultato con un segno incoerente.



Dagli esempi mostrati si comprende facilmente che la somma di due valori con segno va fatta ignorando il riporto, perché quello che conta è che il segno risultante sia coerente: se si sommano due valori positivi, perché il risultato sia valido deve essere positivo; se si somma un valore positivo con uno negativo il risultato è sempre valido; se si sommano due valori negativi, perché il risultato sia valido deve rimanere negativo.

80.5.2.1 Esercizio

Si esegua la somma tra due valori binari a otto cifre con segno, indicando anche il riporto eventuale: $01010101_2 + 01111110_2$.

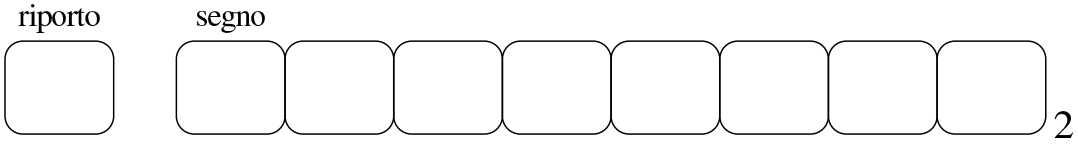


Il risultato della somma è valido?

80.5.2.2 Esercizio

«

Si esegua la somma tra due valori binari a otto cifre con segno, indicando anche il riporto eventuale: $11010101_2 + 01111110_2$.

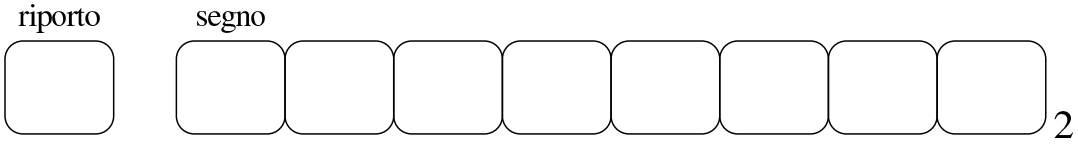


Il risultato della somma è valido?

80.5.2.3 Esercizio

«

Si esegua la somma tra due valori binari a otto cifre con segno, indicando anche il riporto eventuale: $11010101_2 + 10000001_2$.



Il risultato della somma è valido?

80.5.3 Somme e sottrazioni con i valori interi senza segno

«

La somma di due numeri interi senza segno avviene normalmente, senza dare un valore particolare al bit più significativo, pertanto, se si genera un riporto, il risultato non è valido (salva la possibilità di considerarlo assieme al riporto). Se invece si vuole eseguire una sottrazione, il valore da sottrarre va «invertito», con il complemento a due, ma sempre evitando di dare un significato particolare al bit più significativo. Il valore «normale» e quello «invertito» vanno sommati come al solito, ma **se il risultato non genera un riporto**, allora **è sbagliato**, in quanto il sottraendo è più grande del minuendo.

Per comprendere come funziona la sottrazione, si consideri di volere eseguire un'operazione molto semplice: $1-1$. Il minuendo (il primo

valore) sia espresso come 00000001_2; il sottraendo (il secondo valore) che sarebbe uguale, va trasformato attraverso il complemento a due, diventando così pari a 11111111_2. A questo punto si sommano algebricamente i due valori e si ottiene 00000000_2 con riporto di uno. Il riporto di uno dà la garanzia che il risultato è corretto. Volendo provare a sottrarre un valore più grande, si vede che il riporto non viene ottenuto: 1−2. In questo caso il minuendo si esprime come nell'esempio precedente, mentre il sottraendo è 00000010_2 che si trasforma nel complemento a due 11111110_2. Se si sommano i due valori si ottiene semplicemente 11111111_2, senza riporto, ma questo valore che va inteso senza segno è evidentemente errato.

Figura 80.109. Sottrazione tra due numeri interi senza segno, dove il sottraendo ha un valore assoluto minore di quello del minuendo: la presenza del riporto conferma la validità del risultato.

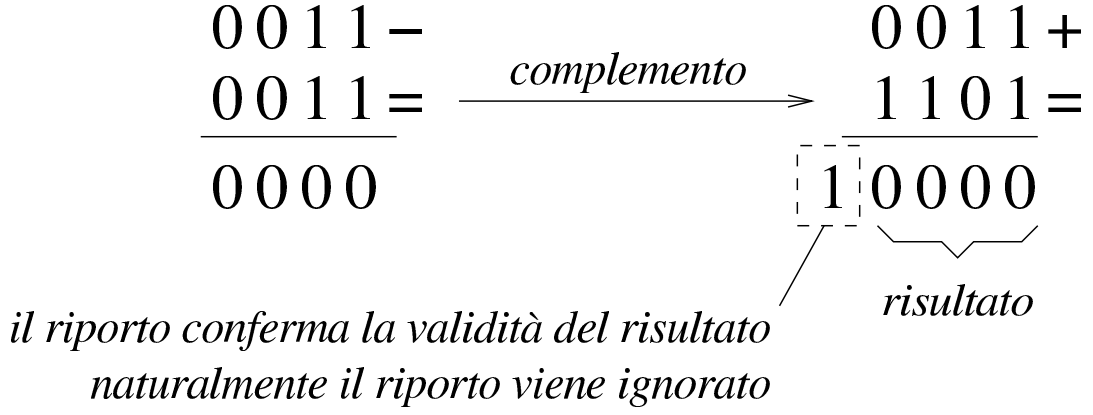
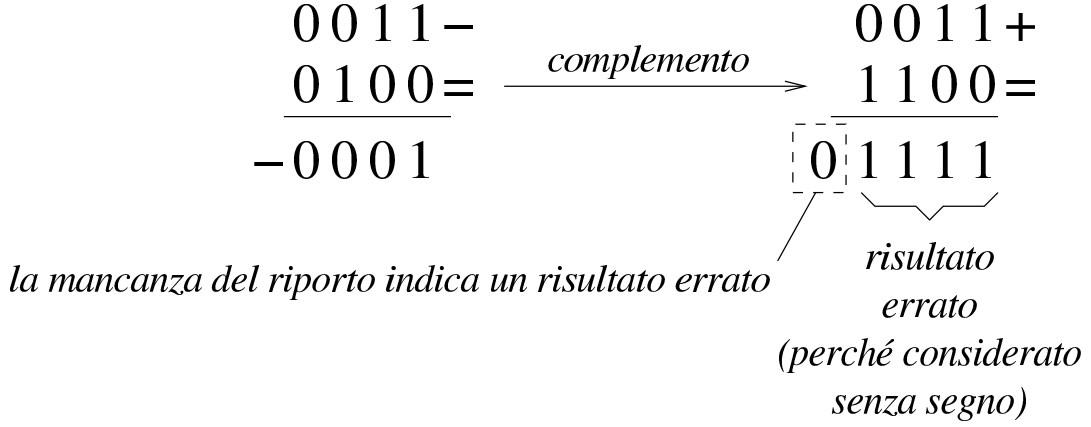
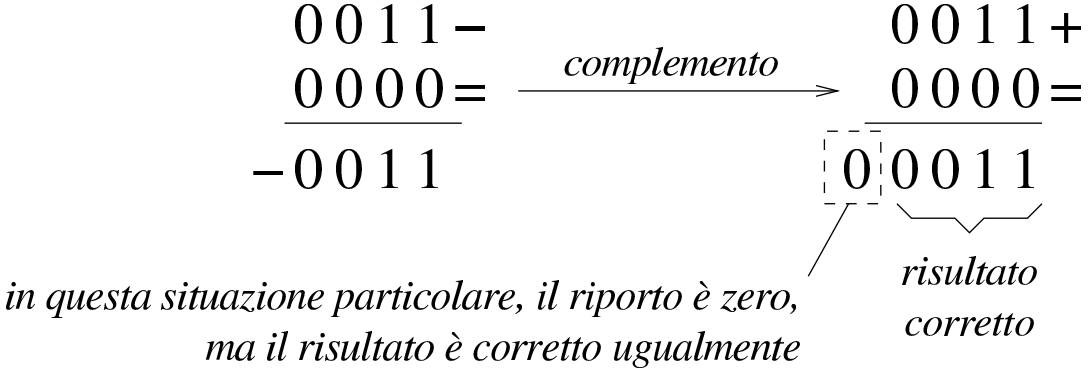


Figura 80.110. Sottrazione tra due numeri interi senza segno, dove il sottraendo ha un valore assoluto maggiore di quello del minuendo: l'assenza di un riporto indica un risultato errato della sottrazione.



Sulla base della spiegazione data, c'è però un problema, dovuto al fatto che il complemento a due di un valore a zero dà sempre zero: se si fa la sottrazione con il complemento, il risultato è comunque corretto, ma non si ottiene un riporto.

Figura 80.111. Sottrazione con sottraendo a zero: non si ottiene riporto, ma il risultato è corretto ugualmente.



Per correggere questo problema, il complemento a due del numero da sottrarre, va eseguito in due fasi: prima si calcola il complemento a uno, poi si somma il minuendo al sottraendo complementato, aggiungendo una unità ulteriore. Le figure successive ripetono gli esempi già mostrati, attuando questo procedimento differente.

Figura 80.112. Il complemento a due viene calcolato in due fasi: prima si calcola il complemento a uno, poi si sommano il minuendo e il sottraendo invertito, più una unità.

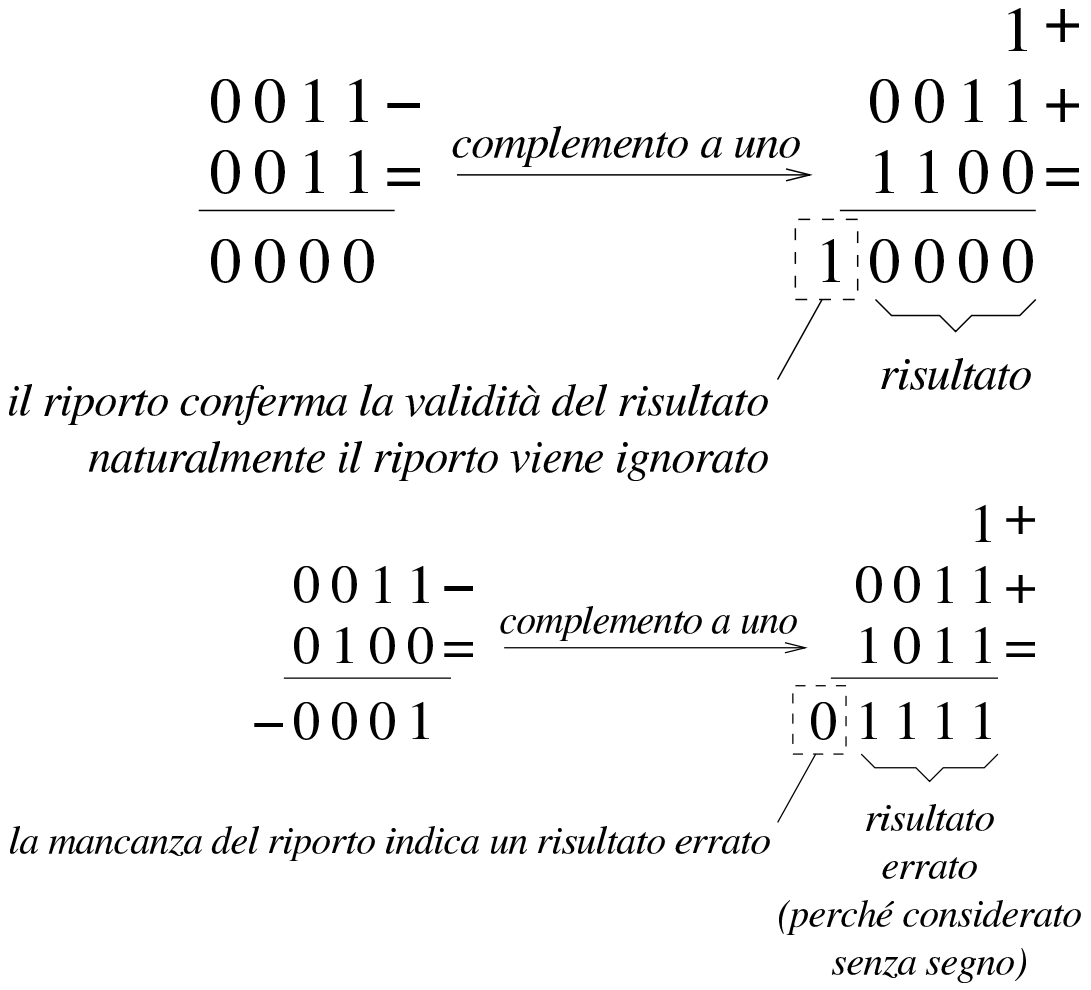
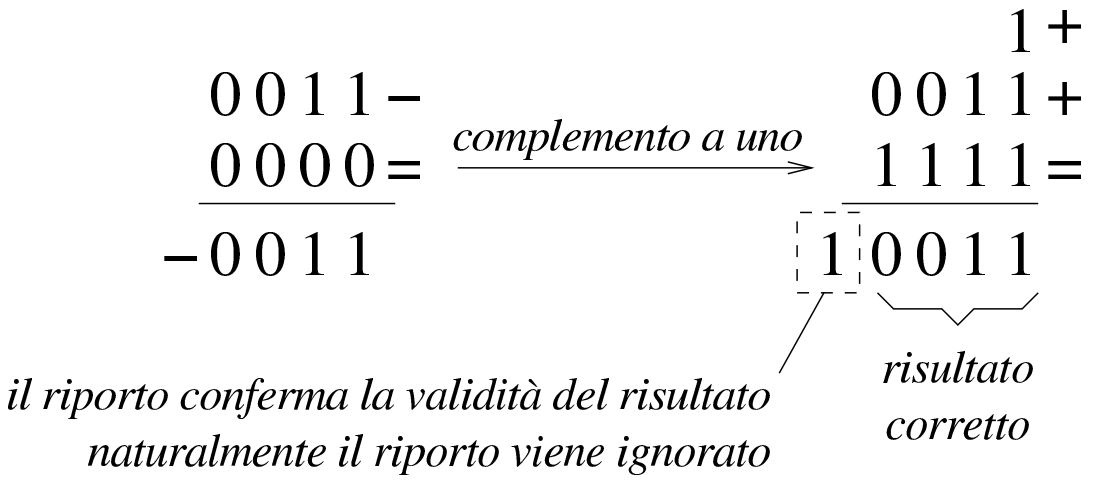


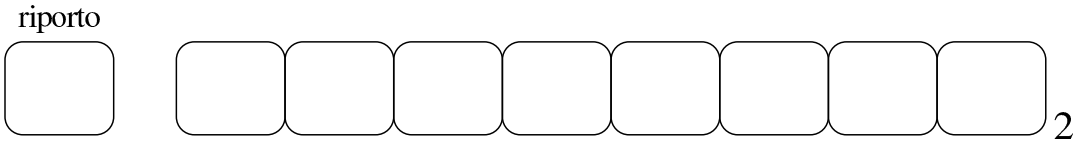
Figura 80.114. Sottrazione con sottraendo a zero: calcolando il complemento a due attraverso il complemento a uno, si ottiene un riporto coerente.



80.5.3.1 Esercizio

«

Si esegua la somma tra due valori binari a otto cifre senza segno, indicando anche il riporto eventuale: $11010101_2 + 01110110_2$.

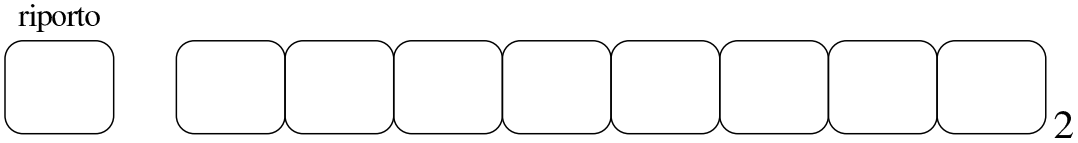


Il risultato della somma è valido?

80.5.3.2 Esercizio

«

Si esegua la somma tra due valori binari a otto cifre senza segno, indicando anche il riporto eventuale: $11010101_2 + 11110110_2$.

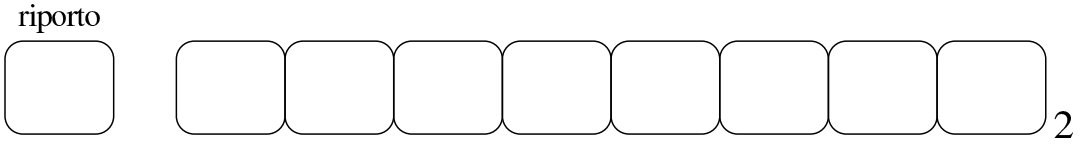


Il risultato della somma è valido?

80.5.3.3 Esercizio

«

Si esegua la sottrazione tra due valori binari a otto cifre senza segno, indicando anche il riporto eventuale: $11010101_2 - 11110110_2$.

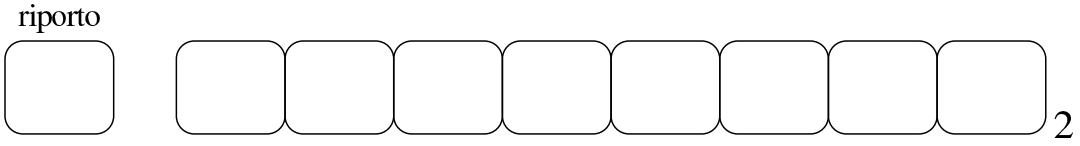


Il risultato della somma è valido?

80.5.3.4 Esercizio

«

Si esegua la sottrazione tra due valori binari a otto cifre senza segno, indicando anche il riporto eventuale: $11010101_2 - 00001111_2$.



Il risultato della sottrazione è valido?

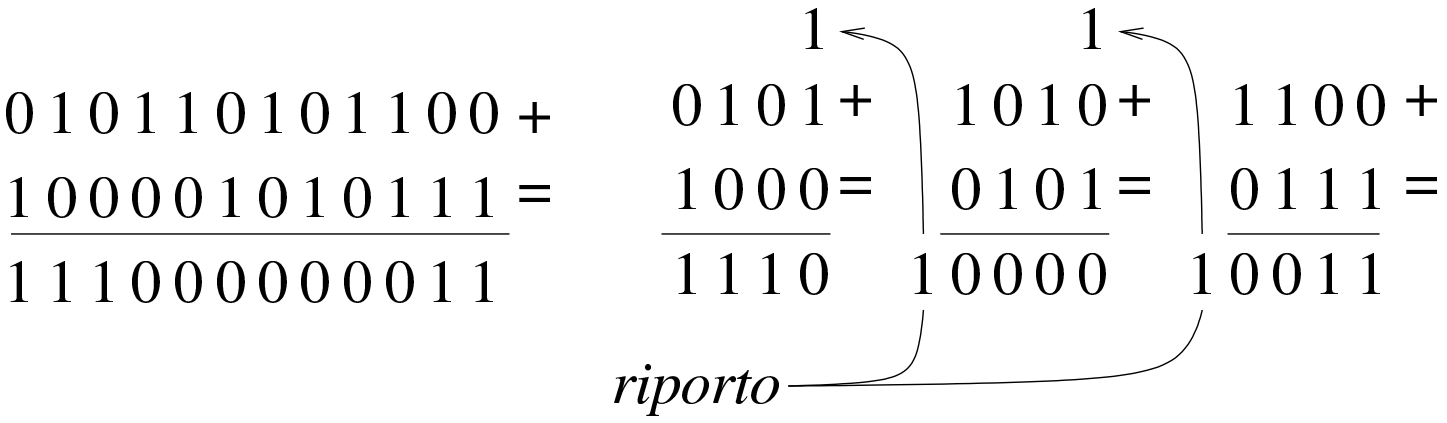
80.5.4 Somme e sottrazioni in fasi successive

«

Quando si possono eseguire somme e sottrazioni solo con una quantità limitata di cifre, mentre si vuole eseguire un calcolo con numeri più grandi della capacità consentita, si possono suddividere le operazioni in diverse fasi. La somma tra due numeri interi è molto semplice, perché ci si limita a tenere conto del riporto ottenuto nelle fasi precedenti. Per esempio, dovendo sommare $0101\ 1010\ 1100_2$ a $1000\ 0101\ 0111_2$ e potendo operare solo a gruppi di quattro bit per volta: si parte dal primo gruppo di bit meno significativo, 1100_2 e 0111_2, si sommano i due valori e si ottiene 0011_2 con riporto di uno; si prosegue sommando 1010_2 con 0101_2 aggiungendo il riporto e ottenendo 0000_2 con riporto di uno; si conclude sommando 0101_2 e

1000_2, aggiungendo il riporto della somma precedente e si ottiene così 1110_2. Quindi, il risultato è $1110\ 0000\ 0011_2$.

Figura 80.119. Somma per fasi successive, tenendo conto del riporto.



Nella sottrazione tra numeri senza segno, il sottraendo va trasformato secondo il complemento a due, quindi si esegue la somma e si considera che ci deve essere un riporto, altrimenti significa che il sottraendo è maggiore del minuendo. Quando si deve eseguire la sottrazione a gruppi di cifre più piccoli di quelli che richiede il valore per essere rappresentato, si può procedere in modo simile a quello che si usa con la somma, con la differenza che «l'assenza del riporto» indica la richiesta di prendere a prestito una cifra.

Per comprendere il procedimento è meglio partire da un esempio. In questo caso si utilizzano i valori già visti, ma invece di sommarli si vuole eseguire la sottrazione. Per la precisione, si intende prendere $1000\ 0101\ 0111_2$ come minuendo e $0101\ 1010\ 1100_2$ come sottraendo. Anche in questo caso si suppone di poter eseguire le operazioni solo a gruppi di quattro bit. Si esegue il complemento a due dei tre gruppetti di quattro bit del sottraendo, in modo indipendente, ottenendo: 1011_2, 0110_2, 0100_2. A questo punto si eseguono le somme, a partire dal gruppo meno significativo. La prima somma,

$0111_2 + 0100_2$, dà 1011_2, senza riporto, pertanto occorre prendere a prestito una cifra dal gruppo successivo: ciò significa che va eseguita la somma del gruppo successivo, sottraendo una unità dal risultato: $0101_2 + 0110_2 - 0001_2 = 1010_2$. Anche per il secondo gruppo non si ottiene il riporto della somma, così, anche dal terzo gruppo di bit occorre prendere a prestito una cifra: $1000_2 + 1011_2 - 0001_2 = 0010_2$. L'ultima volta la somma genera il riporto (da ignorare) che conferma la correttezza del risultato complessivo, ovvero che la sottrazione è avvenuta con successo.

Va però ricordato il problema legato allo zero, il cui complemento a due dà sempre zero. Se si cambiano i valori dell'esempio, lasciando come minuendo quello precedente, 1000 0101 0111_2, ma modificando il sottraendo in modo da avere le ultime quattro cifre a zero, 0101 1010 0000_2, il procedimento descritto non funziona più. Infatti, il complemento a due di 0000_2 rimane 0000_2 e se si somma questo a 0111_2 si ottiene lo stesso valore, ma senza riporti. In questo caso, nonostante l'assenza del riporto, il gruppo dei quattro bit successivi, del sottraendo, va trasformato con il complemento a due, senza togliere l'unità che sarebbe prevista secondo l'esempio precedente. In pratica, per poter eseguire la sottrazione per fasi successive, occorre definire un concetto diverso: il prestito (*borrow*) che non deve scattare quando si sottrae un valore pari a zero.

Se il complemento a due viene ottenuto passando per il complemento a uno, con l'aggiunta di una cifra, si può spiegare in modo più semplice il procedimento della sottrazione per fasi successive: invece di calcolare il complemento a due dei vari tronconi, si calcola semplicemente il complemento a uno e al gruppo meno significativo si aggiunge una unità per ottenere lì l'equivalente di un complemento

a due. Successivamente, il riporto delle somme eseguite va aggiunto al gruppo adiacente più significativo, come si farebbe con la somma: se la sottrazione del gruppo precedente non ha bisogno del prestito di una cifra, si ottiene l'aggiunta una unità al gruppo successivo.

Figura 80.120. Sottrazione per fasi successive, tenendo conto del prestito delle cifre.

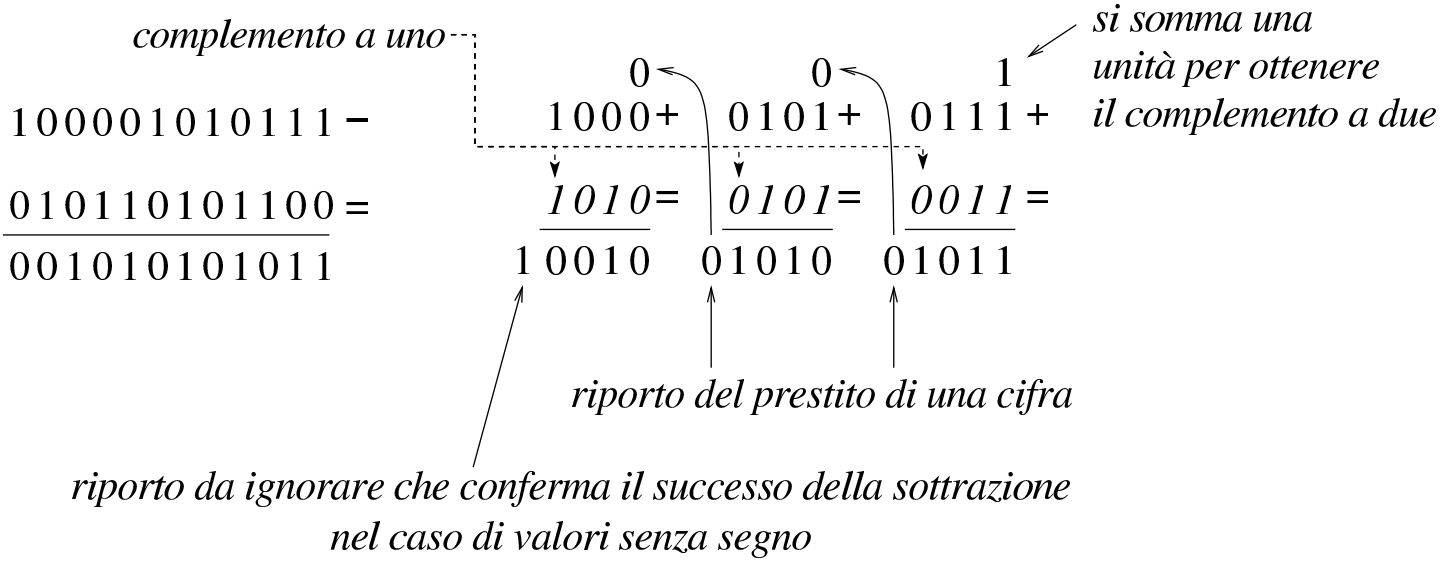
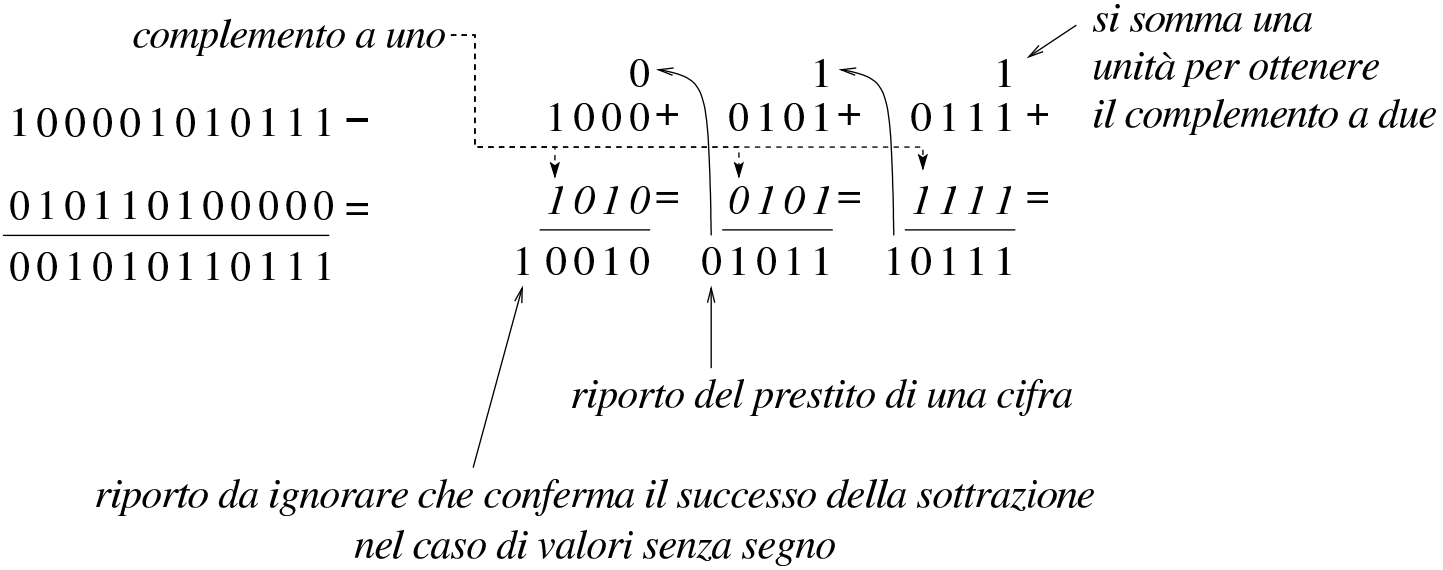


Figura 80.121. Verifica del procedimento anche in presenza di un sottraendo a zero.



La sottrazione per fasi successive funziona anche con valori che,

complessivamente, hanno un segno. L'unica differenza sta nel modo di valutare il risultato complessivo: l'ultimo gruppo di cifre a essere considerato (quello più significativo) è quello che contiene il segno ed è il segno del risultato che deve essere coerente, per stabilire se ciò che si è ottenuto è valido. Pertanto, nel caso di valori con segno, il riporto finale si ignora, esattamente come si fa quando la sottrazione avviene in una fase sola, mentre l'esistenza o meno del traboccamento deriva dal confronto della cifra più significativa: se la sottrazione, dopo la trasformazione in somma con il complemento, implica la somma valori con lo stesso segno, il risultato deve ancora avere quel segno, altrimenti c'è il traboccamento.

Se si volessero considerare gli ultimi due esempi come la sottrazione di valori con segno, il minuendo si intenderebbe un valore negativo, mentre il sottraendo sarebbe un valore positivo. Attraverso il complemento si passa alla somma di due valori negativi, ma dal momento che si ottiene un risultato con segno positivo, ciò manifesta un traboccamento, ovvero un risultato errato, perché non contenibile nello spazio disponibile.

80.6 Scorrimenti, rotazioni, operazioni logiche

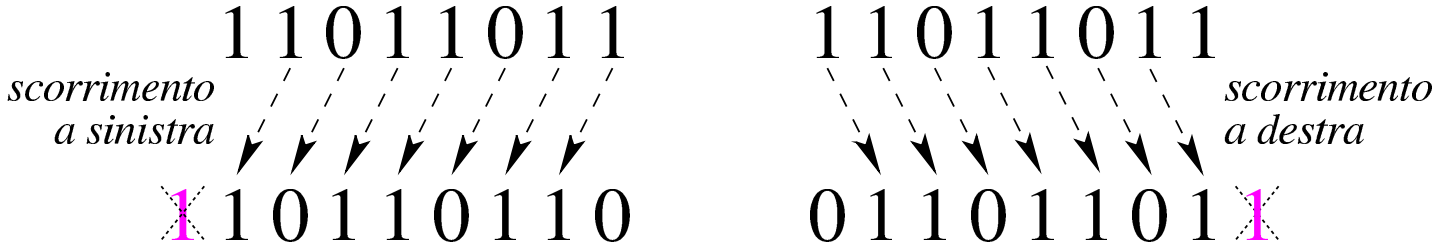
Le operazioni più semplici che si possono compiere con un microprocessore sono quelle che riguardano la logica booleana e lo scorrimento dei bit. Proprio per la loro semplicità è importante conoscere alcune applicazioni interessanti di questi procedimenti elaborativi.

80.6.1 Scorrimento logico

«

Lo scorrimento «logico» consiste nel fare scalare le cifre di un numero binario, verso sinistra (verso la parte più significativa) o verso destra (verso la parte meno significativa). Nell'eseguire questo scorrimento, da un lato si perde una cifra, mentre dall'altro si acquista uno zero.

Figura 80.122. Scorrimento logico a sinistra, perdendo le cifre più significative e scorrimento logico a destra, perdendo le cifre meno significative.

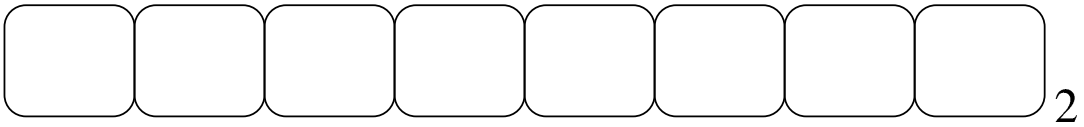


Lo scorrimento di una posizione verso sinistra corrisponde alla moltiplicazione del valore per due, mentre lo scorrimento a destra corrisponde a una divisione intera per due; scorrimenti di n posizioni rappresentano moltiplicazioni e divisioni per 2^n. Le cifre che si perdono nello scorrimento a sinistra si possono considerare come il riporto della moltiplicazione, mentre le cifre che si perdono nello scorrimento a destra sono il resto della divisione.

80.6.1.1 Esercizio

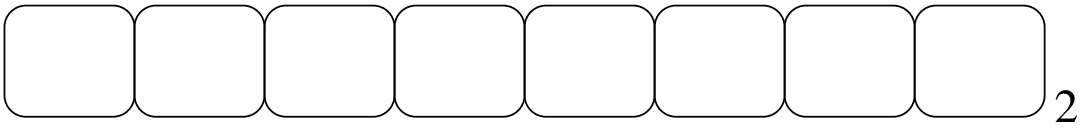
«

Si esegua lo scorrimento logico a sinistra (di una sola cifra) del valore 11010101_2.



80.6.1.2 Esercizio

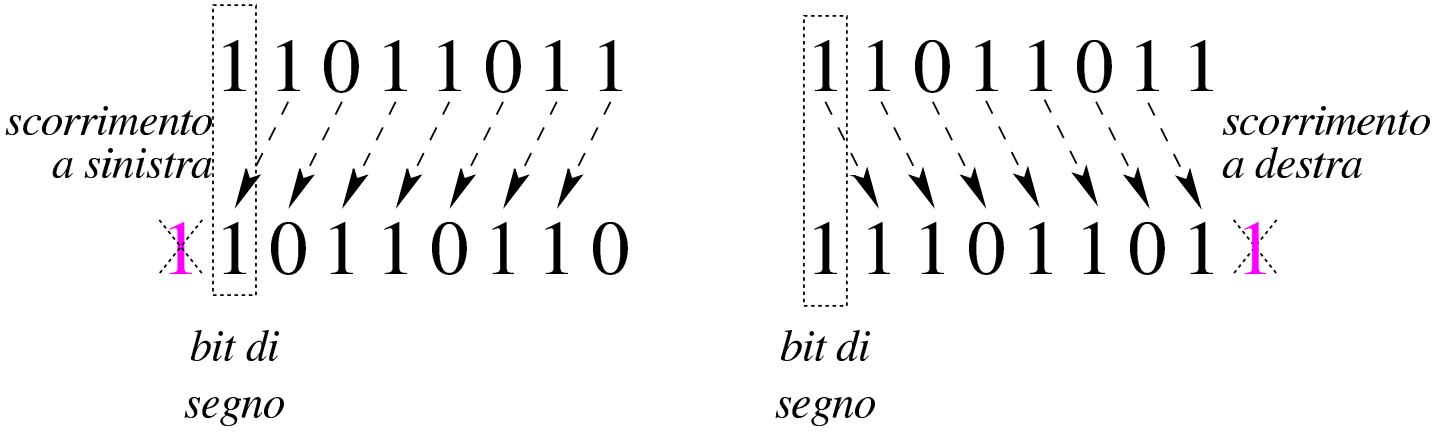
Si esegua lo scorrimento logico a destra (di una sola cifra) del valore 11010101_2.



80.6.2 Scorrimento aritmetico

Il tipo di scorrimento descritto nella sezione precedente, se utilizzato per eseguire moltiplicazioni e divisioni, va bene solo per valori senza segno. Se si intende fare lo scorrimento di un valore con segno, occorre distinguere due casi: lo scorrimento a sinistra è valido se il risultato non cambia di segno; lo scorrimento a destra implica il mantenimento del bit che rappresenta il segno e l'aggiunta di cifre uguali a quella che rappresenta il segno stesso.

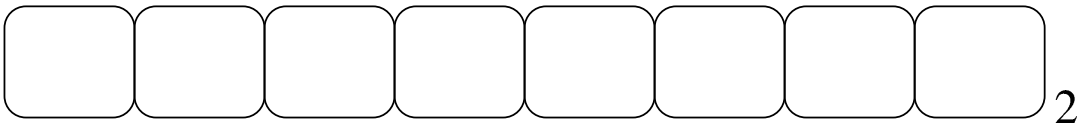
Figura 80.125. Scorrimento aritmetico a sinistra e a destra, di un valore negativo.



80.6.2.1 Esercizio

«

Si esegua lo scorrimento aritmetico a sinistra (di una sola cifra) del valore con segno 01010101_2.

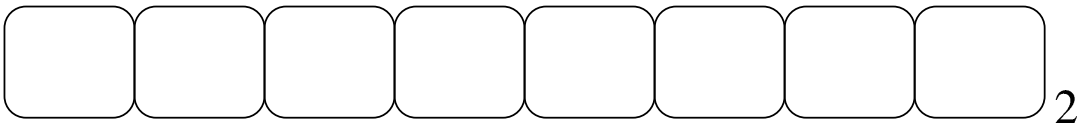


Il risultato dello scorrimento è valido?

80.6.2.2 Esercizio

«

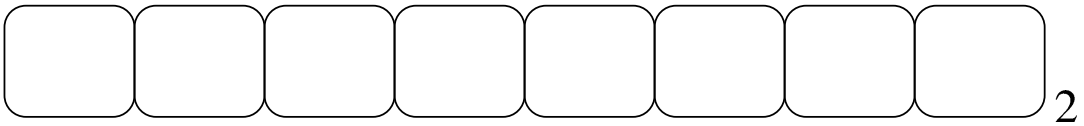
Si esegua lo scorrimento aritmetico a destra (di una sola cifra) del valore con segno 01010101_2.



80.6.2.3 Esercizio

«

Si esegua lo scorrimento aritmetico a destra (di una sola cifra) del valore con segno 11010101_2.



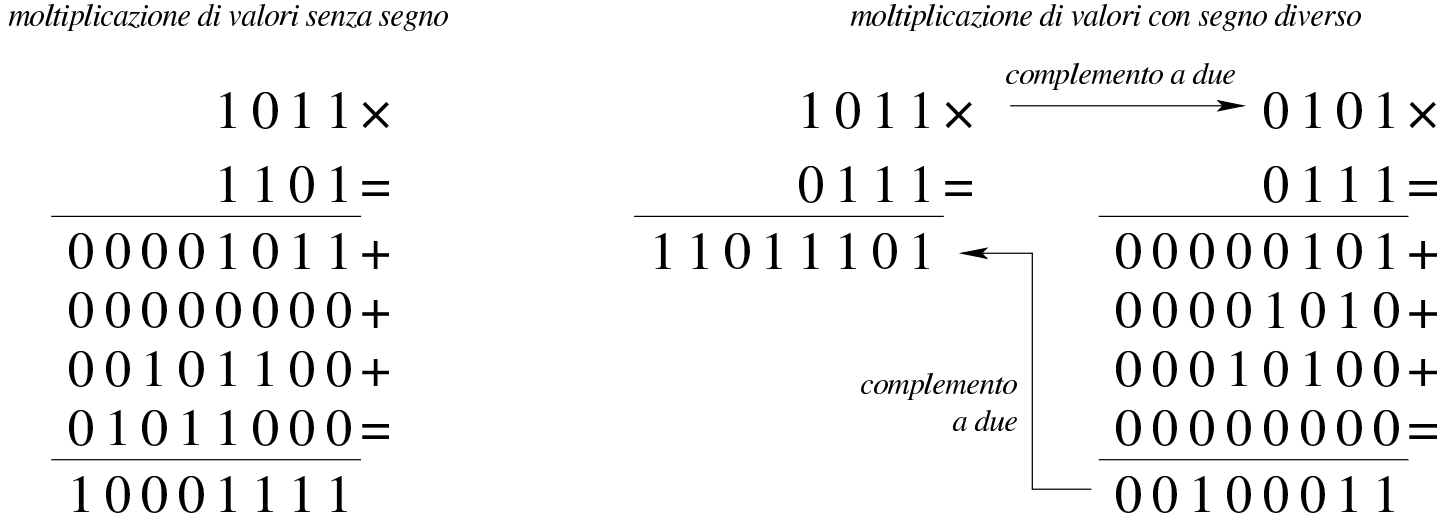
80.6.3 Moltiplicazione

«

La moltiplicazione si ottiene attraverso diverse fasi di scorrimento e somma di un valore, dove però il risultato richiede un numero doppio di cifre rispetto a quelle usate per il moltiplicando e il moltiplicatore. Il procedimento di moltiplicazione deve avvenire sempre con valori senza segno. Se i valori si intendono con segno, quando sono negativi occorre farne prima il complemento a due, in modo da portarli a valori positivi, quindi occorre decidere se il risultato va preso così

come viene o se va invertito a sua volta con il complemento a due: se i valori moltiplicati hanno segno diverso tra loro, il risultato deve essere trasformato con il complemento a due per renderlo negativo, altrimenti il risultato è sempre positivo.

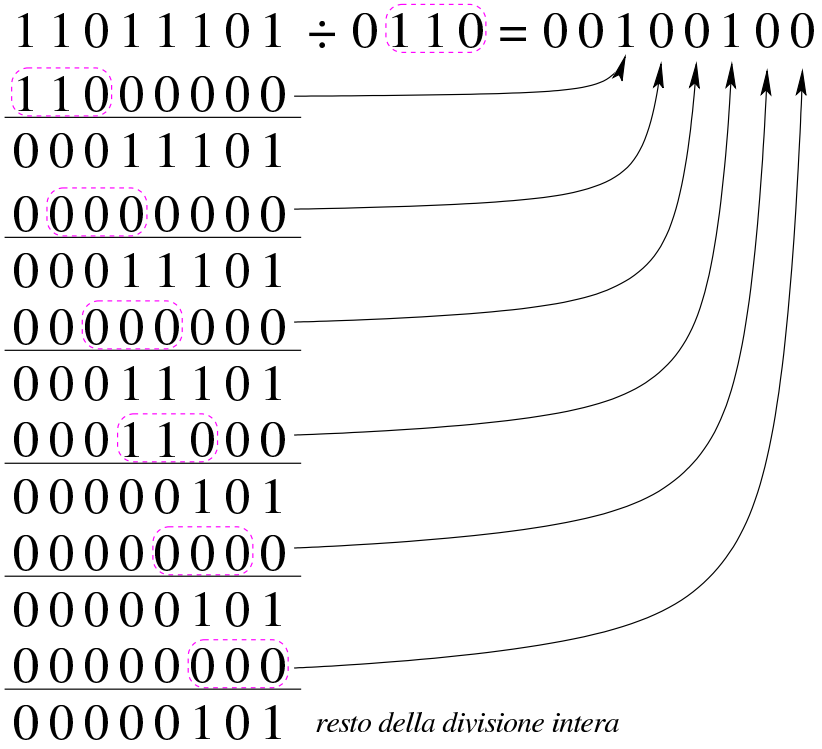
Figura 80.129. Moltiplicazione.



80.6.4 Divisione

La divisione si ottiene attraverso diverse fasi di scorrimento di un valore, che di volta in volta viene sottratto al dividendo, ma solo se la sottrazione è possibile effettivamente. Il procedimento di divisione deve avvenire sempre con valori senza segno. Se i valori si intendono con segno, quando sono negativi occorre farne prima il complemento a due, in modo da portarli a valori positivi, quindi occorre decidere se il risultato va preso così come viene o se va invertito a sua volta con il complemento a due: se dividendo e divisore hanno segni diversi tra loro, il risultato deve essere trasformato con il complemento a due per renderlo negativo, altrimenti il risultato è sempre positivo.

Figura 80.130. Divisione: i valori sono intesi senza segno.



221 : 6 = 36
con il resto di 5

80.6.5 Rotazione

«

La rotazione è uno scorrimento dove le cifre che si perdono da una parte rientrano dall'altra. Esistono due tipi di rotazione; uno «normale» e l'altro che include nella rotazione il bit del riporto. Dal momento che la rotazione non si presta per i calcoli matematici, di solito non viene considerato il segno.

Figura 80.131. Rotazione normale.

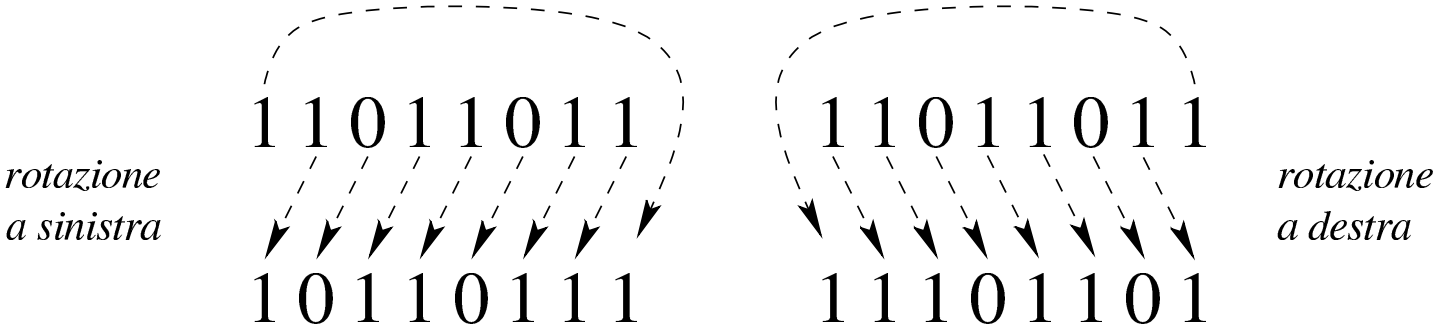
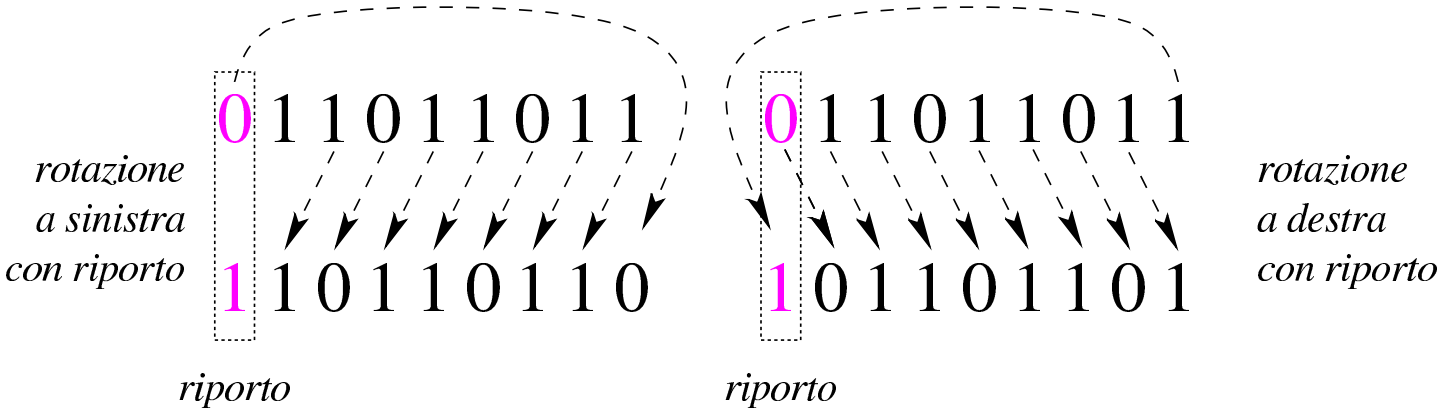


Figura 80.132. Rotazione con riporto.



80.6.6 Operatori logici

Gli operatori logici si possono applicare anche a valori composti da più cifre binarie.

Figura 80.133. AND e OR.

1 1 0 1 1 0 1 1	*a*	1 1 0 1 1 0 1 1	*a*
0 1 1 0 1 1 0 1	*b*	0 1 1 0 1 1 0 1	*b*
0 1 0 0 1 0 0 1	*a AND b*	1 1 1 1 1 1 1 1	*a OR b*

Figura 80.134. XOR e NOT.

1 1 0 1 1 0 1 1	*a*	1 1 0 1 1 0 1 1	*a*
0 1 1 0 1 1 0 1	*b*		
1 0 1 1 0 1 1 0	*a XOR b*	0 0 1 0 0 1 0 0	*NOT a*

È importante osservare che l'operatore NOT esegue in pratica il complemento a uno di un valore.

Capita spesso di trovare in un sorgente scritto in un linguaggio assemblatore un'istruzione che assegna a un registro il risultato dell'operatore XOR su se stesso. Ciò si fa, evidentemente, per azzerarne

il contenuto, quando, probabilmente, l'assegnamento esplicito di un valore a un registro richiede una frazione di tempo maggiore per la sua esecuzione.

Figura 80.135. XOR per azzerare i valori.

$$\begin{array}{ll} 11011011 & a \\ 11011011 & a \\ \hline 00000000 & a\ XOR\ a \end{array}$$

80.6.6.1 Esercizio

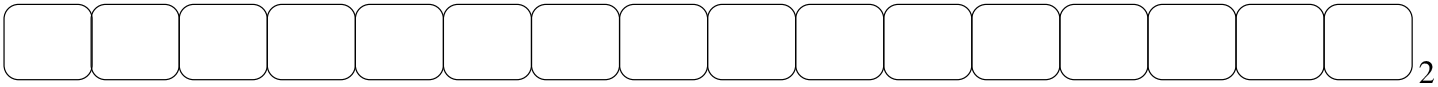
«

Eseguire l'operazione seguente, considerando i valori privi di segno: 0010010101011111_2 AND 0110001111000011_2.

80.6.6.2 Esercizio

«

Eseguire l'operazione seguente, considerando i valori privi di segno: 0010010101011111_2 OR 0110001111000011_2.



80.6.6.3 Esercizio

«

Eseguire l'operazione seguente, considerando i valori privi di segno: 0010010101011111_2 XOR 0110001111000011_2.

80.6.6.4 Esercizio

«

Eseguire l'operazione seguente, considerando i valori privi di segno: NOT 0010010101011111_2.

80.7 Organizzazione della memoria

«

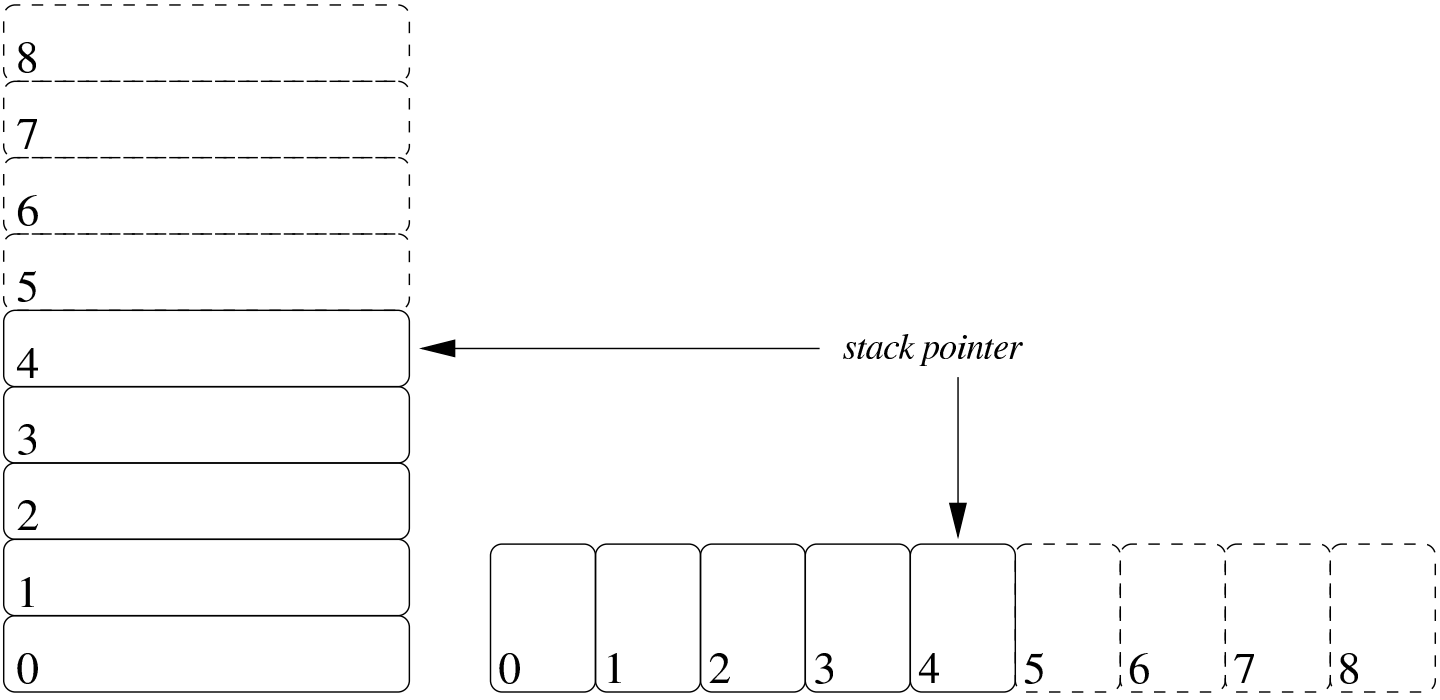
Nello studio del linguaggio C è importante avere un'idea di come venga gestita la memoria di un elaboratore, molto vicina a ciò che si percepirebbe usando direttamente il linguaggio della CPU.

80.7.1 Pila per salvare i dati

«

Quando si scrive con un linguaggio di programmazione molto vicino a quello effettivo del microprocessore, si ha normalmente a disposizione una pila di elementi omogenei (*stack*), usata per accumulare temporaneamente delle informazioni, da espellere poi in senso inverso. Questa pila è gestita attraverso un vettore, dove l'ultimo elemento (quello superiore) è individuato attraverso un indice noto come *stack pointer* e tutti gli elementi della pila sono comunque accessibili, in lettura e in sovrascrittura, se si conosce la loro posizione relativa.

Figura 80.140. Esempio di una pila che può contenere al massimo nove elementi, rappresentata nel modo tradizionale, oppure distesa, come si fa per i vettori. Gli elementi che si trovano oltre l'indice (lo *stack pointer*) non sono più disponibili, mentre gli altri possono essere letti e modificati senza doverli estrarre dalla pila.



Per accumulare un dato nella pila (*push*) si incrementa di una unità l'indice e lo si inserisce in quel nuovo elemento. Per estrarre l'ultimo elemento dalla pila (*pop*) si legge il contenuto di quello corrispondente all'indice e si decrementa l'indice di una unità.

80.7.2 Chiamate di funzioni

«

I linguaggi di programmazione più vicini alla realtà fisica della memoria di un elaboratore, possono disporre solo di variabili globali ed eventualmente di una pila, realizzata attraverso un vettore, come descritto nella sezione precedente. In questa situazione, la chiamata di una funzione può avvenire solo passando i parametri in uno spazio

di memoria condiviso da tutto il programma. Ma per poter generalizzare le funzioni e per consentire la ricorsione, ovvero per rendere le funzioni ***rientranti***, il passaggio dei parametri deve avvenire attraverso la pila in questione.

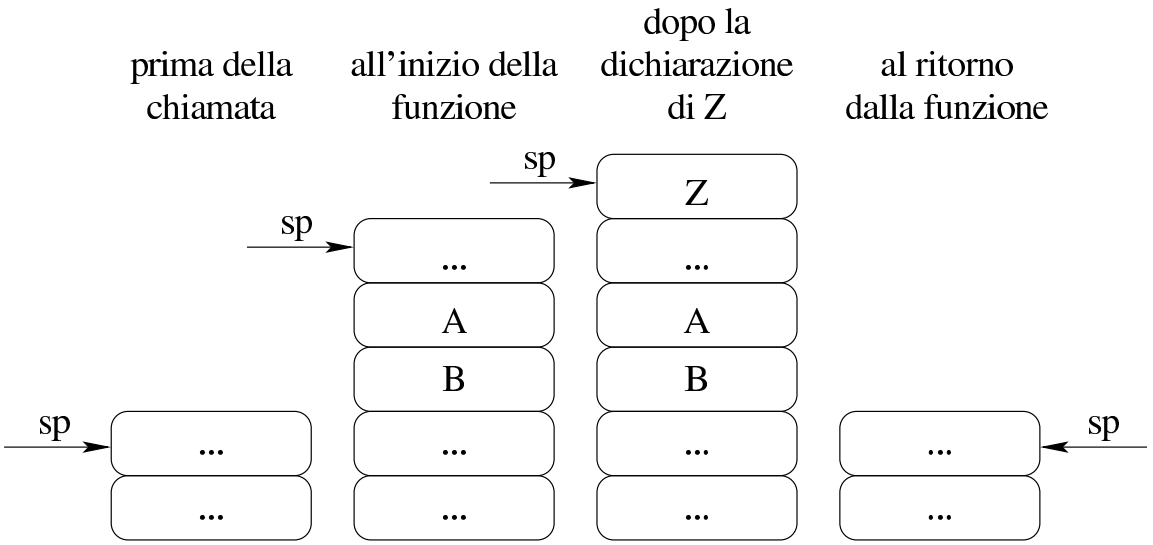
Per mostrare un esempio che consenta di comprendere il meccanismo, si può osservare l'esempio seguente, schematizzato attraverso una pseudocodifica: la funzione '**SOMMA**' prevede l'uso di due parametri (ovvero due argomenti nella chiamata) e di una variabile «locale». Per chiamare la funzione, occorre mettere i valori dei parametri nella pila; successivamente, si dichiara la stessa variabile locale nella pila. Si consideri che il programma inizia e finisce nella funzione '**MAIN**', all'interno della quale si fa la chiamata della funzione '**SOMMA**':

```
SOMMA (X, Y)
    LOCAL Z INTEGER
    Z := X + Y
    RETURN Z
END SOMMA

MAIN ()
    LOCAL A INTEGER
    LOCAL B INTEGER
    LOCAL C INTEGER
    A := 3
    B := 4
    C := SOMMA (A, B)
END MAIN
```

Nel disegno successivo, si schematizza ciò che accade nella pila (nel vettore che rappresenta la pila dei dati), dove si vede che inizialmente c'è una situazione indefinita, con l'indice «sp» (*stack pointer*) in

una certa posizione. Quando viene eseguita la chiamata della funzione, automaticamente si incrementa la pila inserendo gli argomenti della chiamata (qui si mettono in ordine inverso, come si fa nel linguaggio C), mettendo in cima anche altre informazioni che nello schema non vengono chiarite (nel disegno appare un elemento con dei puntini di sospensione).

Figura 80.142. Situazione della pila nelle varie fasi della chiamata della funzione '**SOMMA**'.



La variabile locale «Z» viene allocata in cima alla pila, incrementando ulteriormente l'indice «sp». Al termine, la funzione trasmette in qualche modo il proprio risultato (tale modalità non viene chiarita qui e dipende dalle convenzioni di chiamata) e la pila viene riportata alla sua condizione iniziale.

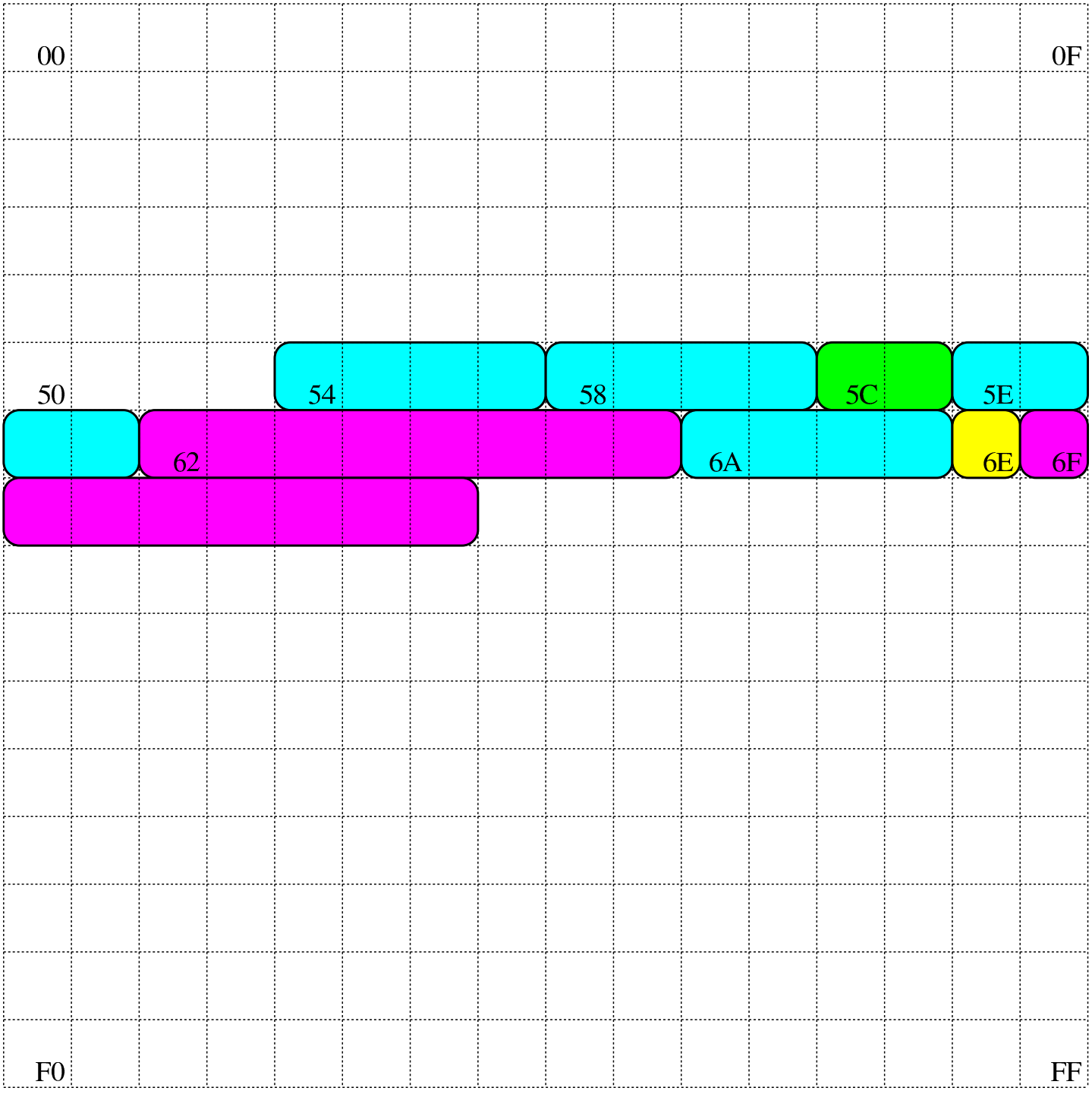
Dal momento che l'esempio di programma contiene dei valori particolari, il disegno di ciò che succede alla pila dei dati può essere reso più preciso, mettendo ciò che contengono effettivamente le varie celle della pila.

Figura 80.143. Situazione della pila nelle varie fasi della chiamata della funzione '**SOMMA**', osservando i contenuti delle varie celle.

80.7.3 Variabili e array

Con un linguaggio di programmazione molto vicino alla realtà fisica dell'elaboratore, la memoria centrale viene vista come un vettore di celle uniformi, corrispondenti normalmente a un byte. All'interno di tale vettore si distendono tutti i dati gestiti, compresa la pila descritta nelle prime sezioni del capitolo. In questo modo, le variabili in memoria si raggiungono attraverso un indirizzo che individua il primo byte che le compone ed è compito del programma il sapere di quanti byte sono composte complessivamente.

Figura 80.144. Esempio di mappa di una memoria di soli 256 byte, dove sono evidenziate alcune variabili. Gli indirizzi dei byte della memoria vanno da 00_{16} a FF_{16}.

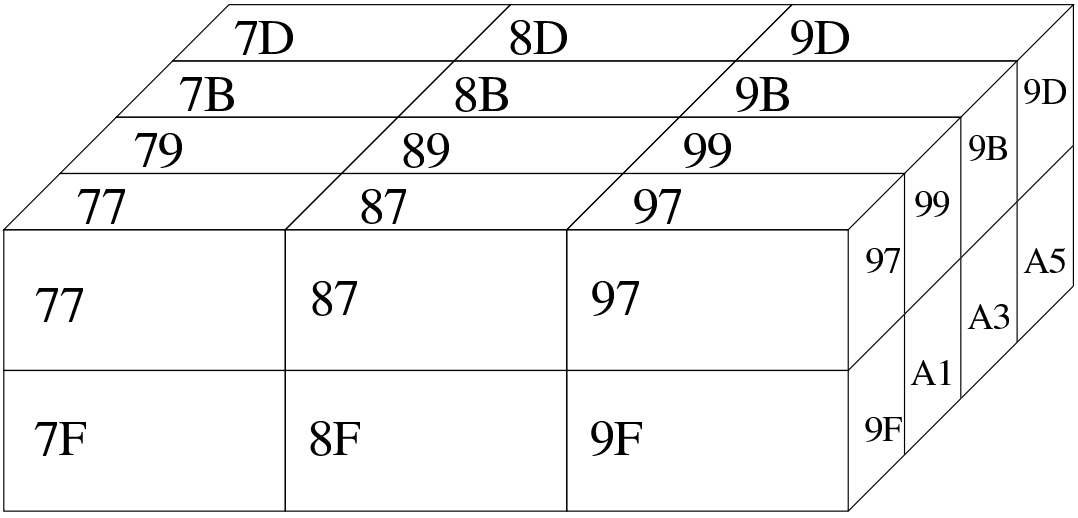


Nel disegno in cui si ipotizza una memoria complessiva di 256 byte, sono state evidenziate alcune aree di memoria:

Indirizzo	Dimensione	Indirizzo	Dimensione
54_{16}	4 byte	58_{16}	4 byte
$5C_{16}$	2 byte	$5E_{16}$	4 byte
62_{16}	8 byte	$6A_{16}$	4 byte
$6E_{16}$	1 byte	$6F_{16}$	8 byte

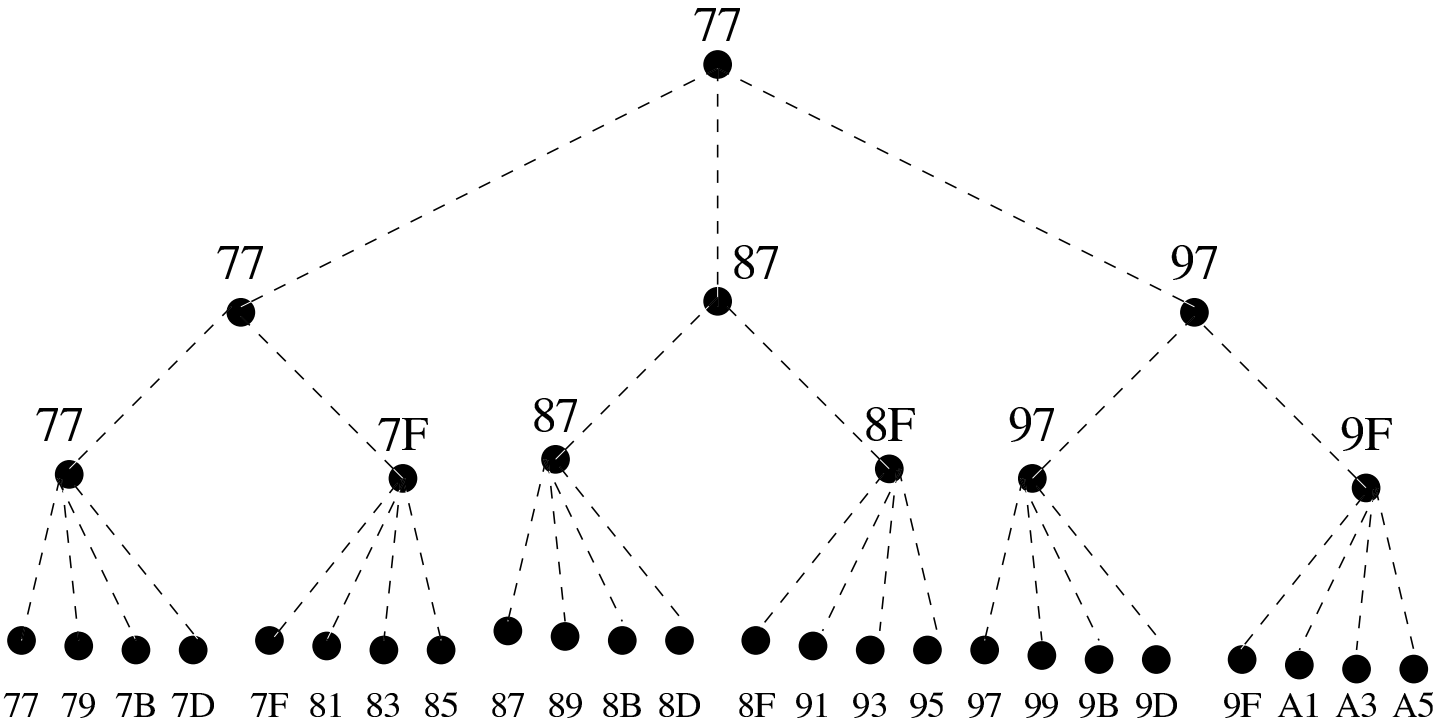
Con una gestione di questo tipo della memoria, la rappresentazione degli array richiede un po' di impegno da parte del programmatore. Nella figura successiva si rappresenta una matrice a tre dimensioni; per ora si ignorino i codici numerici associati alle celle visibili.

Figura 80.146. La matrice a tre dimensioni che si vuole rappresentare, secondo un modello spaziale. I numeri che appaiono servono a trovare successivamente l'abbinamento con le celle di memoria utilizzate.



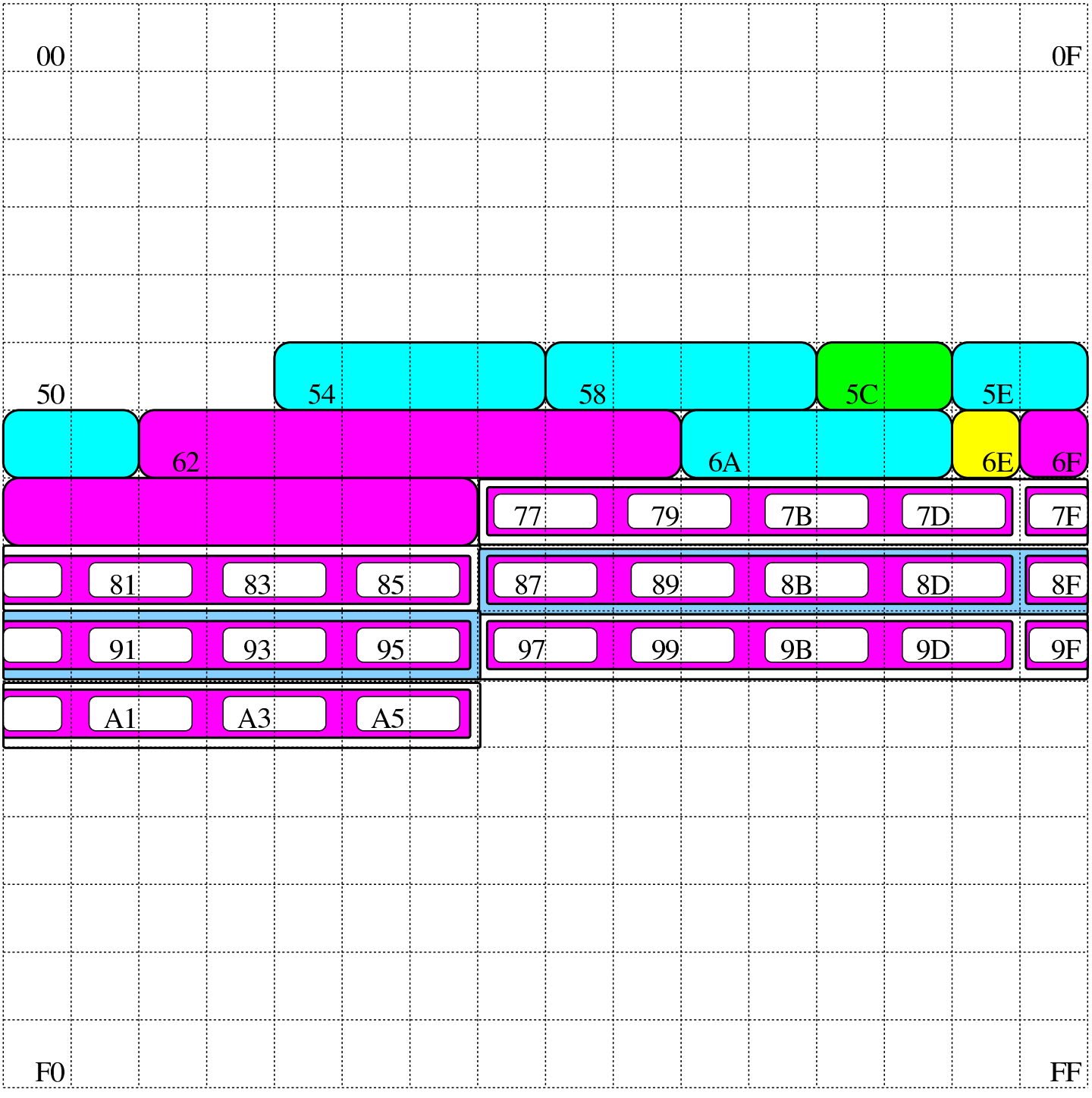
Dal momento che la rappresentazione tridimensionale rischia di creare confusione, quando si devono rappresentare matrici che hanno più di due dimensioni, è più conveniente pensare a strutture ad albero. Nella figura successiva viene tradotta in forma di albero la matrice rappresentata precedentemente.

Figura 80.147. La matrice a tre dimensioni che si vuole rappresentare, tradotta in uno schema gerarchico (ad albero).



Si suppone di rappresentare la matrice in questione nella memoria dell'elaboratore, dove ogni elemento terminale contiene due byte. Supponendo di allocare l'array a partire dall'indirizzo 77_{16} nella mappa di memoria già descritta, si potrebbe ottenere quanto si vede nella figura successiva. A questo punto, si può vedere la corrispondenza tra gli indirizzi dei vari componenti dell'array e le figure già mostrate.

Figura 80.148. Esempio di mappa di memoria in cui si distende un array che rappresenta una matrice a tre dimensioni con tre elementi contenenti ognuno due elementi che a loro volta contengono quattro elementi da due byte.



Si pone quindi il problema di scandire gli elementi dell'array. Con-

siderando che array ha dimensioni «3,2,4» e definendo che gli indici partano da zero, l'elemento [0,0,0] corrisponde alla coppia di byte che inizia all'indirizzo 77_{16}, mentre l'elemento [2,1,3] corrisponde all'indirizzo $A5_{16}$. Per calcolare l'indirizzo corrispondente a un certo elemento occorre usare la formula seguente, dove: le variabili I, J, K rappresentano la dimensioni dei componenti; le variabili i, j, k rappresentano l'indice dell'elemento cercato; la variabile A rappresenta l'indirizzo iniziale dell'array; la variabile s rappresenta la dimensione in byte degli elementi terminali dell'array.

$$A + (i \cdot J \cdot K \cdot s + j \cdot K \cdot s + k \cdot s)$$

$$A + s \cdot (i \cdot J \cdot K + j \cdot K + k)$$

Si vuole calcolare la posizione dell'elemento 2,0,1. Per facilitare i conti a livello umano, si converte l'indirizzo iniziale dell'array in base dieci: $77_{16} = 119_{10}$:

$$119 + 2 \cdot (2 \cdot 2 \cdot 4 + 0 \cdot 4 + 1) = 153$$

Il valore 153_{10} si traduce in base sedici in 99_{16}, che corrisponde effettivamente all'elemento cercato: terzo elemento principale, all'interno del quale si cerca il primo elemento, all'interno del quale si cerca il secondo elemento finale.

80.7.3.1 Esercizio

«

Una certa variabile occupa quattro unità di memoria, a partire dall'indirizzo $2F_{16}$. Qual è l'indirizzo dell'ultima unità di memoria occupata dalla variabile?

Indirizzo iniziale	Indirizzo dell'ultima unità di memoria della variabile
$2F_{16}$	

80.7.3.2 Esercizio

«

In memoria viene rappresentato un array di sette elementi da quattro unità di memoria ciascuno. Se l'indirizzo iniziale di questo array è 17_{16}, qual è l'indirizzo dell'ultima cella di memoria usata da questo array?

Indirizzo iniziale	Indirizzo dell'ultima unità di memoria dell'array
17_{16}	

80.7.3.3 Esercizio

«

In memoria viene rappresentato un array di elementi da quattro unità di memoria ciascuno. Se l'indirizzo iniziale di questo array è 17_{16}, qual è l'indirizzo iniziale del secondo elemento dell'array?

Indirizzo iniziale	Indirizzo del secondo elemento dell'array
17_{16}	

80.7.3.4 Esercizio

«

In memoria viene rappresentato un array di ***n*** elementi da quattro unità di memoria ciascuno. Se l'indirizzo iniziale di questo array è 17_{16}, a quale elemento punta l'indirizzo $2B_{16}$?

Indirizzo iniziale	Indirizzo dato	Elemento dell'array
17_{16}	$2B_{16}$	

80.7.3.5 Esercizio

«

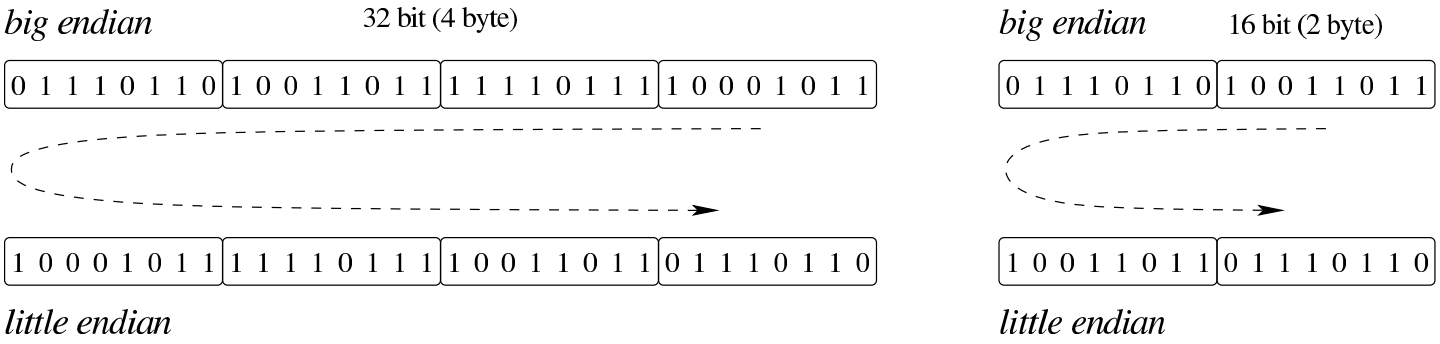
In memoria viene rappresentato un array di ***n*** elementi da quattro unità di memoria ciascuno. Se l'indirizzo iniziale di questo array è 17_{16}, l'indirizzo 22_{16} potrebbe puntare all'inizio di un certo elemento di questo?

80.7.4 Ordine dei byte

«

Come già descritto in questo capitolo, normalmente l'accesso alla memoria avviene conoscendo l'indirizzo iniziale dell'informazione cercata, sapendo poi per quanti byte questa si estende. Il microprocessore, a seconda delle proprie caratteristiche e delle istruzioni ricevute, legge e scrive la memoria a gruppetti di byte, più o meno numerosi. Ma l'ordine dei byte che il microprocessore utilizza potrebbe essere diverso da quello che si immagina di solito.

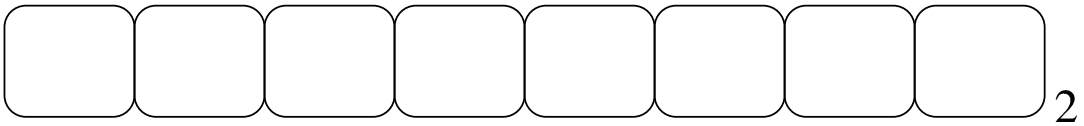
Figura 80.156. Confronto tra *big endian* e *little endian*.



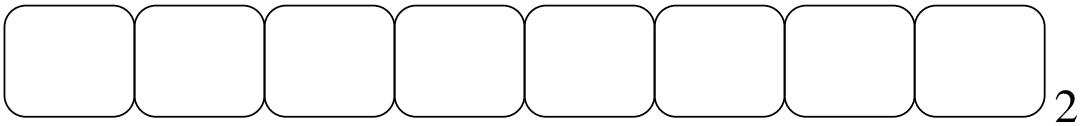
A questo proposito, per quanto riguarda la rappresentazione dei dati nella memoria, si distingue tra *big endian*, corrispondente a una rappresentazione «normale», dove il primo byte è quello più significativo (*big*), e *little endian*, dove la sequenza dei byte è invertita (ma i bit di ogni byte rimangono nella stessa sequenza standard) e il primo byte è quello meno significativo (*little*). La cosa importante da chiarire è che l'effetto dell'inversione nella sequenza porta a risultati differenti, a seconda della quantità di byte che compongono l'insieme letto o scritto simultaneamente dal microprocessore, come si vede nella figura.

80.7.4.1 Esercizio

In memoria viene rappresentata una variabile di 2 byte di lunghezza, a partire dall'indirizzo 21_{16}, contenente il valore 1111110011000000_2. Se la CPU accede alla memoria secondo la modalità *big endian*, che valore si legge all'indirizzo 21_{16} se si pretende di trovare una variabile da un solo byte?



Cosa si legge, invece, se la CPU accede alla memoria secondo la modalità *little endian* (invertita)?

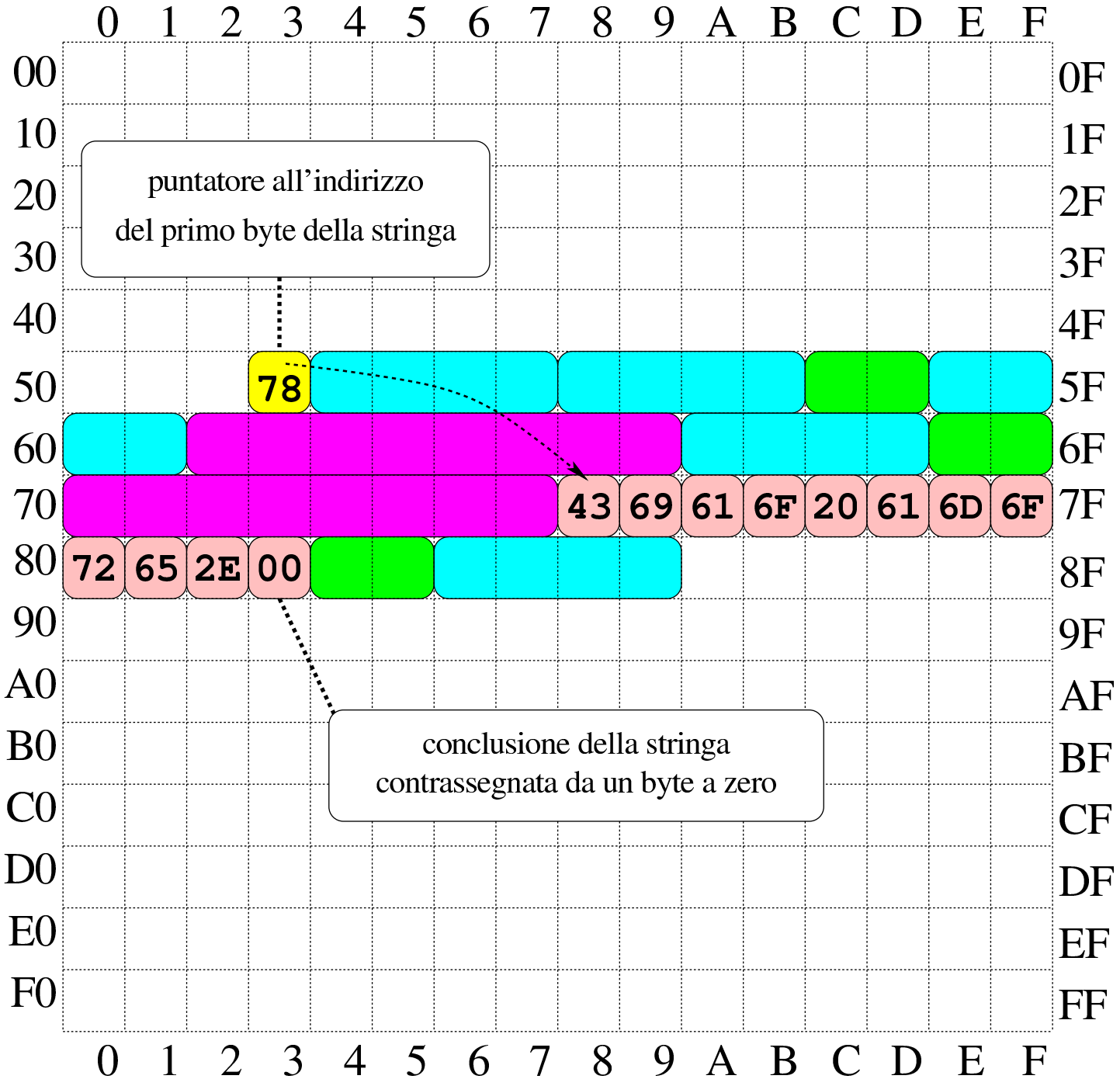


80.7.5 Stringhe, array e puntatori

«

Le stringhe sono rappresentate in memoria come array di caratteri, dove il carattere può impiegare un byte o dimensioni multiple (nel caso di UTF-8, un carattere viene rappresentato utilizzando da uno a quattro byte, a seconda del punto di codifica raggiunto). Il riferimento a una stringa viene fatto come avviene per gli array in generale, attraverso un puntatore all'indirizzo della prima cella di memoria che lo contiene; tuttavia, per non dovere annotare la dimensione di tale array, di norma si conviene che la fine della stringa sia delimitata da un byte a zero, come si vede nell'esempio della figura.

Figura 80.159. Stringa conclusa da un byte a zero (*zero terminated string*), a cui viene fatto riferimento per mezzo di una variabile che contiene il suo indirizzo iniziale. La stringa contiene il testo '**`Ciao amore.`**', secondo la codifica ASCII.



Nella figura si vede che la variabile scalare collocata all'indirizzo 53_{16} contiene un valore da intendere come indirizzo, con il quale si

fa riferimento al primo byte dell'array che rappresenta la stringa (in posizione 78_{16}). La variabile collocata in 53_{16} assume così il ruolo di ***variabile puntatore*** e, secondo il modello ridotto di memoria della figura, è sufficiente un solo byte per rappresentare un tale puntatore, dal momento che servono soltanto valori da 00_{16} a FF_{16}.

80.7.5.1 Esercizio

«

In memoria viene rappresentata la stringa «Ciao a tutti». Sapendo che ogni carattere utilizza un solo byte e che la stringa è terminata regolarmente con il codice nullo di terminazione (00_{16}), quanti byte occupa la stringa in memoria?

80.7.5.2 Esercizio

«

In memoria viene rappresentata la stringa «Ciao a tutti» (come nell'esercizio precedente). Sapendo che la stringa inizia all'indirizzo $3F_{16}$, a quale indirizzo si trova la lettera «u» di «tutti»?

80.7.5.3 Esercizio

«

Se la memoria dell'elaboratore consente di raggiungere indirizzi da 0000_{16} a $FFFF_{16}$, quanto deve essere grande una variabile scalare che si utilizza come puntatore? Si indichi la quantità di cifre binarie.

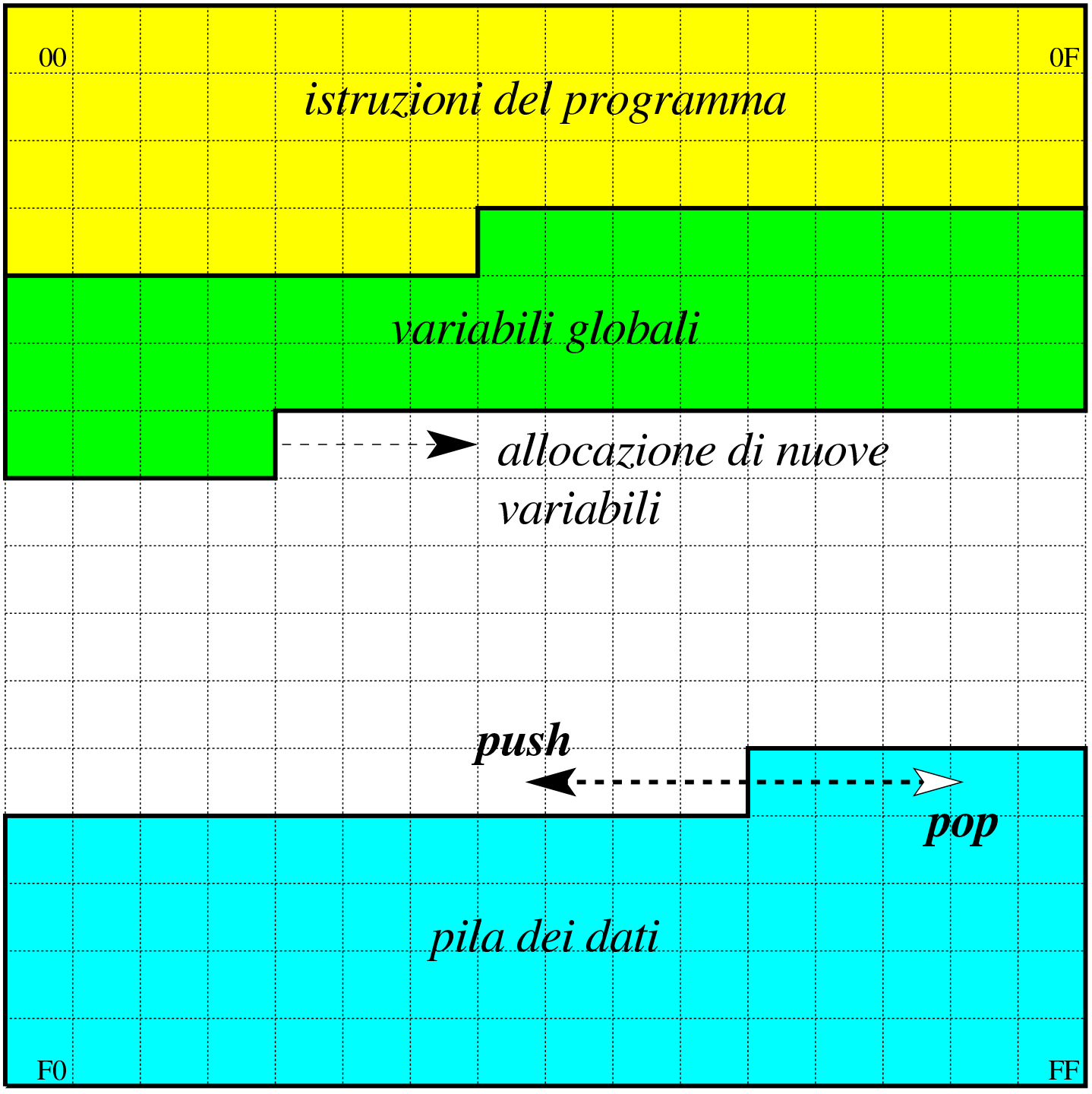
80.7.6 Utilizzo della memoria

«

La memoria dell'elaboratore viene utilizzata sia per contenere i dati, sia per il codice del programma che li utilizza. Ogni programma ha un proprio spazio in memoria, che può essere reale o virtuale; all'interno di questo spazio, la disposizione delle varie componenti potrebbe essere differente. Nei sistemi che si rifanno al modello di

Unix, nella parte più «bassa» della memoria risiede il codice che viene eseguito; subito dopo vengono le variabili globali del programma, mentre dalla parte più «alta» inizia la pila dei dati che cresce verso indirizzi inferiori. Si possono comunque immaginare combinazioni differenti di tale organizzazione, pur rispettando il vincolo di avere tre zone ben distinte per il loro contesto (codice, dati, pila); tuttavia, ci sono situazioni in cui i dati si trovano mescolati al codice, per qualche ragione.

Figura 80.160. Esempio di disposizione delle componenti di un programma in esecuzione in memoria, secondo il modello Unix.



80.8 Riferimenti

- Mario Italiani, Giuseppe Serazzi, *Elementi di informatica*, ETAS libri, 1973, ISBN 8845303632
- Sandro Petrizzelli, *Appunti di elettronica digitale*, http://users.libero.it/sandry/Digitale_01.pdf
- Tony R. Kuphaldt, *Lessons In Electric Circuits*, *Digital*, http://www.faqs.org/docs/electric/ , http://www.faqs.org/docs/electric/Digital/index.html
- Wikipedia, *Sistema numerico binario*, http://it.wikipedia.org/wiki/Sistema_numerico_binario
- Wikipedia, *IEEE 754*, http://it.wikipedia.org/wiki/IEEE_754
- Jonathan Bartlett, *Programming from the ground up*, 2003, http://savannah.nongnu.org/projects/pgubook/
- Paul A. Carter, *PC Assembly Language*, 2006, http://www.drpaulcarter.com/pcasm/

80.9 Soluzioni agli esercizi proposti

Esercizio	Soluzione
80.1.2.1	$11110011_2 = 243_{10}$.
80.1.2.2	$01100110_2 = 102_{10}$.
80.1.3.1	$1357_8 = 751_{10}$.
80.1.3.2	$7531_8 = 3929_{10}$.

Esercizio	Soluzione
80.1.4.1	$15AC_{16} = 5548_{10}$.
80.1.4.2	$CF58_{16} = 53080_{10}$.
80.2.1.1	$1234_{10} = 2322_{8}$.
80.2.1.2	$4321_{10} = 10341_{8}$.
80.2.2.1	$44221_{10} = ACBD_{16}$.
80.2.2.2	$12244_{10} = 2FD4_{16}$.
80.2.3.1	$1234_{10} = 10011010010_{2}$.
80.2.3.2	$4321_{10} = 1000011100001_{2}$.
80.2.4.1	$ABC_{16} = 101010111100_{2} = 5274_{8}$.
80.2.4.2	$7655_{8} = 111110101101_{2} = FAD_{16}$.
80.3.1.1	$43,21_{10} = 53,15341_{8}$.
80.3.1.2	$765,4321_{10} = 2FD,6E9E1_{16}$.
80.3.1.3	$21,11_{10} = 10101,00011_{2}$.
80.3.2.1	$765,432_{8} = 501,55078_{10}$.
80.3.2.2	$AB,CD_{16} = 171,80078_{10}$.
80.3.2.3	$101010,110011_{2} = 42,79687_{10}$.

Esercizio	Soluzione
80.3.3.1	$76{,}55_8 = 00111110{,}10110100_2 = 3E{,}B4_{16}$.
80.3.3.2	$A7{,}C1_{16} = 010100111{,}110000010_2 = 247{,}602_8$.
80.4.1.1	complemento alla base di $0000123456_{10} = 9999876544_{10}$.
80.4.1.2	complemento alla base di $9999123456_{10} = 0000876544_{10}$.
80.4.2.1	complemento a uno di $0011001001000101_2 = 1100110110111010_2$; complemento a due = 1100110110111011_2.
80.4.2.2	complemento a uno di $1111001100010101_2 = 0000110011101010_2$; complemento a due = 0000110011101011_2.
80.4.8.1	$+103_{10} = 0000000001100111_2$.
80.4.8.2	$-103_{10} = 1111111110011001_2$.
80.4.8.3	$1111111111110001_2 = -15_{10}$; complemento a due = 0000000000001111_2.
80.4.8.4	$0000000000110001_2 = +49_{10}$; complemento a due = 1111111111001111_2; se 1111111111001111_2 fosse inteso senza segno sarebbe uguale a 65487_{10}.
80.4.8.5	da 0_{10} a 2047_{10} indica valori positivi; da 2048_{10} a 4095_{10} indica valori negativi.
80.4.8.6	da 0_{10} a 32767_{10} indica valori positivi; da 32768_{10} a 65535_{10} indica valori negativi.
80.5.1.1	11100011_2 con segno si traduce, a sedici cifre in 1111111111100011_2.

Esercizio	Soluzione
80.5.1.2	0000111110001111_2 con segno equivale a $+3983_{10}$, mentre 10001111_2 con segno equivale a -113_{10}; se poi si volesse supporre che la riduzione di cifre mantenga il segno, si avrebbe 00001111_2 che equivale a $+15_{10}$. Pertanto, in questo caso, la riduzione di cifre non può essere valida.
80.5.1.3	11100011_2 con segno equivale a -29_{10}; copiando questo valore in una variabile senza segno, a sedici cifre, si ottiene 0000000011100011_2, pari a 227_{10}. Se, successivamente, si interpreta il nuovo valore con segno, si ottiene $+227_{10}$, che non corrisponde in alcun modo al valore originale.
80.5.2.1	01010101_2 con segno + 01111110_2 con segno = 11010011_2 con riporto di zero. Il risultato non è valido perché, pur sommando due valori positivi, il segno è diventato negativo.
80.5.2.2	11010101_2 con segno + 01111110_2 con segno = 01010011_2 con riporto di uno. Il risultato della somma tra un numero positivo e un numero negativo è sempre valido.
80.5.2.3	11010101_2 con segno + 10000001_2 con segno = 01010110_2 con riporto di uno. Il risultato non è valido perché si sommano due numeri negativi, ma il risultato è positivo.
80.5.3.1	$11010101_2 + 10000001_2 = 01001011_2$ con riporto di uno. Il risultato non è valido perché c'è un riporto.
80.5.3.2	$11010101_2 + 11110110_2 = 11001011_2$ con riporto di uno. Il risultato non è valido perché c'è un riporto.
80.5.3.3	La sottrazione $11010101_2 - 11110110_2$ va trasformata nella somma $11010101_2 + 00001010_2 = 11011111_2$ senza riporto. Il risultato non è valido perché manca il riporto (d'altra parte si sta sottraendo un valore più grande del minuendo, pertanto il risultato senza segno non può essere valido

Esercizio	Soluzione
80.5.3.4	La sottrazione $11010101_2 - 00001111_2$ va trasformata nella somma $11010101_2 + 11110001_2 = 11000110_2$ con riporto di uno. Il risultato è valido perché si ha un riporto
80.6.1.1	Lo scorrimento logico a sinistra di 11010101_2, di una sola cifra, è pari a 10101010_2.
80.6.1.2	Lo scorrimento logico a destra di 11010101_2, di una sola cifra, è pari a 01101010_2.
80.6.2.1	Lo scorrimento aritmetico a sinistra di 01010101_2 (con segno), di una sola cifra, è pari a 10101010_2, ma si ottiene un cambiamento di segno e il risultato non è valido.
80.6.2.2	Lo scorrimento aritmetico a destra di 01010101_2 (con segno), di una sola cifra, è pari a 00101010_2. Il risultato è valido, in quanto è stato possibile preservare il segno e il valore ottenuto è pari alla divisione per due di quello originale.
80.6.2.3	Lo scorrimento aritmetico a destra di 11010101_2 (con segno), di una sola cifra, è pari a 11101010_2. Il risultato è valido, in quanto è stato possibile preservare il segno e il valore ottenuto è pari alla divisione per due di quello originale.
80.6.6.1	0010010101011111_2 AND $0110001111000011_2 = 0010000101000011_2$.
80.6.6.2	0010010101011111_2 OR $0110001111000011_2 = 0110011111011111_2$.
80.6.6.3	0010010101011111_2 XOR $0110001111000011_2 = 0100011010011100_2$.
80.6.6.4	NOT $0010010101011111_2 = 1101101010100000_2$.

Esercizio	Soluzione
80.7.3.1	L'ultima unità di memoria usata dalla variabile scalare si trova all'indirizzo 32_{16}.
80.7.3.2	L'array è lungo 28 unità di memoria e termina all'indirizzo 32_{16} incluso.
80.7.3.3	L'indirizzo del secondo elemento dell'array è $1B_{16}$.
80.7.3.4	L'indirizzo $2B_{16}$ punta al sesto elemento dell'array.
80.7.3.5	L'indirizzo 22_{16} individua una cella di memoria del terzo elemento dell'array, ma non trattandosi dell'inizio di tale elemento, non è utile come indice dello stesso.
80.7.4.1	In modalità *big endian*, la variabile che contiene 1111110011000000_2, se viene letta come se fosse costituita da un solo byte, darebbe 11111100_2, ovvero la porzione più significativa della stessa. Invece, in modalità *little endian*, ciò che si leggerebbe sarebbe la porzione meno significativa: 11000000_2.
80.7.5.1	La stringa «Ciao a tutti», terminata regolarmente, occupa 13 byte.
80.7.5.2	Sapendo che la stringa «Ciao a tutti» inizia all'indirizzo $3F_{16}$, la lettera «u» si trova all'indirizzo 47_{16}.
80.7.5.3	La variabile che consenta di rappresentare puntatori per indirizzi da 0000_{16} a $FFFF_{16}$, deve essere almeno da 16 bit (sedici cifre binarie).

[1] Nel contesto riferito alla definizione di un numero in virgola mobile, si possono usare indifferentemente i termini ***mantissa*** o ***significante***, così come sono indifferenti i termini ***caratteristica*** o ***esponente***.

[2] Si osservi che lo standard IEEE 754 utilizza una «mantissa normalizzata» che indica la frazione di valore tra uno e due: «1,***mantissa***.

Capitolo 81

Nozioni minime sul linguaggio C

81.1 Primo approccio al linguaggio C 2375
81.1.1 Struttura fondamentale 2375
81.1.2 Ciao mondo! 2378
81.1.3 Compilazione 2380
81.1.4 Emissione dati attraverso «printf()» 2382
81.2 Variabili e tipi del linguaggio C 2383
81.2.1 Bit, byte e caratteri 2384
81.2.2 Tipi primitivi 2385
81.2.3 Costanti letterali comuni 2391
81.2.4 Caratteri privi di rappresentazione grafica 2394
81.2.5 Valore numerico delle costanti carattere 2398
81.2.6 Campo di azione delle variabili 2400
81.2.7 Dichiarazione delle variabili 2400
81.2.8 Il tipo indefinito: «void» 2402
81.3 Operatori ed espressioni del linguaggio C 2402
81.3.1 Tipo del risultato di un'espressione 2405
81.3.2 Operatori aritmetici 2406
81.3.3 Operatori di confronto 2410
81.3.4 Operatori logici 2413
81.3.5 Operatori binari 2414

81.3.6 Conversione di tipo 2418
81.3.7 Espressioni multiple 2420
81.4 Strutture di controllo di flusso del linguaggio C 2423
81.4.1 Struttura condizionale: «if» 2423
81.4.2 Struttura di selezione: «switch» 2428
81.4.3 Iterazione con condizione di uscita iniziale: «while» 2433
81.4.4 Iterazione con condizione di uscita finale: «do-while» 2436
81.4.5 Ciclo enumerativo: «for» 2437
81.5 Funzioni del linguaggio C 2441
81.5.1 Dichiarazione di un prototipo 2441
81.5.2 Descrizione di una funzione 2443
81.5.3 Vincoli nei nomi 2447
81.5.4 I/O elementare 2448
81.5.5 Restituzione di un valore 2450
81.6 Riferimenti 2453
81.7 Soluzioni agli esercizi proposti 2453

! 2402 2413 != 2402 2410 * 2402 2406 *= 2402 2406 + 2402 2406 ++ 2402 2406 += 2402 2406 / 2402 2406 /*...*/ 2375 // 2375 /= 2402 2406 0... 2391 0x... 2391 ; 2375 = 2402 2406 == 2402 2410 ? : 2402 2413 break 2428 2433 2437 case 2428 char 2385 const 2400 continue 2433 2437 default 2428 do 2436 double 2385 else 2423 exit() 2450 F 2391 float

2385 for 2437 if 2423 int 2385 L 2391 2391 LL 2391 long 2385 long long 2385 printf() 2382 return 2443 short 2385 signed 2385 switch 2428 U 2391 UL 2391 ULL 2391 unsigned 2385 void 2402 2441 while 2433 # 2375 & 2402 2414 &= 2402 2414 && 2402 2413 ^ 2402 2414 ^= 2402 2414 ~ 2402 2414 ~= 2402 2414 \... 2394 \0 2394 \? 2394 \a 2394 \b 2394 \f 2394 \n 2394 \r 2394 \t 2394 \v 2394 \x... 2394 \" 2394 \\ 2394 \' 2394 | 2402 2414 |= 2402 2414 || 2402 2413 {...} 2375 '...' 2391 , 2420 - 2402 2406 -= 2402 2406 -- 2402 2406 < 2402 2410 <= 2402 2410 << 2402 2414 <<= 2402 2414 > 2402 2410 >= 2402 2410 >> 2402 2414 >>= 2402 2414 % 2402 2406 %= 2402 2406

81.1 Primo approccio al linguaggio C

Il linguaggio C richiede la presenza di un compilatore per generare un file eseguibile (o interpretabile) dal kernel. Se si dispone di un sistema GNU con i cosiddetti «strumenti di sviluppo», intendendo con questo ciò che serve a ricompilare il kernel, si dovrebbe disporre di tutto quello che è necessario per provare gli esempi di questi capitoli. In alternativa, disponendo solo di un sistema MS-Windows, potrebbe essere utile il pacchetto DevCPP che ha la caratteristica di essere molto semplice da installare.

81.1.1 Struttura fondamentale

Il contenuto di un sorgente in linguaggio C può essere suddiviso in tre parti: commenti, direttive del precompilatore e istruzioni C. I commenti vanno aperti e chiusi attraverso l'uso dei simboli '**/***' e '***/**'; se poi il compilatore è conforme a standard più recenti, è

ammissibile anche l'uso di '**//**' per introdurre un commento che termina alla fine della riga.

```
/* Questo è un commento che continua
   su più righe e finisce qui. */

// Qui inizia un altro commento che termina alla fine della
// riga; pertanto, per ogni riga va ripetuta la sequenza
// "//" di apertura.
```

Le direttive del precompilatore rappresentano un linguaggio che guida alla compilazione del codice vero e proprio. L'uso più comune di queste direttive viene fatto per includere porzioni di codice sorgente esterne al file. È importante fare attenzione a non confondersi, dal momento che tali istruzioni iniziano con il simbolo '**#**': non si tratta di commenti.

Il programma C tipico richiede l'inclusione di codice esterno composto da file che terminano con l'estensione '`.h`'. La libreria che viene inclusa più frequentemente è quella necessaria alla gestione dei flussi di standard input, standard output e standard error; si dichiara il suo utilizzo nel modo seguente:

```
#include <stdio.h>
```

Le istruzioni C terminano con un punto e virgola ('**;**') e i raggruppamenti di queste (noti come «istruzioni composte») si fanno utilizzando le parentesi graffe ('**{ }**').[1]

istruzione **;**

{ ***istruzione*** **;** ***istruzione*** **;** ***istruzione*** **;** }

Generalmente, un'istruzione può essere interrotta e ripresa nella riga successiva, dal momento che la sua conclusione è dichiarata chiaramente dal punto e virgola finale. L'istruzione nulla viene rappresentata utilizzando un punto e virgola da solo.

I nomi scelti per identificare ciò che si utilizza all'interno del programma devono seguire regole determinate, definite dal compilatore C a disposizione. Ma per cercare di scrivere codice portabile in altre piattaforme, conviene evitare di sfruttare caratteristiche speciali del proprio ambiente. In particolare:

- un nome può iniziare con una lettera alfabetica e continuare con altre lettere, cifre numeriche e il trattino basso;
- in teoria i nomi potrebbero iniziare anche con il trattino basso, ma è sconsigliabile farlo, se non ci sono motivi validi per questo;[2]
- nei nomi si distinguono le lettere minuscole da quelle maiuscole (pertanto, '**`Nome`**' è diverso da '**`nome`**' e da tante altre combinazioni di minuscole e maiuscole).

La lunghezza dei nomi può essere un elemento critico; generalmente la dimensione massima dovrebbe essere di 32 caratteri, ma ci sono versioni di C che ne possono accettare solo una quantità inferiore. In particolare, il compilatore GNU ne accetta molti di più di 32. In ogni caso, il compilatore non rifiuta i nomi troppo lunghi, semplicemente non ne distingue più la differenza oltre un certo punto.

Il codice di un programma C è scomposto in funzioni, dove normalmente l'esecuzione del programma corrisponde alla chiamata della funzione ***main()***. Questa funzione può essere dichiarata senza parametri, '**`int main (void)`**', oppure con due parametri precisi: '**`int main (int argc, char *argv[])`**'.

81.1.2 Ciao mondo!

«

Come sempre, il modo migliore per introdurre a un linguaggio di programmazione è di proporre un esempio banale, ma funzionante. Al solito si tratta del programma che emette un messaggio e poi termina la sua esecuzione.

Listato 81.3. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/vYaJyc7X* , *http://ideone.com/mxSUL* .

```
/*
 *      Ciao mondo!
 */

#include <stdio.h>

/* La funzione main() viene eseguita automaticamente
   all'avvio. */
int main (void)
{
    /* Si limita a emettere un messaggio. */
    printf ("Ciao mondo!\n");

    /* Attende la pressione di un tasto, quindi termina. */
    getchar ();
    return 0;
}
```

Nel programma sono state inserite alcune righe di commento. In particolare, all'inizio, l'asterisco che si trova nella seconda riga ha soltanto un significato estetico, per guidare la vista verso la conclusione del commento stesso.

Il programma si limita a emettere la stringa «Ciao Mondo!» seguita

da un codice di interruzione di riga, rappresentato dal simbolo '**\n**'.

81.1.2.1 Esercizio

«

Si modifichi l'esempio di programma mostrato, in modo da usare solo commenti del tipo '**//**'. Si può completare a penna il listato successivo

Listato 81.4. Per eseguire l'esercizio attraverso un servizio *pastebin*: *http://codepad.org/Pqit6Nna* , *http://ideone.com/0h1QC* .

```
        Ciao mondo!


#include <stdio.h>

    La funzione main() viene eseguita automaticamente
    all'avvio.

int main (void)
{

        Si limita a emettere un messaggio.

    printf ("Ciao mondo!\n");

        Attende la pressione di un tasto, quindi termina.

    getchar ();
    return 0;
}
```

81.1.2.2 Esercizio

«

Si modifichi l'esempio di programma mostrato, in modo da emettere il testo seguente, come si può vedere:

```
Il mio primo programma
scritto in linguaggio C.
```

Si completi per questo lo schema seguente.

Listato 81.6. Per eseguire l'esercizio attraverso un servizio *pastebin*: *http://codepad.org/5Rl7VtD7* , *http://ideone.com/WxWGX* .

```
#include <stdio.h>
int main (void)
{




    getchar ();
    return 0;
}
```

81.1.3 Compilazione

«

Per compilare un programma scritto in C, nell'ambito di un sistema operativo tradizionale, si utilizza generalmente il comando '**cc**', anche se di solito si tratta di un collegamento simbolico al vero compilatore che si ha a disposizione. Supponendo di avere salvato il file dell'esempio con il nome '`ciao.c`', il comando per la sua compilazione è il seguente:

```
$ cc ciao.c[Invio]
```

Quello che si ottiene è il file ‘`a.out`’ che dovrebbe già avere i permessi di esecuzione.

```
$ ./a.out [Invio]

  Ciao mondo!
```

Se si desidera compilare il programma definendo un nome diverso per il codice eseguibile finale, si può utilizzare l’opzione standard ‘**`-o`**’.

```
$ cc -o ciao ciao.c [Invio]
```

Con questo comando, si ottiene l’eseguibile ‘**`ciao`**’.

```
$ ./ciao [Invio]

  Ciao mondo!
```

In generale, se ciò è possibile, conviene chiedere al compilatore di mostrare gli avvertimenti (*warning*), senza limitarsi ai soli errori. Pertanto, nel caso il compilatore sia GNU C, è bene usare l’opzione ‘**`-Wall`**’:

```
$ cc -Wall -o ciao ciao.c [Invio]
```

81.1.3.1 Esercizio

«

Quale comando si deve dare per compilare il file ‘`prova.c`’ e ottenere il file eseguibile ‘`programma`’?

```
$                                                    [Invio]
```

81.1.4 Emissione dati attraverso «printf()»

«

L'esempio di programma presentato sopra si avvale della funzione ***printf()***[3] per emettere il messaggio attraverso lo standard output. Questa funzione è più sofisticata di quanto possa apparire dall'esempio, in quanto permette di comporre il risultato da emettere. Negli esempi più semplici di codice C appare immancabilmente questa funzione, per cui è necessario descrivere subito, almeno in parte, il suo funzionamento.

```
int printf (stringa_di_formato [, espressione] ...);
```

La funzione ***printf()*** emette attraverso lo standard output la stringa che costituisce il primo parametro, dopo averla rielaborata in base alla presenza di ***specificatori di conversione*** riferiti alle eventuali espressioni che compongono gli argomenti successivi; inoltre restituisce il numero di caratteri emessi.

L'utilizzo più semplice di ***printf()*** è quello che è già stato visto, cioè l'emissione di una stringa senza specificatori di conversione (il codice '**\n**' rappresenta un carattere preciso e non è uno specificatore, piuttosto si tratta di una cosiddetta sequenza di escape).

```
    printf ("Ciao mondo!\n");
```

La stringa può contenere degli specificatori di conversione del tipo '**%d**', '**%c**', '**%f**',... e questi fanno ordinatamente riferimento agli argomenti successivi. L'esempio seguente fa in modo che la stringa incorpori il valore del secondo argomento nella posizione in cui appare '**%d**':

```
    printf ("Totale fatturato: %d\n", 12345);
```

Lo specificatore di conversione '**%d**' stabilisce anche che il valore in questione deve essere trasformato secondo una rappresentazione decimale intera. Per cui, il risultato diviene esattamente quello che ci si aspetta.

```
Totale fatturato: 12345
```

81.1.4.1 Esercizio

Si vuole visualizzare il testo seguente:

```
Imponibile: 1000, IVA: 200.
```

Sulla base delle conoscenze acquisite, si completi l'istruzione seguente:

```
printf ("                                    ", 1000, 200);
```

81.2 Variabili e tipi del linguaggio C

I tipi di dati elementari gestiti dal linguaggio C dipendono dall'architettura dell'elaboratore sottostante. In questo senso, volendo fare un discorso generale, è difficile definire la dimensione delle variabili numeriche; si possono dare solo delle definizioni relative. Solitamente, il riferimento è costituito dal tipo numerico intero ('**int**') la cui dimensione in bit corrisponde a quella della ***parola***, ovvero dalla capacità dell'unità aritmetico-logica del microprocessore, oppure a qualunque altra entità che il microprocessore sia in grado di gestire con la massima efficienza. In pratica, con l'architettura x86 a 32 bit, la dimensione di un intero normale è di 32 bit, ma rimane la stessa anche con l'architettura x86 a 64 bit.

I documenti che descrivono lo standard del linguaggio C, definiscono la «dimensione» di una variabile come ***rango*** (*rank*).

81.2.1 Bit, byte e caratteri

«

A proposito della gestione delle variabili, esistono pochi concetti che sembrano rimanere stabili nel tempo. Il riferimento più importante in assoluto è il byte, che per il linguaggio C è almeno di 8 bit, ma potrebbe essere più grande. Dal punto di vista del linguaggio C, il byte è l'elemento più piccolo che si possa indirizzare nella memoria centrale, questo anche quando la memoria fosse organizzata effettivamente a parole di dimensione maggiore del byte. Per esempio, in un elaboratore che suddivide la memoria in blocchi da 36 bit, si potrebbero avere byte da 9, 12, 18 bit o addirittura 36 bit.[4]

Una volta definito il byte, si considera che il linguaggio C rappresenti ogni variabile scalare come una sequenza continua di byte; pertanto, tutte le variabili scalari sono rappresentate come multipli di byte; di conseguenza anche le variabili strutturate lo sono, con la differenza che in tal caso potrebbero inserirsi dei «buchi» (in byte), dovuti alla necessità di allineare i dati in qualche modo.

Il tipo '**`char`**' (carattere), indifferentemente se si considera o meno il segno, rappresenta tradizionalmente una variabile numerica che occupa esattamente un byte, pertanto, spesso si confondono i termini «carattere» e «byte», nei documenti che descrivono il linguaggio C.

A causa della capacità limitata che può avere una variabile di tipo '**`char`**', il linguaggio C distingue tra un insieme di caratteri «minimo» e un insieme «esteso», da rappresentare però in altra forma.

81.2.1.1 Esercizio

Secondo la logica del linguaggio C, se un byte è formato da 8 bit, ci può essere una variabile scalare da 12 bit? Perché?

81.2.2 Tipi primitivi

I tipi di dati primitivi rappresentano un valore **numerico** singolo, nel senso che anche il tipo '**`char`**' viene trattato come un numero. Il loro elenco essenziale si trova nella tabella successiva.

Tabella 81.14. Elenco dei tipi comuni di dati primitivi elementari in C.

Tipo	Descrizione
`char`	Carattere (generalmente di 8 bit).
`int`	Intero normale.
`float`	Virgola mobile a precisione singola.
`double`	Virgola mobile a precisione doppia.

Come già accennato, non si può stabilire in modo generale quali siano le dimensioni esatte in bit dei vari tipi di dati, ovvero il rango, in quanto l'elemento certo è solo la relazione tra loro.

$$\text{char} \leq \text{int} \leq \text{float} \leq \text{double}$$

Questi tipi primitivi possono essere estesi attraverso l'uso di alcuni qualificatori: '**`short`**', '**`long`**', '**`long long`**', '**`signed`**'[5] e '**`unsigned`**'.[6] I primi tre si riferiscono al rango, mentre gli altri modificano il modo di valutare il contenuto di alcune variabili. La ta-

bella successiva riassume i vari tipi primitivi con le combinazioni ammissibili dei qualificatori.

Tabella 81.16. Elenco dei tipi comuni di dati primitivi in C assieme ai qualificatori usuali.

Tipo	Abbreviazione	Descrizione
`char`		Tipo '**`char`**' per il quale non conta sapere se il segno viene considerato o meno.
`signed char`		Tipo '**`char`**' usato numericamente con segno.
`unsigned char`		Tipo '**`char`**' usato numericamente senza segno.
`short int` `signed short int`	`short` `signed short`	Intero più breve di '**`int`**', con segno.
`unsigned short int`	`unsigned short`	Tipo '**`short`**' senza segno.
`int` `signed int`		Intero normale, con segno.
`unsigned int`	`unsigned`	Tipo '**`int`**' senza segno.
`long int` `signed long int`	`long` `signed long`	Intero più lungo di '**`int`**', con segno.

Tipo	Abbreviazione	Descrizione
`unsigned long int`	`unsigned long`	Tipo '**`long`**' senza segno.
`long long int` `signed long long int`	`long long` `signed long long`	Intero più lungo di '**`long int`**', con segno.
`unsigned long long int`	`unsigned long long`	Tipo '**`long long`**' senza segno.
`float`		Tipo a virgola mobile a precisione singola.
`double`		Tipo a virgola mobile a precisione doppia.
`long double`		Tipo a virgola mobile «più lungo» di '**`double`**'.

Così, il problema di stabilire le relazioni di rango si complica:

$$\text{char} \leq \text{short} \leq \text{int} \leq \text{long} \leq \text{long long}$$

$$\text{float} \leq \text{double} \leq \text{long double}$$

I tipi '**`long`**' e '**`float`**' potrebbero avere un rango uguale, altrimenti non è detto quale dei due sia più grande.

Il programma seguente, potrebbe essere utile per determinare il rango dei vari tipi primitivi nella propria piattaforma.[7]

Listato 81.18. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/92vD92wUlM* , *http://ideone.com/q5unh* .

```
#include <stdio.h>

int main (void)
{
  printf ("char           %d\n", (int) sizeof (char));
  printf ("short int      %d\n", (int) sizeof (short int));
  printf ("int            %d\n", (int) sizeof (int));
  printf ("long int       %d\n", (int) sizeof (long int));
  printf ("long long int %d\n", (int) sizeof (long long int));
  printf ("float          %d\n", (int) sizeof (float));
  printf ("double         %d\n", (int) sizeof (double));
  printf ("long double    %d\n", (int) sizeof (long double));
  getchar ();
  return 0;
}
```

Il risultato potrebbe essere simile a quello seguente:

```
char           1
short int      2
int            4
long int       4
long long int 8
float          4
double         8
long double    12
```

I numeri rappresentano la quantità di caratteri, nel senso di valori '**char**', per cui il tipo '**char**' dovrebbe sempre avere una dimensione unitaria.[8]

I tipi primitivi di variabili mostrati sono tutti utili alla memorizzazione di valori numerici, a vario titolo. A seconda che il valore in questione sia trattato con segno o senza segno, varia lo spettro di valori che possono essere contenuti.

Nel caso di interi ('**`char`**', '**`short`**', '**`int`**', '**`long`**' e '**`long long`**'), la variabile può essere utilizzata per tutta la sua estensione a contenere un numero binario. Pertanto, quando la rappresentazione è senza segno, il massimo valore ottenibile è $(2^{n})-1$, dove n rappresenta il numero di bit a disposizione. Quando invece si vuole trattare il dato come un numero con segno, il valore numerico massimo ottenibile è circa la metà (se si usa la rappresentazione dei valori negativi in complemento a due, l'intervallo di valori va da $(2^{n-1})-1$ a $-(2^{n-1})$)

Nel caso di variabili a virgola mobile non c'è più la possibilità di rappresentare esclusivamente valori senza segno; inoltre, più che esserci un limite nella grandezza rappresentabile, c'è soprattutto un limite nel grado di approssimazione.

Le variabili '**`char`**' sono fatte, in linea di principio, per contenere il codice di rappresentazione di un carattere, secondo la codifica utilizzata nel sistema. Ma il fatto che questa variabile possa essere gestita in modo numerico, permette una facile conversione da lettera a codice numerico corrispondente.

Un tipo di valore che non è stato ancora visto è quello logico: *Vero* è rappresentato da un qualsiasi valore numerico intero diverso da zero, mentre *Falso* corrisponde a zero.

81.2.2.1 Esercizio

«

Dovendo rappresentare numeri interi da 0 a 99999, può bastare una variabile scalare di tipo '**`unsigned char`**', sapendo che il tipo '**`char`**' utilizza 8 bit?

81.2.2.2 Esercizio

«

Qual è l'intervallo di valori che si possono rappresentare con una variabile di tipo '**`unsigned char`**', sapendo che il tipo '**`char`**' utilizza 8 bit?

Valore minimo	Valore massimo

81.2.2.3 Esercizio

«

Qual è l'intervallo di valori che si possono rappresentare con una variabile di tipo '**`signed short int`**', sapendo che il tipo '**`short int`**' utilizza 16 bit e che i valori negativi si esprimono attraverso il complemento a due?

Valore minimo	Valore massimo

81.2.2.4 Esercizio

«

Dovendo rappresentare il valore 12,34, è possibile usare una variabile di tipo ‘**int**’? Se non fosse possibile, quale tipo si potrebbe usare?

81.2.3 Costanti letterali comuni

«

Quasi tutti i tipi di dati primitivi hanno la possibilità di essere rappresentati in forma di costante letterale. In particolare, si distingue tra:

- costanti carattere, rappresentate da un carattere alfanumerico racchiuso tra apici singoli, come ‘**'A'**’, ‘**'B'**’,...;
- costanti intere, rappresentate da un numero senza decimali, e a seconda delle dimensioni può trattarsi di uno dei vari tipi di interi (escluso ‘**char**’);
- costanti con virgola, rappresentate da un numero con decimali (un punto seguito da altre cifre, anche se si tratta solo di zeri) che, indipendentemente dalle dimensioni, di norma sono di tipo ‘**double**’.

Per esempio, 123 è generalmente una costante ‘**int**’, mentre 123.0 è una costante ‘**double**’.

Le costanti che esprimono valori interi possono essere rappresentate con diverse basi di numerazione, attraverso l’indicazione di un prefisso: ‘**0*n***’, dove ***n*** contiene esclusivamente cifre da zero a sette, viene inteso come un numero in base otto; ‘**0x*n***’ o ‘**0X*n***’, dove ***n*** può contenere le cifre numeriche consuete, oltre alle lettere da «A»

a «F» (minuscole o maiuscole, indifferentemente) viene trattato come un numero in base sedici; negli altri casi, un numero composto con cifre da zero a nove è interpretato in base dieci.

Per quanto riguarda le costanti che rappresentano numeri con virgola, oltre alla notazione '***intero*.*decimali***' si può usare la notazione scientifica. Per esempio, '**`7e+15`**' rappresenta l'equivalente di $7 \cdot (10^{15})$, cioè un sette con 15 zeri. Nello stesso modo, '**`7e-5`**', rappresenta l'equivalente di $7 \cdot (10^{-5})$, cioè 0,00007.

Il tipo di rappresentazione delle costanti numeriche, intere o con virgola, può essere specificato aggiungendo un suffisso, costituito da una o più lettere, come si vede nelle tabelle successive. Per esempio, '**`123UL`**' è un numero di tipo '**`unsigned long int`**', mentre '**`123.0F`**' è un tipo '**`float`**'. Si osservi che il suffisso può essere composto, indifferentemente, con lettere minuscole o maiuscole.

Tabella 81.22. Suffissi per le costanti che esprimono valori interi.

Suffisso	Descrizione
assente	In tal caso si tratta di un intero «normale» o più grande, se necessario.
`U`	Tipo senza segno ('**`unsigned`**').
`L`	Intero più grande della dimensione normale ('**`long`**').
`LL`	Intero molto più grande della dimensione normale ('**`long long`**').
`UL`	Intero senza segno, più grande della dimensione normale ('**`unsigned long`**').
`ULL`	Intero senza segno, molto più grande della dimensione normale ('**`unsigned long long`**').

Tabella 81.23. Suffissi per le costanti che esprimono valori con virgola.

Suffisso	Descrizione
assente	Tipo ‘**`double`**’.
`F`	Tipo ‘**`float`**’.
`L`	Tipo ‘**`long double`**’.

È possibile rappresentare anche le stringhe in forma di costante attraverso l’uso degli apici doppi, ma la stringa non è un tipo di dati primitivo, trattandosi piuttosto di un array di caratteri. Per il momento è importante fare attenzione a non confondere il tipo ‘**`char`**’ con la stringa. Per esempio, ‘**`'F'`**’ è un carattere (con un proprio valore numerico), mentre ‘**`"F"`**’ è una stringa, ma la differenza tra i due è notevole. Le stringhe vengono descritte nella sezione 66.5.

81.2.3.1 Esercizio

Indicare il valore, in base dieci, rappresentato dalle costanti che appaiono nella tabella successiva:

Costante	Valore corrispondente in base dieci
`12`	
`012`	
`0x12`	

81.2.3.2 Esercizio

«

Indicare i tipi delle costanti elencate nella tabella successiva:

Costante	Tipo corrispondente
`12`	
`12U`	
`12L`	
`1.2`	
`1.2L`	

81.2.4 Caratteri privi di rappresentazione grafica

«

I caratteri privi di rappresentazione grafica possono essere indicati, principalmente, attraverso tre tipi di notazione: ottale, esadecimale e simbolica. In tutti i casi si utilizza la barra obliqua inversa (‘****’) come carattere di escape, cioè come simbolo per annunciare che ciò che segue immediatamente deve essere interpretato in modo particolare.

La notazione ottale usa la forma ‘***ooo***’, dove ogni lettera ***o*** rappresenta una cifra ottale. A questo proposito, è opportuno notare che se la dimensione di un carattere fosse superiore ai fatidici 8 bit, occorrerebbero probabilmente più cifre (una cifra ottale rappresenta un gruppo di 3 bit).

La notazione esadecimale usa la forma '**\x*hh***', dove ***h*** rappresenta una cifra esadecimale. Anche in questo caso vale la considerazione per cui ci vogliono più di due cifre esadecimali per rappresentare un carattere più lungo di 8 bit.

Dovrebbe essere logico, ma è il caso di osservare che la corrispondenza dei caratteri con i rispettivi codici numerici dipende dalla codifica utilizzata. Generalmente si utilizza la codifica ASCII, riportata anche nella sezione 47.7.5 (in questa fase introduttiva si omette di trattare la rappresentazione dell'insieme di caratteri universale).

La notazione simbolica permette di fare riferimento facilmente a codici di uso comune, quali <*CR*>, <*HT*>,... Inoltre, questa notazione permette anche di indicare caratteri che altrimenti verrebbero interpretati in maniera differente dal compilatore. La tabella successiva riporta i vari tipi di rappresentazione delle costanti carattere attraverso codici di escape.

Tabella 81.26. Elenco dei modi di rappresentazione delle costanti carattere attraverso codici di escape.

Codice	ASCII	Altra codifica
ooo	Notazione ottale in base alla codifica.	idem
\\x***hh***	Notazione esadecimale in base alla codifica.	idem
\\\\	Una singola barra obliqua inversa ('\').	idem
\\'	Un apice singolo destro.	idem
\\"	Un apice doppio.	idem

Codice	ASCII	Altra codifica
`\?`	Un punto interrogativo (per impedire che venga inteso come parte di una sequenza triplice, o *trigraph*).	idem
`\0`	Il codice <*NUL*>.	Il carattere nullo (con tutti i bit a zero).
`\a`	Il codice <*BEL*> (*bell*).	Il codice che, rappresentato sullo schermo o sulla stampante, produce un segnale acustico (*alert*).
`\b`	Il codice <*BS*> (*backspace*).	Il codice che fa arretrare il cursore di una posizione nella riga (*backspace*).
`\f`	Il codice <*FF*> (*form feed*).	Il codice che fa avanzare il cursore all'inizio della prossima pagina logica (*form feed*).
`\n`	Il codice <*LF*> (*line feed*).	Il codice che fa avanzare il cursore all'inizio della prossima riga logica (*new line*).
`\r`	Il codice <*CR*> (*carriage return*).	Il codice che porta il cursore all'inizio della riga attuale (*carriage return*).
`\t`	Una tabulazione orizzontale (<*HT*>).	Il codice che porta il cursore all'inizio della prossima tabulazione orizzontale (*horizontal tab*).
`\v`	Una tabulazione verticale (<*VT*>).	Il codice che porta il cursore all'inizio della prossima tabulazione verticale (*vertical tab*).

A parte i casi di '***ooo***' e '**\x*hh***', le altre sequenze esprimono un concetto, piuttosto di un codice numerico preciso. All'origine del linguaggio C, tutte le altre sequenze corrispondono a un solo carattere non stampabile, ma attualmente non è più garantito che sia così. In particolare, la sequenza '**\n**', nota come *new-line*, potrebbe essere espressa in modo molto diverso rispetto al codice <*LF*> tradizionale. Questo concetto viene comunque approfondito a proposito della gestione dei flussi di file.

In varie situazioni, il linguaggio C standard ammette l'uso di sequenze composte da due o tre caratteri, note come *digraph* e *trigraph* rispettivamente; ciò in sostituzione di simboli la cui rappresentazione, in quel contesto, può essere impossibile. In un sistema che ammetta almeno l'uso della codifica ASCII per scrivere il file sorgente, con l'ausilio di una tastiera comune, non c'è alcun bisogno di usare tali artifici, i quali, se usati, renderebbero estremamente complessa la lettura del sorgente. Pertanto, è bene sapere che esistono queste cose, ma è meglio non usarle mai. Tuttavia, siccome le sequenze a tre caratteri (*trigraph*) iniziano con una coppia di punti interrogativi, se in una stringa si vuole rappresentare una sequenza del genere, per evitare che il compilatore la traduca diversamente, è bene usare la sequenza '**\?\?**', come suggerisce la tabella.

Nell'esempio introduttivo appare già la notazione '**\n**' per rappresentare l'inserzione di un codice di interruzione di riga alla fine del messaggio di saluto:

```
...
    printf ("Ciao mondo!\n");
...
```

Senza di questo, il cursore resterebbe a destra del messaggio alla fine dell'esecuzione di quel programma, ponendo lì l'invito.

81.2.5 Valore numerico delle costanti carattere

Il linguaggio C distingue tra i caratteri di un insieme fondamentale e ridotto, da quelli dell'insieme di caratteri universale (ISO 10646). Il gruppo di caratteri ridotto deve essere rappresentabile in una variabile '**char**' (descritta nelle sezioni successive) e può essere gestito direttamente in forma numerica, se si conosce il codice corrispondente a ogni simbolo (di solito si tratta della codifica ASCII).

Se si può essere certi che nella codifica le lettere dell'alfabeto latino siano disposte esattamente in sequenza (come avviene proprio nella codifica ASCII), si potrebbe scrivere '**'A'+1**' e ottenere l'equivalente di '**'B'**'. Tuttavia, lo standard prescrive che sia garantito il funzionamento solo per le cifre numeriche. Pertanto, per esempio, '**'0'+3**' (zero espresso come carattere, sommato a un tre numerico) deve essere equivalente a '**'3'**' (ovvero un «tre» espresso come carattere).

Listato 81.28. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/5EZCPetn* , *http://ideone.com/KuRkv* .

```
#include <stdio.h>

int main (void)
{
    char c;
```

```
    for (c = '0'; c <= 'Z'; c++)
      {
        printf ("%c", c);
      }
    printf ("\n");
    getchar ();
    return 0;
}
```

Il programma di esempio che si vede nel listato appena mostrato, se prodotto per un ambiente in cui si utilizza la codifica ASCII, genera il risultato seguente:

```
0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

81.2.5.1 Esercizio

Indicare che valore si ottiene dalle espressioni elencate nella tabella successiva. Il primo caso appare risolto, come esempio:

Espressione	Costante carattere equivalente
'3'+1	'4'
'3'-2	
'5'+4	

81.2.6 Campo di azione delle variabili

«

Il campo di azione delle variabili in C viene determinato dalla posizione in cui queste vengono dichiarate e dall'uso di qualificatori particolari. Nella fase iniziale dello studio del linguaggio basta considerare, approssimativamente, che quanto dichiarato all'interno di una funzione ha valore locale per la funzione stessa, mentre quanto dichiarato al di fuori, ha valore globale per tutto il file. Pertanto, in questo capitolo si usano genericamente le definizioni di «variabile locale» e «variabile globale», senza affrontare altre questioni. Nella sezione 66.3 viene trattato questo argomento con maggiore dettaglio.

81.2.7 Dichiarazione delle variabili

«

La dichiarazione di una variabile avviene specificando il tipo e il nome della variabile, come nell'esempio seguente dove viene creata la variabile ***numero*** di tipo intero:

```
int numero;
```

La variabile può anche essere inizializzata contestualmente, assegnandole un valore, come nell'esempio seguente in cui viene dichiarata la stessa variabile ***numero*** con il valore iniziale di 1000:

```
int numero = 1000;
```

Una costante è qualcosa che non varia e generalmente si rappresenta attraverso una notazione che ne definisce il valore, ovvero attraverso una costante letterale. Tuttavia, a volte può essere più comodo definire una costante in modo simbolico, come se fosse una variabile, per facilitarne l'utilizzo e la sua identificazione all'interno del programma. Si ottiene questo con il modificatore '**`const`**'. Ovviamente, è obbligatorio inizializzala contestualmente alla sua dichiarazione.

L'esempio seguente dichiara la costante simbolica ***pi*** con il valore del π:

```
const float pi = 3.14159265;
```

Le costanti simboliche di questo tipo, sono delle variabili per le quali il compilatore non concede che avvengano delle modifiche; pertanto, il programma eseguibile che si ottiene potrebbe essere organizzato in modo tale da caricare questi dati in segmenti di memoria a cui viene lasciato poi il solo permesso di lettura.

Tradizionalmente, l'uso di costanti simboliche di questo tipo è stato limitato, preferendo delle ***macro-variabili*** definite e gestite attraverso il precompilatore (come viene descritto nella sezione 66.2). Tuttavia, un compilatore ottimizzato è in grado di gestire al meglio le costanti definite nel modo illustrato dall'esempio, utilizzando anche dei valori costanti letterali nella trasformazione in linguaggio assemblatore, rendendo così indifferente, dal punto di vista del risultato, l'alternativa delle macro-variabili. Pertanto, la stessa guida *GNU coding standards* chiede di definire le costanti come variabili-costanti, attraverso il modificatore '**`const`**'.

81.2.7.1 Esercizio

«

Indicare le istruzioni di dichiarazione delle variabili descritte nella tabella successiva. I primi due casi appaiono risolti, come esempio:

Descrizione	Dichiarazione corrispondente
Variabile «a» in qualità di carattere senza segno.	`unsigned char x;`
Variabile «b» in qualità di carattere senza segno, inizializzata al valore 21.	`unsigned char x = 21;`

Descrizione	Dichiarazione corrispondente
Variabile «d» in qualità di intero normale (con segno).	
Variabile «e» in qualità di intero più grande del solito, senza segno, inizializzata al valore 2111.	
Variabile «f» inizializzata al valore 21,11.	
Costante simbolica «g» inizializzata al valore 21,11.	

81.2.8 Il tipo indefinito: «void»

«

Lo standard del linguaggio C definisce un tipo particolare di valore, individuato dalla parola chiave '**`void`**'. Si tratta di un valore indefinito che a seconda del contesto può rappresentare il nulla o qualcosa da ignorare esplicitamente. A ogni modo, volendo ipotizzare una variabile di tipo '**`void`**', questa occuperebbe zero byte.

81.3 Operatori ed espressioni del linguaggio C

«

L'operatore è qualcosa che esegue un qualche tipo di funzione, su uno o più operandi, restituendo un valore.[9] Il valore restituito è di tipo diverso a seconda degli operandi utilizzati. Per esempio, la somma di due interi genera un risultato intero. Gli operandi descritti di seguito sono quelli più comuni e importanti.

Le espressioni sono formate spesso dalla valutazione di sottoespressioni (espressioni più piccole). Va osservato che ci sono circostanze in cui il contesto non impone che ci sia un solo ordine possibile nella valutazione delle sottoespressioni, ma il programmatore deve tenere conto di questa possibilità, per evitare che il risultato dipenda dalle scelte non prevedibili del compilatore.

Tabella 81.35. Ordine di precedenza tra gli operatori previsti nel linguaggio C. Gli operatori sono raggruppati a livelli di priorità equivalente, partendo dall'alto con la priorità maggiore, scendendo progressivamente alla priorità minore. Le variabili ***a***, ***b*** e ***c*** rappresentano la collocazione delle sottoespressioni da considerare ed esprimono l'ordine di associatività: prima ***a***, poi ***b***, poi ***c***.

Operatori	Annotazioni
`(`***a***`)` `[`***a***`]` ***a***`->`***b*** ***a***`.`***b***	Le parentesi tonde usate per raggruppare una porzione di espressione hanno la precedenza su ogni altro operatore. Le parentesi quadre riguardano gli array; gli operatori '**->**' e '**.**', riguardano le strutture e le unioni.
`!`***a*** `~`***a*** `++`***a*** `--`***a*** `+`***a*** `-`***a*** `*`***a*** `&`***a*** `(`***tipo***`)` `sizeof` ***a***	Gli operatori '**+**' e '**-**' di questo livello sono da intendersi come «unari», ovvero si riferiscono al segno di quanto appare alla loro destra. Gli operatori '*****' e '**&**' di questo livello riguardano la gestione dei puntatori; le parentesi tonde si riferiscono al cast.

Operatori	Annotazioni
`a*b` `a/b` `a%b`	Moltiplicazione, divisione e resto della divisione intera.
`a+b` `a-b`	Somma e sottrazione.
`a<<b` `a>>b`	Scorrimento binario.
`a<b` `a<=b` `a>b` `a=>b`	Confronto.
`a==b` `a!=b`	Confronto.
`a&b`	AND bit per bit.
`a^b`	XOR bit per bit.
`a|b`	OR bit per bit.
`a&&b`	AND nelle espressioni logiche.
`a||b`	OR nelle espressioni logiche.
`c?b:a`	Operatore condizionale
`b=a` `b+=a` `b-=a` `b*=a` `b/=a` `b%=a` `b&=a` `b^=a` `b|=a` `b<<=a` `b>>=a`	Operatori di assegnamento.
`a, b`	Sequenza di espressioni (espressione multipla).

81.3.1 Tipo del risultato di un'espressione

Un'espressione è un qualche cosa composto da operandi e da operatori, che nel complesso si traduce in un qualche risultato. Per esempio, '`5+6`' è un'espressione aritmetica che si traduce nel numero 11. Così come le variabili, le costanti simboliche e le costanti letterali, hanno un tipo, con il quale si definisce in che modo vengono rappresentate in memoria, anche il risultato delle espressioni ha un tipo, in quanto tale risultato deve poi essere rappresentabile in memoria in qualche modo.

La regola che definisce di che tipo è il risultato di un'espressione è piuttosto articolata, ma in generale è sufficiente rendersi conto che si tratta della scelta più logica in base al contesto. Per esempio, l'espressione già vista, '`5+6`', essendo la somma di due interi con segno, dovrebbe dare come risultato un intero con segno. Nello stesso modo, un'espressione del tipo '`5.1-6.3`', essendo costituita da operandi in virgola mobile (precisamente '`double`'), dà il risultato −1,2, rappresentato sempre in virgola mobile (sempre '`double`'). Va osservato che la regola di principio vale anche per le divisioni, per cui '`11/2`' dà 5, di tipo intero ('`int`'), perché per avere un risultato in virgola mobile occorrerebbe invece scrivere '`11.0/2.0`'.

Si osservi che se in un'espressione si mescolano operandi interi assieme a operandi in virgola mobile, il risultato dell'espressione dovrebbe essere di tipo a virgola mobile. Per esempio, '`5+6.3`' dà il valore 11,3, in virgola mobile ('`double`'). Inoltre, se gli operandi hanno tra loro un rango differente, dovrebbe prevalere il rango maggiore.

81.3.1.1 Esercizio

«

Indicare il tipo che si dovrebbe ottenere dalla valutazione delle espressioni proposte. Il primo caso appare risolto, come esempio:

Espressione	Tipo che dovrebbe avere il risultato dell'espressione
`4+3.5`	`double`
`4+4`	
`4/3`	
`4.0/3`	
`4L*3`	

81.3.2 Operatori aritmetici

«

Gli operatori che intervengono su valori numerici sono elencati nella tabella successiva. Per dare un significato alle descrizioni della tabella, occorre tenere presenta una caratteristica importante del linguaggio, per la quale, la maggior parte delle espressioni restituisce un valore. Per esempio, '**`b = a = 1`**' fa sì che la variabile ***a*** ottenga il valore 1 e che, successivamente, la variabile ***b*** ottenga il valore di ***a***. In questo senso, al problema dell'ordine di precedenza dei vari operatori si aggiunge anche l'ordine in cui le espressioni restituiscono un valore. Per esempio, '**`d = e++`**' comporta l'incremento di una unità del contenuto della variabile ***e***, ma ciò solo **dopo** averne restituito il valore che viene assegnato alla variabile ***d***. Pertanto, se inizialmente la variabile ***e*** contiene il valore 1, dopo l'elaborazione

dell'espressione completa, la variabile ***d*** contiene il valore 1, mentre la variabile ***e*** contiene il valore 2.

Tabella 81.37. Elenco degli operatori aritmetici e di quelli di assegnamento relativi a valori numerici.

Operatore e operandi	Descrizione
++*op*	Incrementa di un'unità l'operando prima che venga restituito il suo valore.
op++	Incrementa di un'unità l'operando dopo averne restituito il suo valore.
--*op*	Decrementa di un'unità l'operando prima che venga restituito il suo valore.
op--	Decrementa di un'unità l'operando dopo averne restituito il suo valore.
+*op*	Non ha alcun effetto.
-*op*	Inverte il segno dell'operando (prima di restituirne il valore).
op1 + *op2*	Somma i due operandi.
op1 - *op2*	Sottrae dal primo il secondo operando.
op1 * *op2*	Moltiplica i due operandi.
op1 / *op2*	Divide il primo operando per il secondo.
op1 % *op2*	Calcola il resto della divisione tra il primo e il secondo operando, i quali devono essere costituiti da valori interi.
var = *valore*	Assegna alla variabile il valore alla destra.
op1 += *op2*	*op1* = (*op1* + *op2*)

Operatore e operandi	Descrizione
op1 `-=` *op2*	*op1* `=` (*op1* `-` *op2*)
op1 `*=` *op2*	*op1* `=` (*op1* `*` *op2*)
op1 `/=` *op2*	*op1* `=` (*op1* `/` *op2*)
op1 `%=` *op2*	*op1* `=` (*op1* `%` *op2*)

81.3.2.1 Esercizio

«

Osservando i pezzi di codice indicati, si scriva il valore contenuto nelle variabili a cui si assegna un valore, attraverso l'elaborazione di un'espressione. Il primo caso appare risolto, come esempio:

Codice	Valore contenuto nelle variabili dopo l'esecuzione del codice mostrato
`int a = 3;` `int b;` `b = a++;`	***a*** contiene 4; ***b*** contiene 3.
`int a = 3;` `int b;` `b = --a;`	
`int a = 3;` `int b = 2;` `b = a + b;`	

Codice	Valore contenuto nelle variabili dopo l'esecuzione del codice mostrato
`int a = 7;` `int b = 2;` `b = a % b;`	
`int a = 7;` `int b;` `b = (a = a * 2);`	
`int a = 3;` `int b = 2;` `b += a;`	
`int a = 7;` `int b = 2;` `b %= a;`	
`int a = 7;` `int b;` `b = (a *= 2);`	

81.3.2.2 Esercizio

Scrivere diversi programmi per verificare l'esercizio precedente. Viene proposto il primo, da usare come modello per gli altri.

Listato 81.39. Per svolgere l'esercitazione attraverso un servizio *pastebin*: *http://codepad.org/ZHmO0ycC* , *http://ideone.com/Rv6o1* .

```
#include <stdio.h>
int main (void)
{
    int a = 3;
    int b;
    b = a++;
    printf ("a contiene %d;\n", a);
    printf ("b contiene %d.\n", b);
    getchar ();
    return 0;
}
```

81.3.3 Operatori di confronto

«

Gli operatori di confronto determinano la relazione tra due operandi. Il risultato dell'espressione composta da due operandi posti a confronto è un numero intero ('**int**') e precisamente si ottiene uno se il confronto è valido e zero in caso contrario. Gli operatori di confronto sono elencati nella tabella successiva.

Il linguaggio C non ha una rappresentazione specifica per i valori booleani *Vero* e *Falso*,[10] ma si limita a interpretare un valore pari a zero come *Falso* e un valore diverso da zero come *Vero*. Va osservato, quindi, che il numero usato come valore booleano, può essere espresso anche in virgola mobile, benché sia preferibile di gran lunga un intero normale.

Tabella 81.40. Elenco degli operatori di confronto. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
op1 `==` ***op2***	1 (*Vero*) se gli operandi si equivalgono.
op1 `!=` ***op2***	1 (*Vero*) se gli operandi sono differenti.
op1 `<` ***op2***	1 (*Vero*) se il primo operando è minore del secondo.
op1 `>` ***op2***	1 (*Vero*) se il primo operando è maggiore del secondo.
op1 `<=` ***op2***	1 (*Vero*) se il primo operando è minore o uguale al secondo.
op1 `>=` ***op2***	1 (*Vero*) se il primo operando è maggiore o uguale al secondo.

81.3.3.1 Esercizio

Osservando i pezzi di codice indicati, si scriva il valore contenuto nella variabile ***c***. Il primo caso appare risolto, come esempio:

Codice	Valore contenuto nella variabile ***c*** l'esecuzione del codice mostrato
`int a = 5;` `signed char b = -4;` `int c = a > b;`	***c*** contiene 1.
`int a = 4 + 1;` `signed char b = 5;` `int c = a == b;`	

Codice	Valore contenuto nella variabile ***c*** l'esecuzione del codice mostrato
`int a = 4 + 1;` `signed char b = 5;` `int c = a != b;`	
`unsigned int a = 4 + 3;` `signed char b = -5;` `int c = a >= b;`	
`unsigned int a = 4 + 3;` `signed char b = -5;` `int c = a <= b;`	

81.3.3.2 Esercizio

«

Scrivere diversi programmi per verificare l'esercizio precedente. Viene proposto il primo, da usare come modello per gli altri.

Listato 81.42. Per svolgere l'esercitazione attraverso un servizio *pastebin*: *http://codepad.org/a3E5IGIT* , *http://ideone.com/0zob2* .

```
#include <stdio.h>
int main (void)
{
    int a = 5;
    signed char b = -4;
    int c = a > b;
    printf ("(%d > %d) produce %d\n", a, b, c);
    getchar ();
    return 0;
}
```

81.3.4 Operatori logici

Quando si vogliono combinare assieme diverse espressioni logiche, comprendendo in queste anche delle variabili che contengono un valore booleano, si utilizzano gli operatori logici (noti normalmente come: AND, OR, NOT, ecc.). Il risultato di un'espressione logica complessa è quello dell'ultima espressione elementare valutata effettivamente, in quanto le sottoespressioni che non possono cambiare l'esito della condizione complessiva non vengono valutate. Gli operatori logici sono elencati nella tabella successiva.

Tabella 81.43. Elenco degli operatori logici. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
`!` ***op***	Inverte il risultato logico dell'operando.
op1 `&&` ***op2***	Se il risultato del primo operando è *Falso* non valuta il secondo.
op1 `||` ***op2***	Se il risultato del primo operando è *Vero* non valuta il secondo.

Un tipo particolare di operatore logico è l'operatore condizionale, il quale permette di eseguire espressioni diverse in relazione al risultato di una condizione. La sua sintassi si esprime nel modo seguente:

```
condizione ? espressione1 : espressione2
```

In pratica, se l'espressione che rappresenta la condizione si avvera, viene eseguita la prima espressione che segue il punto interrogativo,

altrimenti viene eseguita quella che segue i due punti.

81.3.4.1 Esercizio

«

Osservando i pezzi di codice indicati, si scriva il valore contenuto nella variabile ***c***. Il primo caso appare risolto, come esempio:

Codice	Valore contenuto nella variabile ***c*** dopo l'esecuzione del codice mostrato
`int a = 5;` `signed char b = -4;` `int c = ! (a > b);`	***c*** contiene 0.
`int a = 4;` `signed char b = (3 < 5);` `int c = a && b;`	
`int a = 4;` `signed char b = (3 < 5);` `int c = a || b;`	
`unsigned int a = 4 + 3;` `signed char b = -5;` `int c = (b > 0) || (a > b);`	

81.3.5 Operatori binari

«

Il linguaggio C consente di eseguire alcune operazioni binarie, sui **valori interi**, come spesso è possibile fare con un linguaggio assemblatore, anche se non è possibile interrogare degli indicatori (*flag*) che informino sull'esito delle azioni eseguite. Sono disponibili le operazioni elencate nella tabella successiva.

Tabella 81.45. Elenco degli operatori binari. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
op1 `&` ***op2***	AND bit per bit.
op1 `|` ***op2***	OR bit per bit.
op1 `^` ***op2***	XOR bit per bit (OR esclusivo).
op1 `<<` ***op2***	Scorrimento a sinistra di ***op2*** bit. A destra vengono aggiunti bit a zero
op1 `>>` ***op2***	Scorrimento a destra di ***op2*** bit. Il valore dei bit aggiunti a sinistra potrebbe tenere conto del segno.
`~`***op1***	Complemento a uno.
op1 `&=` ***op2***	***op1*** `=` `(`***op1*** `&` ***op2***`)`
op1 `|=` ***op2***	***op1*** `=` `(`***op1*** `|` ***op2***`)`
op1 `^=` ***op2***	***op1*** `=` `(`***op1*** `^` ***op2***`)`
op1 `<<=` ***op2***	***op1*** `=` `(`***op1*** `<<` ***op2***`)`
op1 `>>=` ***op2***	***op1*** `=` `(`***op1*** `>>` ***op2***`)`
op1 `~=` ***op2***	***op1*** `=` `~`***op2***

A seconda del compilatore e della piattaforma, lo scorrimento a destra potrebbe essere di tipo aritmetico, ovvero potrebbe tenere conto del segno. Pertanto, non potendo fare sempre affidamento su questa

ipotesi, è prudente far sì che i valori di cui si fa lo scorrimento a destra siano sempre senza segno, o comunque positivi.

Per aiutare a comprendere il meccanismo vengono mostrati alcuni esempi. In particolare si utilizzano due operandi di tipo '**char**' (a 8 bit) senza segno: ***a*** contenente il valore 42, pari a 00101010_2; ***b*** contenente il valore 51, pari a 00110011_2.

c = a & b

00101010_2	(42_{10})	AND
00110011_2	(51_{10})	=
00100010_2	(34_{10})	

c = a | b

00101010_2	(42_{10})	OR
00110011_2	(51_{10})	=
00111011_2	(59_{10})	

c = a ^ b

00101010_2	(42_{10})	XOR
00110011_2	(51_{10})	=
00011001_2	(25_{10})	

Lo scorrimento, invece, viene mostrato sempre solo per una singola unità: ***a*** contenente sempre il valore 42; ***b*** contenente il valore 1.

c = a << b

00101010_2	(42_{10})	<<
00000001_2	(1_{10})	=
01010100_2	(84_{10})	

c = a >> b

00101010_2	(42_{10})	>>
00000001_2	(1_{10})	=
00010101_2	(21_{10})	

c = ~a

00101010_2	(42_{10})	
11010101_2	(213_{10})	

81.3.5.1 Esercizio

«

Osservando i pezzi di codice indicati, si scriva il valore contenuto nella variabile ***c***. L'architettura a cui ci si riferisce prevede l'uso del complemento a due per la rappresentazione dei numeri negativi e lo scorrimento a destra è di tipo aritmetico (in quanto preserva il segno). I primi casi appaiono risolti, come esempio:

Codice	Valore contenuto nella variabile ***c*** dopo l'esecuzione del codice mostrato
`int a = -20;` `int c = a >> 1;`	***c*** contiene −10.
`int a = 5;` `int b = 12;` `int c = a & b;`	
`int a = 5;` `int b = 12;` `int c = a | b;`	
`int a = 5;` `int b = 12;` `int c = a ^ b;`	
`int a = 5;` `int c = a << 1;`	
`int a = 21;` `int c = a >> 1;`	

81.3.5.2 Esercizio

Scrivere diversi programmi per verificare l'esercizio precedente. Viene proposto un esempio, riferito a un caso che non appare nell'esercizio precedente, con cui si ottiene il complemento a uno.

Listato 81.49. Per svolgere l'esercitazione attraverso un servizio *pastebin*: *http://codepad.org/CqxVMIHG* , *http://ideone.com/iIEL0* .

```
#include <stdio.h>
int main (void)
{
    int a = 21;
    int c = ~a;
    printf ("c contiene %d\n", c);
    getchar ();
    return 0;
}
```

81.3.6 Conversione di tipo

«

Quando si assegna un valore a una variabile, nella maggior parte dei casi, il contesto stabilisce il tipo di questo valore in modo corretto. Di fatto, è il tipo della variabile ricevente che stabilisce la conversione necessaria. Tuttavia, il problema si pone anche durante la valutazione di un'espressione.

Per esempio, '**`5/4`**' viene considerata la divisione di due interi e, di conseguenza, l'espressione restituisce un valore intero, cioè 1. Diverso sarebbe se si scrivesse '**`5.0/4.0`**', perché in questo caso si tratterebbe della divisione tra due numeri a virgola mobile (per la precisione, di tipo '**`double`**') e il risultato è un numero a virgola mobile.

Quando si pone il problema di risolvere l'ambiguità si utilizza esplicitamente la conversione del tipo, attraverso un ***cast***:

```
(tipo) espressione
```

In pratica, si deve indicare tra parentesi tonde il nome del tipo di dati in cui deve essere convertita l'espressione che segue. Il problema sta nella precedenza che ha il cast nell'insieme degli altri operatori e in generale conviene utilizzare altre parentesi per chiarire la relazione che ci deve essere.

```
int x = 10;
double y;
…
y = (double) x/9;
```

In questo caso, la variabile intera ***x*** viene convertita nel tipo '**`double`**' (a virgola mobile) prima di eseguire la divisione. Dal momento che il cast ha precedenza sull'operazione di divisione, non si pongono problemi, inoltre, la divisione avviene trasformando implicitamente il 9 intero in un 9,0 di tipo '**`double`**'. In pratica, l'operazione avviene utilizzando valori '**`double`**' e restituendo un risultato '**`double`**'.

81.3.6.1 Esercizio

Indicare il tipo che si dovrebbe ottenere dalla valutazione delle espressioni proposte e il risultato effettivo. Il primo caso appare risolto, come esempio:

Espressione	Tipo che dovrebbe avere il risultato dell'espressione
`4+((double) 3)`	`(double) 7`
`(int) (4.4+4.9)`	
`(double) 4/3`	

Espressione	Tipo che dovrebbe avere il risultato dell'espressione
`((double) 4)/3`	
`4 * ((long int) 3)`	

81.3.7 Espressioni multiple

Un'istruzione, cioè qualcosa che termina con un punto e virgola, può contenere diverse espressioni separate da una virgola. Tenendo presente che in C l'assegnamento di una variabile è anche un'espressione, la quale restituisce il valore assegnato, si veda l'esempio seguente:

```
int x;
int y;
…
y = 10, x = 20, y = x*2;
```

L'esempio mostra un'istruzione contenente tre espressioni: la prima assegna a y il valore 10, la seconda assegna a x il valore 20 e la terza sovrascrive y assegnandole il risultato del prodotto $x \cdot 2$. In pratica, alla fine la variabile y contiene il valore 40 e x contiene 20.

Un'espressione multipla, come quella dell'esempio, restituisce il valore dell'ultima a essere eseguita. Tornando all'esempio, visto, gli si può apportare una piccola modifica per comprendere il concetto:

```
int x;
int y;
int z;
…
z = (y = 10, x = 20, y = x*2);
```

La variabile *z* si trova a ricevere il valore dell'espressione '*y = x*2*', perché è quella che viene eseguita per ultima nel gruppo raccolto tra parentesi.

A proposito di «espressioni multiple» vale la pena di ricordare ciò che accade con gli assegnamenti multipli, con l'esempio seguente:

```
y = x = 10;
```

Qui si vede l'assegnamento alla variabile *y* dello stesso valore che viene assegnato alla variabile *x*. In pratica, sia *x* che *y* contengono alla fine il numero 10, perché le precedenze sono tali che è come se fosse scritto: '*y = (x = 10)*'.

81.3.7.1 Esercizio

Osservando i pezzi di codice indicati, si scriva il valore contenuto nella variabile *c*. Il primo caso appare risolto, come esempio:

Codice	Valore contenuto nella variabile *c* dopo l'esecuzione del codice mostrato
`int a = -20;` `int b = 10;` `int c = (a *= 2, b += 10, c = a + b);`	*c* contiene −20.
`int a = -20;` `int b = 10;` `int c = (b = 2, b = a, c = a + b);`	

Codice	Valore contenuto nella variabile ***c*** dopo l'esecuzione del codice mostrato
`int a = -20;` `int b = 10;` `int c = (b = 2, b = a++, c = a + b);`	
`int a = -20;` `int b = 10;` `int c = (b = 2, b = ++a, c = a + b);`	

81.3.7.2 Esercizio

«

Scrivere diversi programmi per verificare l'esercizio precedente. Viene proposto un esempio, riferito al caso iniziale risolto.

Listato 81.56. Per svolgere l'esercitazione attraverso un servizio *pastebin*: *http://codepad.org/v4Aj19Ae19* , *http://ideone.com/ZTA1L* .

```
#include <stdio.h>
int main (void)
{
    int a = -20;
    int b = 10;
    int c = (a *= 2, b += 10, c = a + b);
    printf ("c contiene %d\n", c);
    getchar ();
    return 0;
}
```

81.4 Strutture di controllo di flusso del linguaggio C

Il linguaggio C gestisce praticamente tutte le strutture di controllo di flusso degli altri linguaggi di programmazione, compreso *go-to* che comunque è sempre meglio non utilizzare e qui, volutamente, non viene presentato.

Le strutture di controllo permettono di sottoporre l'esecuzione di una parte di codice alla verifica di una condizione, oppure permettono di eseguire dei cicli, sempre sotto il controllo di una condizione. La parte di codice che viene sottoposta a questo controllo, può essere una singola istruzione, oppure un gruppo di istruzioni (precisamente si chiamerebbe istruzione composta). Nel secondo caso, è necessario delimitare questo gruppo attraverso l'uso delle parentesi graffe.

Dal momento che è comunque consentito di realizzare un gruppo di istruzioni che in realtà ne contiene una sola, probabilmente è meglio utilizzare sempre le parentesi graffe, in modo da evitare equivoci nella lettura del codice. Dato che le parentesi graffe sono usate nel codice C, se queste appaiono nei modelli sintattici indicati, significa che fanno parte delle istruzioni e non della sintassi.

Negli esempi, i rientri delle parentesi graffe seguono le indicazioni della guida *GNU coding standards*.

81.4.1 Struttura condizionale: «if»

La struttura condizionale è il sistema di controllo fondamentale dell'andamento del flusso delle istruzioni.

```
if (condizione) istruzione
```

```
if (condizione) istruzione else istruzione
```

Se la condizione si verifica, viene eseguita l'istruzione o il gruppo di istruzioni che segue; quindi il controllo passa alle istruzioni successive alla struttura. Se viene utilizzata la sotto-struttura che si articola a partire dalla parola chiave '**else**', nel caso non si verifichi la condizione, viene eseguita l'istruzione che ne dipende. Sotto vengono mostrati alcuni esempi completi, dove è possibile variare il valore assegnato inizialmente alla variabile ***importo*** per verificare il comportamento delle istruzioni.

Listato 81.57. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/BbrdEx7f* , *http://ideone.com/qZ30j* .

```
#include <stdio.h>
int main (void)
{
    int importo;
    importo = 10001;
    if (importo > 10000) printf ("L'offerta è vantaggiosa\n");
    getchar ();
    return 0;
}
```

Listato 81.58. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/5OQZsFk1* , *http://ideone.com/9s9DH* .

```
#include <stdio.h>
int main (void)
{
    int importo;
```

```
    int memorizza;
    importo = 10001;
    if (importo > 10000)
      {
        memorizza = importo;
        printf ("L'offerta è vantaggiosa\n");
      }
    else
      {
        printf ("Lascia perdere\n");
      }
    getchar ();
    return 0;
}
```

L'esempio successivo, in particolare, mostra un modo grazioso per allineare le sottocondizioni, senza eccedere negli annidamenti.

Listato 81.59. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/99OA99Zbff*, *http://ideone.com/aQKgZ*.

```
#include <stdio.h>
int main (void)
{
    int importo;
    int memorizza;
    importo = 10001;
    if (importo > 10000)
      {
        memorizza = importo;
        printf ("L'offerta è vantaggiosa\n");
      }
    else if (importo > 5000)
```

```
        {
          memorizza = importo;
          printf ("L'offerta è accettabile\n");
        }
      else
        {
          printf ("Lascia perdere\n");
        }
      getchar ();
      return 0;
}
```

81.4.1.1 Esercizio

«

Partendo dalla struttura successiva, si scriva un programma che, in base al valore della variabile ***x***, mostri dei messaggi differenti: se ***x*** è inferiore a 1000 oppure è maggiore di 10000, si viene avvisati che il valore non è valido; se invece ***x*** è valido, se questo è maggiore di 5000, si viene avvisati che «il livello è alto», se invece fosse inferiore si viene avvisati che «il livello è basso»; infine, se il valore è pari a 5000, si viene avvisati che il livello è ottimale.

Listato 81.60. Per svolgere l'esercitazione si può usare eventualmente un servizio *pastebin*: *http://codepad.org/0vfX5Un9* , *http://ideone.com/gVhow* .

```
#include <stdio.h>
int main (void)
{
    int x;
    x = 5000;

    if ((x < 1000) || (x > 10000))
```

```
        {
          printf ("Il valore di x non è valido!\n");
        }
      else if …
        {
          …
          …
          …
        }
      getchar ();
      return 0;
}
```

81.4.1.2 Esercizio

Si osservi il programma successivo è si indichi cosa viene visualizzato alla sua esecuzione, spiegando il perché.

```
#include <stdio.h>
int main (void)
{
    int x;
    x = -1;

    if (x)
      {
        printf ("Sono felice :-)\n");
      }
    else
      {
        printf ("Sono triste :-(\n");
      }
    getchar ();
    return 0;
}
```

81.4.2 Struttura di selezione: «switch»

La struttura di selezione che si attua con l'istruzione '**`switch`**', è un po' troppo complessa per essere rappresentata facilmente attraverso uno schema sintattico. In generale, questa struttura permette di **saltare** a una certa posizione interna alla struttura, in base al risultato di un'espressione. L'esempio seguente mostra la visualizzazione del nome del mese, in base al valore di un intero.

Listato 81.62. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/UOMoRmPm* , *http://ideone.com/Z9PqE* .

```
#include <stdio.h>
int main (void)
{
```

```
    int mese;
    mese = 11;

    switch (mese)
      {
        case 1: printf ("gennaio\n"); break;
        case 2: printf ("febbraio\n"); break;
        case 3: printf ("marzo\n"); break;
        case 4: printf ("aprile\n"); break;
        case 5: printf ("maggio\n"); break;
        case 6: printf ("giugno\n"); break;
        case 7: printf ("luglio\n"); break;
        case 8: printf ("agosto\n"); break;
        case 9: printf ("settembre\n"); break;
        case 10: printf ("ottobre\n"); break;
        case 11: printf ("novembre\n"); break;
        case 12: printf ("dicembre\n"); break;
      }
    getchar ();
    return 0;
}
```

Come si vede, dopo l'istruzione con cui si emette il nome del mese attraverso lo standard output, viene richiesto di uscire dalla struttura, attraverso l'istruzione '**`break`**', perché altrimenti si passerebbe all'esecuzione delle istruzioni del caso successivo, se presente. Sulla base di questo principio, un gruppo di casi può essere raggruppato assieme, quando si vuole che ognuno di questi esegua lo stesso insieme di istruzioni.

Listato 81.63. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/p3uFTLyn* , *http://ideone.com/glcnI* .

```
#include <stdio.h>
int main (void)
{
    int anno;
    int mese;
    int giorni;
    anno = 2013;
    mese = 2;

    switch (mese)
      {
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:
        case 12:
            giorni = 31;
            break;
        case 4:
        case 6:
        case 9:
        case 11:
            giorni = 30;
            break;
        case 2:
            if (((anno % 4 == 0) && !(anno % 100 == 0))
                || (anno % 400 == 0))
              {
```

```
                giorni = 29;
              }
            else
              {
                giorni = 28;
              }
            break;
      }
    printf ("Il mese %d dell'anno %d ha %d giorni.\n",
            mese, anno, giorni);
    getchar ();
    return 0;
}
```

È anche possibile dichiarare un caso predefinito che si verifichi quando nessuno degli altri si avvera.

Listato 81.64. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/8TIUpduT* , *http://ideone.com/3YhHc* .

```
#include <stdio.h>
int main (void)
{
    int mese;
    mese = 13;

    switch (mese)
      {
        case 1: printf ("gennaio\n"); break;
        case 2: printf ("febbraio\n"); break;
        case 3: printf ("marzo\n"); break;
        case 4: printf ("aprile\n"); break;
        case 5: printf ("maggio\n"); break;
        case 6: printf ("giugno\n"); break;
```

```
        case 7: printf ("luglio\n"); break;
        case 8: printf ("agosto\n"); break;
        case 9: printf ("settembre\n"); break;
        case 10: printf ("ottobre\n"); break;
        case 11: printf ("novembre\n"); break;
        case 12: printf ("dicembre\n"); break;
        default: printf ("mese non corretto\n"); break;
      }
    getchar ();
    return 0;
}
```

81.4.2.1 Esercizio

«

In un esempio già mostrato, appare la porzione di codice seguente. Si spieghi nel dettaglio come viene calcolata la quantità di giorni di febbraio:

```
        case 2:
            if (((anno % 4 == 0) && !(anno % 100 == 0))
                || (anno % 400 == 0))
              {
                giorni = 29;
              }
            else
              {
                giorni = 28;
              }
            break;
```

81.4.3 Iterazione con condizione di uscita iniziale: «while»

L'iterazione si ottiene normalmente in C attraverso l'istruzione '**while**', la quale esegue un'istruzione, o un gruppo di queste, finché la condizione continua a restituire il valore *Vero*. La condizione viene valutata prima di eseguire il gruppo di istruzioni e poi ogni volta che termina un ciclo, prima dell'esecuzione del successivo.

```
while (condizione) istruzione
```

L'esempio seguente fa apparire per 10 volte la lettera «x».

Listato 81.66. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/ZKrSA3IF* , *http://ideone.com/68bD684* .

```
#include <stdio.h>
int main (void)
{
    int i = 0;

    while (i < 10)
      {
        i++;
        printf ("x");
      }
    printf ("\n");
    getchar ();
    return 0;
}
```

Ma si osservi anche la variante seguente, con cui si ottiene un codice

più semplice in linguaggio macchina:

Listato 81.67. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/RVF64ri64O* , *http://ideone.com/EFVta* .

```
#include <stdio.h>
int main (void)
{
    int i = 10;

    while (i)
      {
        i--;
        printf ("x");
      }
    printf ("\n");
    getchar ();
    return 0;
}
```

Nel blocco di istruzioni di un ciclo '**while**', ne possono apparire alcune particolari, che rappresentano dei salti incondizionati nell'ambito del ciclo:

- '**break**', che serve a uscire definitivamente dalla struttura del ciclo;
- '**continue**', che serve a interrompere l'esecuzione del gruppo di istruzioni, riprendendo immediatamente con il ciclo successivo (a partire dalla valutazione della condizione).

L'esempio seguente è una variante del calcolo di visualizzazione mostrato sopra, modificato in modo da vedere il funzionamento del-

l'istruzione '**break**'. All'inizio della struttura, '**while (1)**' equivale a stabilire che il ciclo è senza fine, perché la condizione è sempre vera. In questo modo, solo la richiesta esplicita di interruzione dell'esecuzione della struttura (attraverso l'istruzione '**break**') permette l'uscita da questa.

Listato 81.68. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/Eyewc3QS* , *http://ideone.com/M0Mwz* .

```
#include <stdio.h>
int main (void)
{
    int i = 0;

    while (1)
      {
        if (i >= 10)
          {
            break;
          }
        i++;
        printf ("x");
      }
    printf ("\n");
    getchar ();
    return 0;
}
```

81.4.3.1 Esercizio

«

Sulla base delle conoscenze acquisite, si scriva un programma che calcola il fattoriale di un numero senza segno, contenuto nella va-

riabile x. Il fattoriale di x si ottiene con una serie di moltiplicazioni successive: $x \cdot (x-1) \cdot (x-2) \cdot \ldots \cdot 1$.

81.4.3.2 Esercizio

«

Sulla base delle conoscenze acquisite, si scriva un programma che verifica se un numero senza segno, contenuto nella variabile x, è un numero primo.

81.4.4 Iterazione con condizione di uscita finale: «do-while»

«

Una variante del ciclo '**while**', in cui l'analisi della condizione di uscita avviene dopo l'esecuzione del blocco di istruzioni che viene iterato, è definito dall'istruzione '**do**'.

```
do blocco_di_istruzioni while (condizione);
```

In questo caso, si esegue un gruppo di istruzioni una volta, poi se ne ripete l'esecuzione finché la condizione restituisce il valore *Vero*.

```
int i = 0;

do
  {
    i++;
    printf ("x");
  }
while (i < 10);
printf ("\n");
```

L'esempio mostrato è quello già usato in precedenza per visualizzare una sequenza di dieci «x», con l'adattamento necessario a utilizzare questa struttura di controllo.

La struttura di controllo '**do**...**while**' è in disuso, perché, generalmente, al suo posto si preferisce gestire i cicli di questo tipo attraverso una struttura '**while**', pura e semplice.

81.4.4.1 Esercizio

Modificare il programma che verifica se un numero è primo, usando un ciclo '**do**...**while**'.

81.4.5 Ciclo enumerativo: «for»

In presenza di iterazioni in cui si deve incrementare o decrementare una variabile a ogni ciclo, si usa preferibilmente la struttura '**for**', che in C permetterebbe un utilizzo più ampio di quello comune:

```
for ( [espressione1] ; [espressione2] ; [espressione3] ) istruzione
```

La forma tipica di un'istruzione '**for**' è quella per cui la prima espressione corrisponde all'assegnamento iniziale di una variabile, la seconda a una condizione che deve verificarsi fino a che si vuole che sia eseguita l'istruzione (o il gruppo di istruzioni) e la terza all'incremento o decremento della variabile inizializzata con la prima espressione. In pratica, l'utilizzo normale del ciclo '**for**' potrebbe esprimersi nella sintassi seguente:

```
for (var = n; condizione; var++) istruzione
```

Il ciclo '**for**' potrebbe essere definito anche in maniera differente, più generale: la prima espressione viene eseguita una volta sola all'inizio del ciclo; la seconda viene valutata all'inizio di ogni ciclo e

il gruppo di istruzioni viene eseguito solo se il risultato è *Vero*; l'ultima viene eseguita alla fine dell'esecuzione del gruppo di istruzioni, prima che si ricominci con l'analisi della condizione.

L'esempio già visto, in cui viene visualizzata per 10 volte una «x», potrebbe tradursi nel modo seguente, attraverso l'uso di un ciclo '**for**'.

Listato 81.70. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/4Rw1BygV* , *http://ideone.com/wckol* .

```
#include <stdio.h>
int main (void)
{
    int i;

    for (i = 0; i < 10; i++)
      {
        printf ("x");
      }
    printf ("\n");
    getchar ();
    return 0;
}
```

Anche nelle istruzioni controllate da un ciclo '**for**' si possono collocare istruzioni '**break**' e '**continue**', con lo stesso significato visto per il ciclo '**while**' e '**do**...**while**'.

Sfruttando la possibilità di inserire più espressioni in una singola istruzione, si possono realizzare dei cicli '**for**' molto più complessi, anche se questo è sconsigliabile per evitare di scrivere codice troppo difficile da interpretare. In questo modo, l'esempio precedente

potrebbe essere ridotto a quello che segue, dove si usa un punto e virgola solitario per rappresentare un'istruzione nulla.

Listato 81.71. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/Tz85p3aO* , *http://ideone.com/Nqq5d* .

```
#include <stdio.h>
int main (void)
{
    int i;

    for (i = 0; i < 10; printf ("x"), i++)
      {
        ;
      }
    printf ("\n");
    getchar ();
    return 0;
}
```

Se si utilizzano istruzioni multiple, separate con la virgola, occorre tenere presente che **l'espressione che esprime la condizione deve rimanere singola** (se per la condizione si usasse un'espressione multipla, conterebbe solo la valutazione dell'ultima). Naturalmente, nel caso della condizione, si possono costruire condizioni complesse con l'ausilio degli operatori logici, ma rimane il fatto che l'operatore virgola ('`,`') non dovrebbe avere senso lì.

Nel modello sintattico iniziale si vede che le tre espressioni sono opzionali e rimane solo l'obbligo di mettere i punti e virgola relativi. L'esempio seguente mostra un ciclo senza fine che viene interrotto attraverso un'istruzione '**`break`**'.

Listato 81.72. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/oM2mmrei* , *http://ideone.com/JUF2V* .

```
#include <stdio.h>
int main (void)
{
    int i = 0;
    for (;;)
      {
        if (i >= 10)
          {
            break;
          }
        printf ("x");
        i++;
      }
    getchar ();
    return 0;
}
```

81.4.5.1 Esercizio

«

Modificare il programma che calcola il fattoriale di un numero, usando un ciclo '**for**'.

81.4.5.2 Esercizio

«

Modificare il programma che verifica se un numero è primo, usando un ciclo '**for**'.

81.5 Funzioni del linguaggio C

Il linguaggio C offre le funzioni come mezzo per realizzare la scomposizione del codice in subroutine. Prima di poter essere utilizzate attraverso una chiamata, le funzioni devono essere dichiarate, anche se non necessariamente descritte. In pratica, se si vuole indicare nel codice una chiamata a una funzione che viene descritta più avanti, occorre almeno dichiararne il prototipo.

Le funzioni del linguaggio C prevedono il passaggio di parametri solo **per valore**, con tutti i tipi di dati, esclusi gli array (che invece vanno passati per riferimento, attraverso il puntatore alla loro posizione iniziale in memoria).

Il linguaggio C, attraverso la libreria standard, offre un gran numero di funzioni comuni che vengono importate nel codice attraverso l'istruzione '**`#include`**' del precompilatore. In pratica, in questo modo si importa la parte di codice necessaria alla dichiarazione e descrizione di queste funzioni. Per esempio, come si è già visto, per poter utilizzare la funzione ***printf()*** si deve inserire la riga '**`#include <stdio.h>`**' nella parte iniziale del file sorgente.

81.5.1 Dichiarazione di un prototipo

Quando la descrizione di una funzione può essere fatta solo dopo l'apparizione di una sua chiamata, occorre dichiararne il prototipo all'inizio, secondo la sintassi seguente:

```
tipo nome ( [tipo [ nome ] [,...] ] );
```

Il tipo, posto all'inizio, rappresenta il tipo di valore che la funzione restituisce. Se la funzione non deve restituire alcunché, si utilizza il

tipo '**void**'. Se la funzione utilizza dei parametri, il tipo di questi deve essere elencato tra le parentesi tonde. L'istruzione con cui si dichiara il prototipo termina regolarmente con un punto e virgola.

Lo standard C stabilisce che una funzione che non richiede parametri deve utilizzare l'identificatore '**void**' in modo esplicito, all'interno delle parentesi.

Segue la descrizione di alcuni esempi.

- ```
  int fattoriale (int);
  ```
  In questo caso, viene dichiarato il prototipo della funzione '**fattoriale**', che richiede un parametro di tipo '**int**' e restituisce anche un valore di tipo '**int**'.

- ```
  int fattoriale (int n);
  ```
  Come nell'esempio precedente, dove in più, per comodità si aggiunge il nome del parametro che comunque viene ignorato dal compilatore.

- ```
  void elenca ();
  ```
  Si tratta della dichiarazione di una funzione che fa qualcosa senza bisogno di ricevere alcun parametro e senza restituire alcun valore (*void*).

- ```
  void elenca (void);
  ```
  Esattamente come nell'esempio precedente, solo che è indicato in modo esplicito il fatto che la funzione non riceve argomenti (il tipo '**void**' è stato messo all'interno delle parentesi), come prescrive lo standard.

81.5.1.1 Esercizio

Scrivere i prototipi delle funzioni descritte nello schema successivo:

Nome della funzione	Tipo di valore restituito	Parametri
`alfa`	non restituisce alcunché	***x*** di tipo intero senza segno ***y*** di tipo carattere ***z*** di tipo a virgola mobile normale
`beta`	intero normale senza segno	non ci sono parametri
`gamma`	numero a virgola mobile di tipo normale	***x*** di tipo intero con segno ***y*** di tipo carattere senza segno

81.5.2 Descrizione di una funzione

La descrizione della funzione, rispetto alla dichiarazione del prototipo, richiede l'indicazione dei nomi da usare per identificare i parametri (mentre nel prototipo questi sono facoltativi) e naturalmente l'aggiunta delle istruzioni da eseguire. Le parentesi graffe che appaiono nello schema sintattico fanno parte delle istruzioni necessarie.

```
tipo nome ( [tipo parametro [, ...] ] )
{
        istruzione ;
        ...
}
```

Per esempio, la funzione seguente esegue il prodotto tra i due parametri forniti e ne restituisce il risultato:

```
int prodotto (int x, int y)
{
    return (x * y);
}
```

I parametri indicati tra parentesi, rappresentano una dichiarazione di variabili locali[11] che contengono inizialmente i valori usati nella chiamata. Il valore restituito dalla funzione viene definito attraverso l'istruzione '**return**', come si può osservare dall'esempio. Naturalmente, nelle funzioni di tipo '**void**' l'istruzione '**return**' va usata senza specificare il valore da restituire, oppure si può fare a meno del tutto di tale istruzione.

Nei manuali tradizionale del linguaggio C si descrivono le funzioni nel modo visto nell'esempio precedente; al contrario, nella guida *GNU coding standards* si richiede di mettere il nome della funzione in corrispondenza della colonna uno, così:

```
int
prodotto (int x, int y)
{
    return (x * y);
}
```

Le variabili dichiarate all'interno di una funzione, oltre a quelle dichiarate implicitamente come mezzo di trasporto degli argomenti della chiamata, sono visibili solo al suo interno, mentre quelle dichiarate al di fuori di tutte le funzioni, sono variabili globali, accessibili potenzialmente da ogni parte del programma.[12] Se una variabile locale ha un nome coincidente con quello di una variabile globale, allora, all'interno della funzione, quella variabile globale non è accessibile.

Le regole da seguire, almeno in linea di principio, per scrivere pro-

grammi chiari e facilmente modificabili, prevedono che si debba fare in modo di rendere le funzioni indipendenti dalle variabili globali, fornendo loro tutte le informazioni necessarie attraverso i parametri. In questo modo diventa del tutto indifferente il fatto che una variabile locale vada a mascherare una variabile globale; inoltre, ciò permette di non dover tenere a mente il ruolo di queste variabili globali e (se non si usano le variabili «statiche») fa sì che si ottenga una funzione completamente «rientrante».

81.5.2.1 Esercizio

«

Completare i programmi successivi con la dichiarazione dei prototipi e con la descrizione delle funzioni necessarie.

Listato 81.80. Per svolgere l'esercizio attraverso un servizio *pastebin*: *http://codepad.org/04dX04L2kd* , *http://ideone.com/y7APt* .

```
#include <stdio.h>
//
// Mettere qui il prototipo della funzione «fattoriale».
//
// Mettere qui la descrizione della funzione «fattoriale».
//
int main (void)
{
    unsigned int x = 7;
    unsigned int f;
    f = fattoriale (x);
    printf ("Il fattoriale di %d è pari a %d.\n", x, f);
    getchar ();
    return 0;
}
```

Listato 81.81. Per svolgere l'esercizio attraverso un servizio *pastebin*: *http://codepad.org/g8Og2JQ1* , *http://ideone.com/aTWpX* .

```
#include <stdio.h>
//
// Mettere qui il prototipo della funzione «primo».
//
// Mettere qui la descrizione della funzione «primo».
//
int main (void)
{
    unsigned int x = 11;
    if (primo (x))
      {
        printf ("%d è un numero primo.\n", x);
      }
    else
      {
        printf ("%d non è un numero primo.\n", x);
      }
    getchar ();
    return 0;
}
```

Listato 81.82. Per svolgere l'esercizio attraverso un servizio *pastebin*: *http://codepad.org/Sof9C5lT* , *http://ideone.com/J5X1A* .

```
#include <stdio.h>
//
// Mettere qui il prototipo della funzione «interesse».
//
// Mettere qui la descrizione della funzione «interesse».
//
```

```
// L'interesse si ottiene come capitale * tasso * tempo.
//
int main (void)
{
    double    capitale = 10000; // Euro
    double       tasso = 0.03;  // pari al 3 %
    unsigned int tempo = 3      // anni
    double   interessi;
    interessi = interesse (capitale, tasso, tempo);
    printf ("Un capitale di %f Euro ", capitale);
    printf ("investito al tasso del %f%% ", tasso * 100);
    printf ("Per %d anni, dà interessi per %f Euro.\n",
            tempo, interessi);
    getchar ();
    return 0;
}
```

81.5.3 Vincoli nei nomi

Quando si definiscono variabili e funzioni nel proprio programma, occorre avere la prudenza di non utilizzare nomi che coincidano con quelli delle librerie che si vogliono usare e che non possano andare in conflitto con l'evoluzione del linguaggio. A questo proposito va osservata una regola molto semplice: non si possono usare nomi «esterni» che inizino con il trattino basso ('_'); in tutti gli altri casi, invece, non si possono usare i nomi che iniziano con un trattino basso e continuano con una lettera maiuscola o un altro trattino basso.

Il concetto di nome esterno viene descritto a proposito della compilazione di un programma che si sviluppa in più file-oggetto da collegare assieme (sezione 66.3). L'altro vincolo ser-

ve a impedire, per esempio, la creazione di nomi come '**_Bool**' o '**__STDC_IEC_559__**'. Rimane quindi la possibilità di usare nomi che inizino con un trattino basso, purché continuino con un carattere minuscolo e siano visibili solo nell'ambito del file sorgente che si compone.

81.5.4 I/O elementare

«

L'input e l'output elementare che si usa nella prima fase di apprendimento del linguaggio C si ottiene attraverso l'uso di due funzioni fondamentali: ***printf()*** e ***scanf()***. La prima si occupa di emettere una stringa dopo averla trasformata in base a dei codici di composizione determinati; la seconda si occupa di ricevere input (generalmente da tastiera) e di trasformarlo secondo codici di conversione simili alla prima. Infatti, il problema che si incontra inizialmente, quando si vogliono emettere informazioni attraverso lo standard output per visualizzarle sullo schermo, sta nella necessità di convertire in qualche modo tutti i dati che non siano già di tipo '**char**'. Dalla parte opposta, quando si inserisce un dato che non sia da intendere come un semplice carattere alfanumerico, serve una conversione adatta nel tipo di dati corretto.

Per utilizzare queste due funzioni, occorre includere il file di intestazione '`stdio.h`', come è già stato visto più volte negli esempi.

Le due funzioni, ***printf()*** e ***scanf()***, hanno in comune il fatto di disporre di una quantità variabile di parametri, dove solo il primo è stato precisato. Per questa ragione, la stringa che costituisce il primo argomento deve contenere tutte le informazioni necessarie a individuare quelli successivi; pertanto, si fa uso di ***specificatori di conver-***

sione che definiscono il tipo e l'ampiezza dei dati da trattare. A titolo di esempio, lo specificatore '**%i**' si riferisce a un valore intero di tipo '**int**', mentre '**%li**' si riferisce a un intero di tipo '**long int**'.

Vengono mostrati solo alcuni esempi, perché una descrizione più approfondita nell'uso delle funzioni ***printf()*** e ***scanf()*** appare in altre sezioni (67.3 e 69.17). Si comincia con l'uso di ***printf()***:

```
…
double capitale = 1000.00;
double tasso    = 0.5;
int    montante = (capitale * tasso) / 100;
…
printf ("%s: il capitale %f, ", "Ciao", capitale);
printf ("investito al tasso %f%% ", tasso);
printf ("ha prodotto un montante pari a %d.\n", montante);
…
```

Gli specificatori di conversione usati in questo esempio si possono considerare quelli più comuni: '**%s**' incorpora una stringa; '**%f**' traduce in testo un valore che originariamente è di tipo '**double**'; '**%d**' traduce in testo un valore '**int**'; inoltre, '**%%**' viene trasformato semplicemente in un carattere percentuale nel testo finale. Alla fine, l'esempio produce l'emissione del testo: «Ciao: il capitale 1000.00, investito al tasso 0.500000% ha prodotto un montante pari a 1005.»

La funzione ***scanf()*** è un po' più difficile da comprendere: la stringa che definisce il procedimento di interpretazione e conversione deve confrontarsi con i dati provenienti dallo standard input. L'uso più semplice di questa funzione prevede l'individuazione di un solo dato:

```
…
int importo;
…
printf ("Inserisci l'importo: ");
scanf ("%d", &importo);
…
```

Il pezzo di codice mostrato emette la frase seguente e resta in attesa dell'inserimento di un valore numerico intero, seguito da [*Invio*]:

```
Inserisci l'importo: _
```

Questo valore viene inserito nella variabile ***importo***. Si deve osservare il fatto che gli argomenti successivi alla stringa di conversione sono dei puntatori, per cui, avendo voluto inserire il dato nella variabile ***importo***, questa è stata indicata preceduta dall'operatore '**&**' in modo da fornire alla funzione l'indirizzo corrispondente (si veda la sezione 66.5 sulla gestione dei puntatori).

Con una stessa funzione ***scanf()*** è possibile inserire dati per diverse variabili, come si può osservare dall'esempio seguente, ma in questo caso, per ogni dato viene richiesta la separazione con spazi orizzontali o anche con la pressione di [*Invio*].

```
printf ("Inserisci il capitale e il tasso:");
scanf ("%d%f", &capitale, &tasso);
```

81.5.5 Restituzione di un valore

«

In un sistema Unix e in tutti i sistemi che si rifanno a quel modello, i programmi, di qualunque tipo siano, al termine della loro esecuzione, restituiscono un valore che può essere utilizzato da uno script di

shell per determinare se il programma ha fatto ciò che si voleva o se è intervenuto qualche tipo di evento che lo ha impedito.

Convenzionalmente si tratta di un valore numerico, con un intervallo di valori abbastanza ristretto, in cui zero rappresenta una conclusione normale, ovvero priva di eventi indesiderati, mentre qualsiasi altro valore rappresenta un'anomalia. A questo proposito si consideri quello «strano» atteggiamento degli script di shell, per cui zero equivale a *Vero*.

Lo standard del linguaggio C prescrive che la funzione ***main()*** debba restituire un tipo intero, contenente un valore compatibile con l'intervallo accettato dal sistema operativo: tale valore intero è ciò che dovrebbe lasciare di sé il programma, al termine del proprio funzionamento.

Se il programma deve terminare, per qualunque ragione, in una funzione diversa da ***main()***, non potendo usare l'istruzione '**`return`**' per questo scopo, si può richiamare la funzione ***exit()***:

```
exit (valore_restituito);
```

La funzione ***exit()*** provoca la conclusione del programma, dopo aver provveduto a scaricare i flussi di dati e a chiudere i file. Per questo motivo, non restituisce un valore all'interno del programma, al contrario, fa in modo che il programma restituisca il valore indicato come argomento.

Per poterla utilizzare occorre includere il file di intestazione '`stdlib.h`' che tra l'altro dichiara già due macro-variabili adatte a definire la conclusione corretta o errata del programma: ***EXIT_SUCCESS*** e ***EXIT_FAILURE***.[13] L'esempio seguente

mostra in che modo queste macro-variabili potrebbero essere usate:

```
#include <stdlib.h>
…
…
if (…)
  {
    exit (EXIT_SUCCESS);
  }
else
  {
    exit (EXIT_FAILURE);
  }
```

Naturalmente, se si può concludere il programma nella funzione ***main()***, si può fare lo stesso con l'istruzione '**`return`**':

```
#include <stdlib.h>
…
…
int main (…)
{
    …
    if (…)
      {
        return (EXIT_SUCCESS);
      }
    else
      {
        return (EXIT_FAILURE);
      }
    …
}
```

81.5.5.1 Esercizio

«

Modificare uno degli esercizi già fatti, dove si verifica se un numero è primo, allo scopo di far concludere il programma con '**EXIT_SUCCESS**' se il numero è primo effettivamente; in caso contrario il programma deve terminare con il valore corrispondente a '**EXIT_FAILURE**'.

In un sistema operativo in cui si possa utilizzare una shell POSIX, per verificare il valore restituito dal programma appena terminato è possibile usare il comando seguente:

```
$ echo $? [Invio]
```

Si ricorda che la conclusione con successo di un programma si traduce normalmente nel valore zero.

81.6 Riferimenti

«

- Brian W. Kernighan, Dennis M. Ritchie, *Il linguaggio C: principi di programmazione e manuale di riferimento*, Pearson, ISBN 88-7192-200-X, *http://cm.bell-labs.com/cm/cs/cbook/*
- Open Standards, *C - Approved standards*, *http://www.open-std.org/jtc1/sc22/wg14/www/standards*
- *ISO/IEC 9899:TC2*, *http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf*
- Richard Stallman e altri, *GNU coding standards*, *http://www.gnu.org/prep/standards/*
- Autori vari, *GCC manual*, *http://gcc.gnu.org/onlinedocs/gcc/* , *http://gcc.gnu.org/onlinedocs/gcc.pdf*

81.7 Soluzioni agli esercizi proposti

«

Esercizio	Soluzione
81.1.2.1	`// Ciao mondo!` `#include <stdio.h>` `// La funzione main() viene eseguita automaticamente` `// all'avvio.` `int main (void)` `{` `  // Si limita a emettere un messaggio.` `  printf ("Ciao mondo!\n");` `  // Attende la pressione di un tasto, quindi termina.` `  getchar ();` `  return 0;` `}`
81.1.2.2	`#include <stdio.h>` `int main (void)` `{` `  printf ("Il mio primo programma\n");` `  printf ("scritto in linguaggio C.\n");` `  getchar ();` `  return 0;` `}`
81.1.3.1	`$` **`cc -o programma prova.c`** [*Invio*] Ma se si dispone del compilatore GNU C, è meglio usare l'opzione '**`-Wall`**': `$` **`cc -Wall -o programma prova.c`** [*Invio*]
81.1.4.1	`printf ("Imponibile: %d, IVA: %d.\n", 1000, 200);`
81.2.1.1	Con un byte da 8 bit non dovrebbe esserci la possibilità di avere una variabile scalare da 12 bit, perché di norma il byte è esattamente un sottomultiplo del rango delle variabili scalari disponibili.
81.2.2.1	Una variabile scalare di tipo '**`unsigned char`**' (da 8 bit) può rappresentare valori da 0 a 255, pertanto non è possibile assegnare a una tale variabile valori fino a 99999.
81.2.2.2	Una variabile scalare di tipo '**`unsigned char`**' (da 8 bit) può rappresentare valori da 0 a 255.
81.2.2.3	Una variabile scalare di tipo '**`signed short int`**' (da 16 bit) può rappresentare valori da −32768 a +32767.
81.2.2.4	Dovendo rappresentare il valore 12,34, si devono usare variabili in virgola mobile. Possono andare bene tutti i tipi: '**`float`**', '**`double`**' e '**`long double`**'.

Esercizio	Soluzione
81.2.3.1	La costante letterale '**12**' corrisponde a 12_{10}; la costante '**012**' rappresenta il numero 12_8, ovvero 10_{10}; la costante '**0x12**' indica il numero 12_{16}, ovvero 18_{10}.
81.2.3.2	La costante letterale '**12**' è di tipo '**int**'; la costante '**12U**' è di tipo '**unsigned int**'; la costante '**12L**' è di tipo '**long int**'; la costante '**1.2**' è di tipo '**double**'; la costante '**1.2**' è di tipo '**long double**'.
81.2.5.1	L'espressione ''**3'-2**' corrisponde in pratica alla costante carattere ''**1**''; l'espressione ''**5'+4**' corrisponde in pratica alla costante carattere ''**9**''.
81.2.7.1	`int d;` `unsigned long int e = 2111;` `float f = 21.11;` `const float g = 21.11;`
81.3.1.1	L'espressione '**4+4**' dovrebbe dare un risultato di tipo '**int**'; l'espressione '**4/3**' dovrebbe dare un risultato di tipo '**int**'; l'espressione '**4.0/3**' dovrebbe essere di tipo '**double**'; l'espressione '**4L*3**' dovrebbe essere di tipo '**long int**'.
81.3.2.1	`int a = 3;` `int b;` `b = --a;` ***a*** contiene 2 e ***b*** contiene 2.
81.3.2.1	`int a = 3;` `int b = 2;` `b = a + b;` ***a*** contiene 3 e ***b*** contiene 5.
81.3.2.1	`int a = 7;` `int b = 2;` `b = a % b;` ***a*** contiene 7 e ***b*** contiene 1.
81.3.2.1	`int a = 7;` `int b;` `b = (a = a * 2);` ***a*** contiene 14 e ***b*** contiene 14.
81.3.2.1	`int a = 3;` `int b = 2;` `b += a;` ***a*** contiene 3 e ***b*** contiene 5.
81.3.2.1	`int a = 7;` `int b = 2;` `b %= a;` ***a*** contiene 7 e ***b*** contiene 2.

Esercizio	Soluzione
81.3.2.1	`int a = 7;` `int b;` `b = (a *= 2);` ***a*** contiene 14 e ***b*** contiene 14.
81.3.2.2	`#include <stdio.h>` `int main (void)` `{` `  int a = 3;` `  int b;` `  b = --a;` `  printf ("a contiene %d;\n", a);` `  printf ("b contiene %d.\n", b);` `  getchar ();` `  return 0;` `}`
81.3.2.2	`#include <stdio.h>` `int main (void)` `{` `  int a = 3;` `  int b = 2;` `  b = a + b;` `  printf ("a contiene %d;\n", a);` `  printf ("b contiene %d.\n", b);` `  getchar ();` `  return 0;` `}`
81.3.2.2	`#include <stdio.h>` `int main (void)` `{` `  int a = 7;` `  int b = 2;` `  b = a % b;` `  printf ("a contiene %d;\n", a);` `  printf ("b contiene %d.\n", b);` `  getchar ();` `  return 0;` `}`

Esercizio	Soluzione
81.3.2.2	`#include <stdio.h>` `int main (void)` `{` `  int a = 7;` `  int b;` `  b = (a = a * 2);` `  printf ("a contiene %d;\n", a);` `  printf ("b contiene %d.\n", b);` `  getchar ();` `  return 0;` `}`
81.3.2.2	`#include <stdio.h>` `int main (void)` `{` `  int a = 3;` `  int b = 2;` `  b += a;` `  printf ("a contiene %d;\n", a);` `  printf ("b contiene %d.\n", b);` `  getchar ();` `  return 0;` `}`
81.3.2.2	`#include <stdio.h>` `int main (void)` `{` `  int a = 7;` `  int b = 2;` `  b %= a;` `  printf ("a contiene %d;\n", a);` `  printf ("b contiene %d.\n", b);` `  getchar ();` `  return 0;` `}`

Esercizio	Soluzione
81.3.2.2	`#include <stdio.h>` `int main (void)` `{` `  int a = 7;` `  int b;` `  b = (a *= 2);` `  printf ("a contiene %d;\n", a);` `  printf ("b contiene %d.\n", b);` `  getchar ();` `  return 0;` `}`
81.3.3.1	`int a = 4 + 1;` `signed char b = 5;` `int c = a == b;` ***c*** contiene 1.
81.3.3.1	`int a = 4 + 1;` `signed char b = 5;` `int c = a != b;` ***c*** contiene 0.
81.3.3.1	`unsigned int a = 4 + 3;` `signed char b = -5;` `int c = a >= b;` ***c*** contiene 1.
81.3.3.1	`unsigned int a = 4 + 3;` `signed char b = -5;` `int c = a <= b;` ***c*** contiene 0.
81.3.3.2	`#include <stdio.h>` `int main (void)` `{` `  int a = 4 + 1;` `  signed char b = 5;` `  int c = a == b;` `  printf ("(%d == %d) produce %d\n", a, b, c);` `  getchar ();` `  return 0;` `}`

Esercizio	Soluzione
81.3.3.2	`#include <stdio.h>` `int main (void)` `{` `  int a = 4 + 1;` `  signed char b = 5;` `  int c = a != b;` `  printf ("(%d != %d) produce %d\n", a, b, c);` `  getchar ();` `  return 0;` `}`
81.3.3.2	`#include <stdio.h>` `int main (void)` `{` `  unsigned int a = 4 + 3;` `  signed char b = -5;` `  int c = a >= b;` `  printf ("(%d >= %d) produce %d\n", a, b, c);` `  getchar ();` `  return 0;` `}`
81.3.3.2	`#include <stdio.h>` `int main (void)` `{` `  unsigned int a = 4 + 3;` `  signed char b = -5;` `  int c = a <= b;` `  printf ("(%d <= %d) produce %d\n", a, b, c);` `  getchar ();` `  return 0;` `}`
81.3.4.1	`int a = 4;` `signed char b = (3 < 5);` `int c = a && b;` ***c*** contiene 1.
81.3.4.1	`int a = 4;` `signed char b = (3 < 5);` `int c = a || b;` ***c*** contiene 1.

Esercizio	Soluzione
81.3.4.1	`unsigned int a = 4 + 3;` `signed char b = -5;` `int c = (b > 0) || (a > b);` ***c*** contiene 1.
81.3.5.1	`int a = 5;` `int b = 12;` `int c = a & b;` ***c*** contiene 4.
81.3.5.1	`int a = 5;` `int b = 12;` `int c = a | b;` ***c*** contiene 13.
81.3.5.1	`int a = 5;` `int b = 12;` `int c = a ^ b;` ***c*** contiene 9.
81.3.5.1	`int a = 5;` `int c = a << 1;` ***c*** contiene 10.
81.3.5.1	`int a = 21;` `int c = a >> 1;` ***c*** contiene 10.
81.3.5.2	`#include <stdio.h>` `int main (void)` `{` `  int a = 5;` `  int b = 12;` `  int c = a & b;` `  printf ("c contiene %d\n", c);` `  getchar ();` `  return 0;` `}`

Esercizio	Soluzione
81.3.5.2	`#include <stdio.h>` `int main (void)` `{` `  int a = 5;` `  int b = 12;` `  int c = a | b;` `  printf ("c contiene %d\n", c);` `  getchar ();` `  return 0;` `}`
81.3.5.2	`#include <stdio.h>` `int main (void)` `{` `  int a = 5;` `  int b = 12;` `  int c = a ^ b;` `  printf ("c contiene %d\n", c);` `  getchar ();` `  return 0;` `}`
81.3.5.2	`#include <stdio.h>` `int main (void)` `{` `  int a = 5;` `  int c = a << 1;` `  printf ("c contiene %d\n", c);` `  getchar ();` `  return 0;` `}`
81.3.5.2	`#include <stdio.h>` `int main (void)` `{` `  int a = 21;` `  int c = a >> 1;` `  printf ("c contiene %d\n", c);` `  getchar ();` `  return 0;` `}`

Esercizio	Soluzione
81.3.6.1	L'espressione '`(int) (4.4+4.9)`' è equivalente a '`(int) 9`'; l'espressione '`(double) 4/3`' è equivalente a '`(double) 1`'; l'espressione '`((double) 4)/3`' è equivalente a '`((double) 1.33333`'.
81.3.7.1	`int a = -20;` `int b = 10;` `int c = (b = 2, b = a, c = a + b);` ***c*** contiene −40.
81.3.7.1	`int a = -20;` `int b = 10;` `int c = (b = 2, b = a++, c = a + b);` ***c*** contiene −39.
81.3.7.1	`int a = -20;` `int b = 10;` `int c = (b = 2, b = ++a, c = a + b);` ***c*** contiene −38.
81.3.7.2	`#include <stdio.h>` `int main (void)` `{` `  int a = -20;` `  int b = 10;` `  int c = (b = 2, b = a, c = a + b);` `  printf ("c contiene %d\n", c);` `  getchar ();` `  return 0;` `}`
81.3.7.2	`#include <stdio.h>` `int main (void)` `{` `  int a = -20;` `  int b = 10;` `  int c = (b = 2, b = a++, c = a + b);` `  printf ("c contiene %d\n", c);` `  getchar ();` `  return 0;` `}`

Esercizio	Soluzione
81.3.7.2	`#include <stdio.h>` `int main (void)` `{` `  int a = -20;` `  int b = 10;` `  int c = (b = 2, b = ++a, c = a + b);` `  printf ("c contiene %d\n", c);` `  getchar ();` `  return 0;` `}`
81.4.1.1	`#include <stdio.h>` `int main (void)` `{` `  int x;` `  x = 5000;` `  if ((x < 1000) || (x > 10000))` `  {` `  printf ("Il valore di x non è valido!\n");` `  }` `  else if (x > 5000)` `  {` `  printf ("Il livello di x è alto: %d\n", x);` `  }` `  else if (x < 5000)` `  {` `  printf ("Il livello di x è basso: %d\n", x);` `  }` `  else` `  {` `  printf ("Il livello di x è ottimale: %d\n", x);` `  }` `  getchar ();` `  return 0;` `}`

Esercizio	Soluzione
81.4.1.2	```#include <stdio.h>
int main (void)
{
    int x;
    x = -1;

    if (x)
      {
        printf ("Sono felice :-)\n");
      }
    else
      {
        printf ("Sono triste :-(\n");
      }
    getchar ();
    return 0;
}```
Il programma visualizza la scritta «Sono felice :-)», perché un qualunque valore numerico diverso da zero viene inteso pari a *Vero*. |
| 81.4.2.1 | ```case 2:
    if (((anno % 4 == 0) && !(anno % 100 == 0))
        || (anno % 400 == 0))
      {
        giorni = 29;
      }
    else
      {
        giorni = 28;
      }
    break;```
Se l'anno è divisibile per quattro (pertanto la divisione per quattro non dà resto) e se l'anno non è divisibile per 100 (quindi non si tratta di un secolo), oppure se l'anno è divisibile per 400, il mese di febbraio ha 29, mentre ne ha 28 negli altri casi. In pratica, di norma gli anni bisestili sono quelli il cui anno è divisibile per quattro, ma questa regola non si applica se l'anno è l'inizio di un secolo, ma ogni quattro secoli si fa eccezione (pertanto, anche se di norma l'anno che inizia un secolo non è bisestile, il secolo che si ha ogni 400 anni è invece, nuovamente, bisestile). |

Esercizio	Soluzione
81.4.3.1	`#include <stdio.h>` `int main (void)` `{` `  unsigned int x = 4;` `  unsigned int f = x;` `  unsigned int i = (x - 1);` `  while (i > 0)` `  {` `  f = f * i;` `  i--;` `  }` `  printf ("%d! è pari a %d\n", x, f);` `  getchar ();` `  return 0;` `}`

Esercizio	Soluzione
81.4.3.2	

```
#include <stdio.h>
int main (void)
{
    int x = 11;
    int i = 2;
    if (x <= 1)
      {
        printf ("%d non è un numero primo.\n", x);
      }
    else
      {
        while (i < x)
          {
            if ((x % i) == 0)
              {
                printf ("%d è divisibile per %d.\n", x, i);
                break;
              }
            i++;
          }
        if (i >= x)
          {
            printf ("%d è un numero primo.\n", x);
          }
      }
    getchar ();
    return 0;
}
```

Esercizio	Soluzione
81.4.4.1	`#include <stdio.h>` `int main (void)` `{` `  int x = 11;` `  int i = 2;` `  if (x <= 1)` `  {` `  printf ("%d non è un numero primo.\n", x);` `  }` `  else` `  {` `  do` `  {` `  if ((x % i) == 0)` `  {` `  printf ("%d è divisibile per %d.\n", x, i);` `  break;` `  }` `  i++;` `  }` `  while (i < x);` `  if (i >= x)` `  {` `  printf ("%d è un numero primo.\n", x);` `  }` `  }` `  getchar ();` `  return 0;` `}`

Esercizio	Soluzione
81.4.5.1	`#include <stdio.h>` `int main (void)` `{` `  unsigned int x = 4;` `  unsigned int f = x;` `  unsigned int i;` `  for (i = (x - 1); i > 0; i--)` `  {` `  f = f * i;` `  }` `  printf ("%d! è pari a %d\n", x, f);` `  getchar ();` `  return 0;` `}`
81.4.5.2	`#include <stdio.h>` `int main (void)` `{` `  int x = 11;` `  int i;` `  if (x <= 1)` `  {` `  printf ("%d non è un numero primo.\n", x);` `  }` `  else` `  {` `  for (i = 2; i < x; i++)` `  {` `  if ((x % i) == 0)` `  {` `  printf ("%d è divisibile per %d.\n", x, i);` `  break;` `  }` `  }` `  if (i >= x)` `  {` `  printf ("%d è un numero primo.\n", x);` `  }` `  }` `  getchar ();` `  return 0;` `}`

Esercizio	Soluzione
81.5.1.1	`void alfa (unsigned int x, char y, double z);` `unsigned int beta (void);` `double gamma (int x, signed char y);`
81.5.2.1	`#include <stdio.h>` `unsigned int fattoriale (unsigned int x);` `unsigned int` `fattoriale (unsigned int x)` `{` `  unsigned int f = x;` `  unsigned int i;` `  for (i = (x - 1); i > 0; i--)` `  {` `  f = f * i;` `  }` `  return i;` `}` `int main (void)` `{` `  unsigned int x = 7;` `  unsigned int f;` `  f = fattoriale (x);` `  printf ("Il fattoriale di %d è pari a %d.\n", x, f);` `  getchar ();` `  return 0;` `}`

Esercizio	Soluzione
81.5.2.1	

```
#include <stdio.h>
unsigned int primo (unsigned int x);
unsigned int
primo (unsigned int x)
{
    unsigned int i;
    if (x <= 1)
      {
        return 0;
      }
    for (i = 2; i < x; i++)
      {
        if ((x % i) == 0)
          {
            return 0;
          }
      }
    if (i >= x)
      {
        return 1;
      }
    else
      {
        return 0;
      }
}
int main (void)
{
    unsigned int x = 11;
    if (primo (x))
      {
        printf ("%d è un numero primo.\n", x);
      }
    else
      {
        printf ("%d non è un numero primo.\n", x);
      }
    getchar ();
    return 0;
}
```

Esercizio	Soluzione
81.5.2.1	``` #include <stdio.h> double interesse (double c, double i, unsigned int t); double interesse (double c, double i, unsigned int t) { return (c * i * t); } int main (void) { double capitale = 10000; // Euro double tasso = 0.03; // pari al 3 % unsigned int tempo = 3; // anni double interessi; interessi = interesse (capitale, tasso, tempo); printf ("Un capitale di %f Euro ", capitale); printf ("investito al tasso del %f%% ", tasso * 100); printf ("Per %d anni, dà interessi per %f Euro.\n", tempo, interessi); getchar (); return 0; } ```

Esercizio	Soluzione
81.5.5.1	`#include <stdio.h>` `#include <stdlib.h>` `unsigned int primo (unsigned int x);` `unsigned int` `primo (unsigned int x)` `{` `  unsigned int i;` `  if (x <= 1)` `  {` `  return 0;` `  }` `  for (i = 2; i < x; i++)` `  {` `  if ((x % i) == 0)` `  {` `  return 0;` `  }` `  }` `  if (i >= x)` `  {` `  return 1;` `  }` `  else` `  {` `  return 0;` `  }` `}` `int main (void)` `{` `  unsigned int x = 11;` `  if (primo (x))` `  {` `  return (EXIT_SUCCESS);` `  }` `  else` `  {` `  return (EXIT_FAILURE);` `  }` `}`

[1] È bene osservare che un'istruzione composta, ovvero un raggruppamento di istruzioni tra parentesi graffe, non è concluso dal punto e virgola finale.

[2] In particolare, i nomi che iniziano con due trattini bassi (‘__’), oppure con un trattino basso seguito da una lettera maiuscola (‘_*X*’) sono riservati.

[3] Il linguaggio C, puro e semplice, non comprende alcuna funzione, benché esistano comunque molte funzioni più o meno standardizzate, come nel caso di ***printf()***.

[4] Sono esistiti anche elaboratori in grado di indirizzare il singolo bit in memoria, come il Burroughs B1900, ma rimane il fatto che il linguaggio C si interessi di raggiungere un byte intero alla volta.

[5] Il qualificatore ‘**`signed`**’ si può usare solo con il tipo ‘**`char`**’, dal momento che il tipo ‘**`char`**’ puro e semplice può essere con o senza segno, in base alla realizzazione particolare del linguaggio che dipende dall’architettura dell’elaboratore e dalle convenzioni del sistema operativo.

[6] La distinzione tra valori con segno o senza segno, riguarda solo i numeri interi, perché quelli in virgola mobile sono sempre espressi con segno.

[7] Come si può osservare, la dimensione è restituita dall’operatore ‘**`sizeof`**’, il quale, nell’esempio, risulta essere preceduto dalla notazione ‘**`(int)`**’. Si tratta di un cast, perché il valore restituito dall’operatore è di tipo speciale, precisamente si tratta del tipo ‘**`size_t`**’. Il cast è solo precauzionale perché generalmente tutto funziona in modo regolare senza questa indicazione.

[8] Per la precisione, il linguaggio C stabilisce che il «byte» corrisponda all’unità di memorizzazione minima che, però, sia anche in grado di rappresentare tutti i caratteri di un insieme minimo. Pertanto, ciò che restituisce l’operatore ***sizeof()*** è, in realtà, una quantità di byte,

solo che non è detto si tratti di byte da 8 bit.

[9] Gli operandi di '`? :`' sono tre.

[10] Lo standard prevede il tipo di dati '`_Bool`' che va inteso come un valore numerico compreso tra zero e uno. Ciò significa che il tipo '`_Bool`' si presta particolarmente a rappresentare valori logici (binari), ma ciò sempre secondo la logica per la quale lo zero corrisponde a *Falso*, mentre qualunque altro valore corrisponde a *Vero*.

[11] Per la precisione, i parametri di una funzione corrispondono alla dichiarazione di variabili di tipo automatico.

[12] Questa descrizione è molto semplificata rispetto al problema del campo di azione delle variabili in C; in particolare, quelle che qui vengono chiamate «variabili globali», non hanno necessariamente un campo di azione esteso a tutto il programma, ma in condizioni normali sono limitate al file in cui sono dichiarate. La questione viene approfondita in modo più adatto a questo linguaggio nella sezione 66.3.

[13] In pratica, ***EXIT_SUCCESS*** equivale a zero, mentre ***EXIT_FAILURE*** equivale a uno.

Capitolo 82

Puntatori, array e stringhe in C

82.1 Espressioni a cui si assegnano dei valori 2477
82.1.1 Esercizio 2477
82.2 Puntatori 2478
82.3 Dichiarazione di una variabile puntatore 2478
82.3.1 Esercizio 2479
82.4 Dereferenziazione 2479
82.4.1 Esercizio 2481
82.5 «Little endian» e «big endian» 2482
82.5.1 Esercizio 2485
82.6 Chiamata di funzione con puntatori 2487
82.6.1 Esercizio 2489
82.6.2 Esercizio 2490
82.7 Array 2490
82.8 Array a una dimensione 2490
82.8.1 Esercizio 2493
82.8.2 Esercizio 2493
82.8.3 Esercizio 2494
82.9 Array multidimensionali 2495
82.9.1 Esercizio 2498
82.9.2 Esercizio 2499

82.9.3 Esercizio 2499
82.10 Natura dell'array 2500
82.10.1 Esercizio 2504
82.10.2 Esercizio 2505
82.11 Array e funzioni 2506
82.12 Aritmetica dei puntatori 2507
82.12.1 Esercizio 2510
82.13 Stringhe 2511
82.13.1 Esercizio 2515
82.13.2 Esercizio 2516
82.14 Puntatori a puntatori 2517
82.14.1 Esercizio 2520
82.15 Puntatori a più dimensioni 2522
82.15.1 Esercizio 2527
82.16 Parametri della funzione main() 2530
82.17 Puntatori a variabili distrutte 2533
82.18 Soluzioni agli esercizi proposti 2534

`*` 2478 `**` 2517 2522 `***` 2517 `argc` 2530 `argv` 2530 `main()` 2530 `&` 2478

Nel linguaggio C, per poter utilizzare gli array si gestiscono dei puntatori alle zone di memoria contenenti tali strutture.

82.1 Espressioni a cui si assegnano dei valori

Quando si utilizza un operatore di assegnamento, come '=' o altri operatori composti, ciò che si mette alla sinistra rappresenta la «variabile ricevente» del risultato dell'espressione che si trova alla destra dell'operatore (nel caso di operatori di assegnamento composti, l'espressione alla destra va considerata come quella che si ottiene scomponendo l'operatore). Ma il linguaggio C consente di rappresentare quella «variabile ricevente» attraverso un'espressione, come nel caso dei puntatori che vengono descritti in questo capitolo. Pertanto, per evitare confusione, la documentazione dello standard chiama l'espressione a sinistra dell'operatore di assegnamento un *lvalue* (*Left value* o *Location value*).

Il concetto di *lvalue* serve a chiarire che un'espressione può rappresentare una «variabile», ovvero una certa posizione in memoria, pur senza averle dato un nome.

82.1.1 Esercizio

Nelle espressioni seguenti, indicare quali sono i componenti che costituiscono un *lvalue*:

Espressione	*lvalue*
`x = 4, y = 3 * 2`	***x*** **e** ***y***
`y = 3 * x`	
`z += 3 * x`	
`j = i++ * 5`	

82.2 Puntatori

«

Una variabile, di qualunque tipo sia, rappresenta normalmente un valore posto da qualche parte nella memoria del sistema. Attraverso l'operatore di indirizzamento e-commerciale ('**&**'), è possibile ottenere il puntatore (riferito alla rappresentazione ideale di memoria del linguaggio C) a una variabile «normale». Tale valore può essere inserito in una variabile particolare, adatta a contenerlo: una ***variabile puntatore***.

Per esempio, se ***p*** è una variabile puntatore adatta a contenere l'indirizzo di un intero, l'esempio mostra in che modo assegnare a tale variabile il puntatore alla variabile ***i***:

```
int i = 10;
…
// L'indirizzo di «i» viene assegnato al puntatore «p».
p = &i;
```

82.3 Dichiarazione di una variabile puntatore

«

La dichiarazione di una variabile puntatore avviene in modo simile a quello delle variabili normali, con l'aggiunta di un asterisco prima del nome. L'esempio seguente dichiara la variabile ***p*** come puntatore a un tipo '**int**'.

```
int *p;
```

Sia chiaro che la variabile dichiarata in questo modo ha il nome ***p*** ed è di tipo '**int ***', ovvero puntatore al tipo intero normale. Pertanto, l'asterisco, benché lo si rappresenti attaccato al nome della variabile, qui fa parte della dichiarazione del tipo.

Normalmente, il puntatore è costituito da un numero che rappresenta un indirizzo di memoria. Il fatto di precisare il tipo di variabile a cui si riferisce il puntatore, consente di sapere per quanti byte si estende l'informazione in questione.

82.3.1 Esercizio

Nella tabella successiva sono riportate delle istruzioni, a fianco delle quali si fanno delle domande. Si risponda a tali domande.

Codice	Questione
`int a = 20;` `b = &a;`	Quale dovrebbe essere il tipo della variabile ***b***?
	Come si dichiara la variabile ***x***, in qualità di puntatore al tipo '**`long long int`**'?
`long int *z;`	Cosa può contenere la variabile ***z***?

82.4 Dereferenziazione

Così come esiste l'operatore di indirizzamento, costituito dalla e-commerciale ('**`&`**'), con il quale si ottiene il puntatore corrispondente a una variabile, è disponibile un operatore di «dereferenziazione», con cui è possibile raggiungere la zona di memoria a cui si riferisce un puntatore, come se si trattasse di una variabile comune. L'operatore di dereferenziazione è l'asterisco ('**`*`**').

Attenzione a non fare confusione con gli asterischi: una cosa è quello usato per dichiarare o per dereferenziare un puntatore e un'altra è l'operatore con cui invece si ottiene la moltiplicazione.

Nell'esempio seguente, l'area di memoria a cui si riferisce il puntatore ***p*** viene sovrascritta con il valore 123:

```
int *p;
…
*p = 123;
```

Nell'esempio seguente, l'area di memoria a cui si riferisce il puntatore ***p***, corrispondente in pratica alla variabile ***v***, viene sovrascritta con il valore 456:

```
int v;
int *p;
…
p = &v;
*p = 456;
```

Nell'esempio appena apparso, si osserva alla fine che è possibile fare riferimento alla stessa area di memoria, sia attraverso la variabile ***v***, sia attraverso il puntatore dereferenziato ****p***.

L'esempio seguente serve a chiarire un po' meglio il ruolo delle variabili puntatore:

```
int v = 10;
int *p;
int *p2;
…
p = &v;
…
p2 = p;
…
*p2 = 20;
```

Alla fine, la variabile ***v*** e i puntatori dereferenziati ****p*** e ****p2*** contengono tutti lo stesso valore; ovvero, i puntatori ***p*** e ***p2*** individuano entrambi l'area di memoria corrispondente alla variabile ***v***, la quale si trova a contenere il valore 20.

Si osservi che l'asterisco è un operatore che, evidentemente, ha la precedenza rispetto a quelli di assegnamento. Eventualmente si possono usare le parentesi per togliere ambiguità al codice:

```
(*p2) = 20;
```

82.4.1 Esercizio

Nella tabella successiva sono riportate delle istruzioni, a fianco delle quali si fanno delle domande. Si risponda a tali domande.

Codice	Questione
`long int *i;` `long int  j;` … `i = j;`	L'ultima istruzione è errata: quale potrebbe essere la soluzione giusta?

Codice	Questione
`int *i;` `int  j = 10;` … `i = &j;` `(*i)++;`	Cosa contiene alla fine la variabile *j*?
`long int *i;` `int  j;` … `*i = j;`	L'ultima istruzione contiene un problema: come lo si può correggere?

82.5 «Little endian» e «big endian»

Il tipo di dati a cui un puntatore si rivolge, fa parte integrante dell'informazione rappresentata dal puntatore stesso. Ciò è importante perché quando si dereferenzia un puntatore occorre sapere quanto è grande l'area di memoria a cui si deve accedere a partire dal puntatore. Per questa ragione, quando si assegna a una variabile puntatore un altro puntatore, questo deve essere compatibile, nel senso che deve riferirsi allo stesso tipo di dati, altrimenti si rischia di ottenere un risultato inatteso. A questo proposito, l'esempio seguente contiene probabilmente un errore:

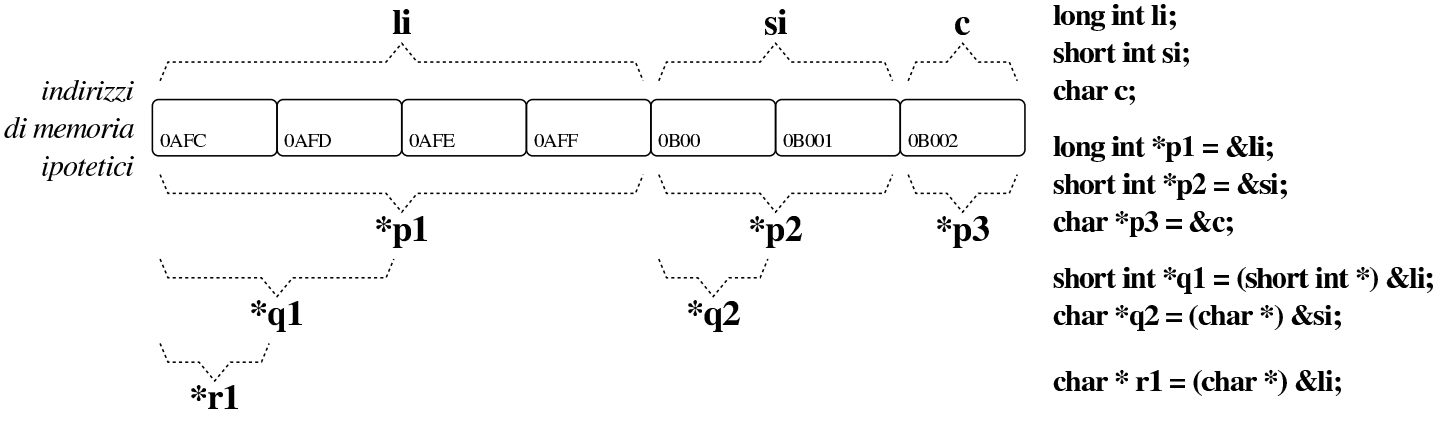
```
char *pc;
int  *pi;
…
pi = pc;   // I due puntatori si riferiscono a dati di tipo
           // differente!
…
```

Quando invece si vuole trasformare realmente un puntatore in modo che si riferisca a un tipo di dati differente, si può usare un cast, come

si farebbe per convertire i valori numerici:

```
char *pc;
int  *pi;
…
pi = (int *) pc;  // Il programmatore dimostra di essere
                  // consapevole di ciò che sta facendo
                  // attraverso un cast!
…
…
```

Nello schema seguente appare un esempio che dovrebbe consentire di comprendere la differenza che c'è tra i puntatori, in base al tipo di dati a cui fanno riferimento. In particolare, ***p1***, ***q1*** e ***r1*** fanno tutti riferimento all'indirizzo ipotetico $0AFC_{16}$, ma l'area di memoria che considerano è diversa, pertanto ****p1***, ****q1*** e ****r1*** sono tra loro «variabili» differenti, anche se si sovrappongono parzialmente.



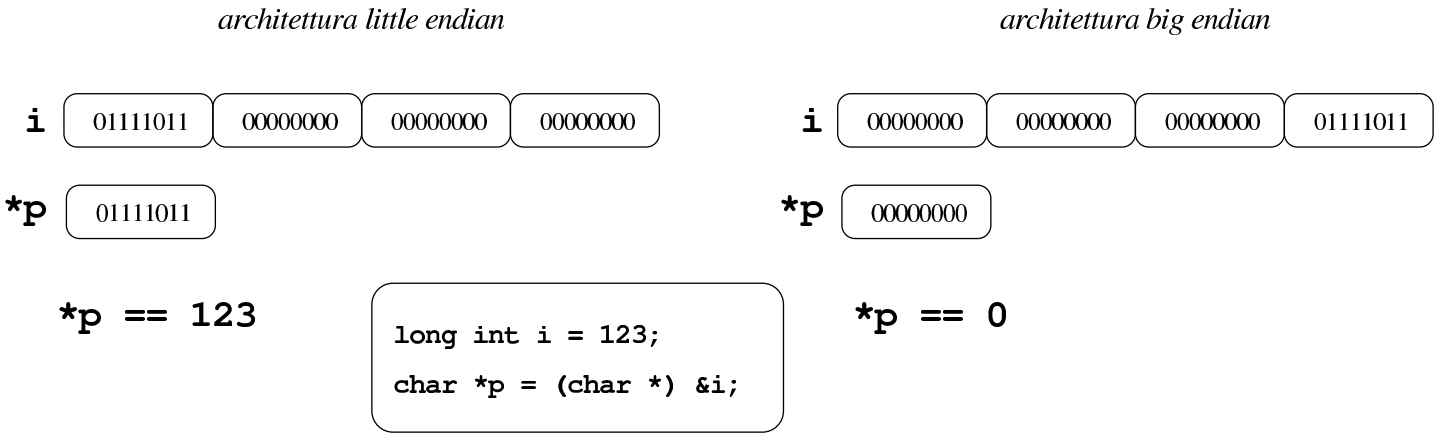
L'esempio seguente rappresenta un programma completo che ha lo scopo di determinare se l'architettura dell'elaboratore è di tipo *big endian* o di tipo *little endian*. Per capirlo si dichiara una variabile di tipo '**`long int`**' che si intende debba essere di rango superiore rispetto al tipo '**`char`**', assegnandole un valore abbastanza basso da poter essere rappresentato anche in un tipo '**`char`**' senza segno. Con un puntatore di tipo '**`char *`**' si vuole accedere all'inizio della varia-

bile contenente il numero intero '**long int**': se già nella porzione letta attraverso il puntatore al primo «carattere» si trova il valore assegnato alla variabile di tipo intero, vuol dire che i byte sono invertiti e si ha un'architettura *little endian*, mentre diversamente si presume che sia un'architettura *big endian*.

Listato 82.13. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/IRCiWUyg* , *http://ideone.com/aFfAG* .

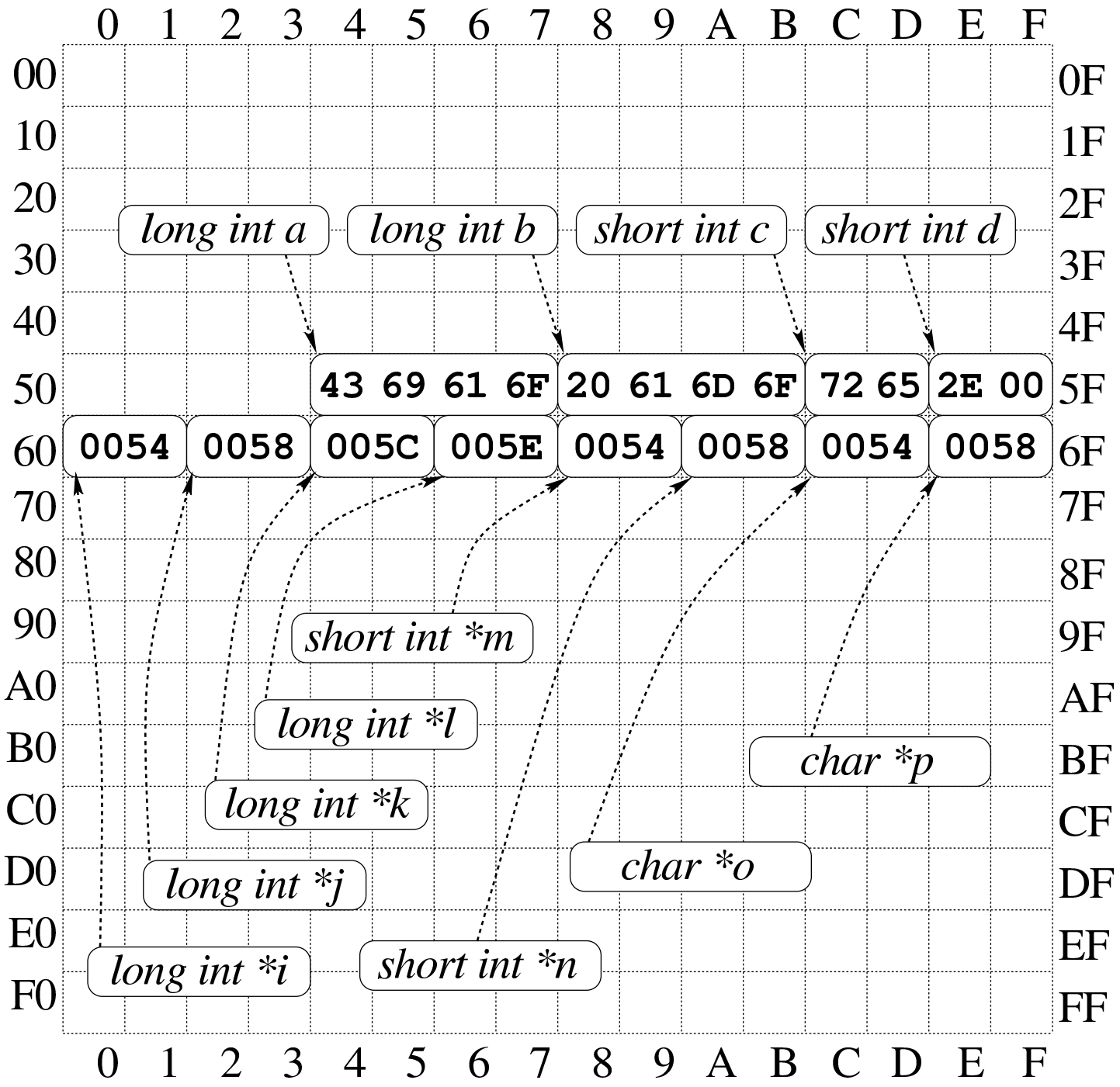
```
#include <stdio.h>
int main (void)
{
    long int i = 123;
    char *p = (char *) &i;
    if (*p == 123)
      {
        printf ("little endian\n");
      }
    else
      {
        printf ("big endian\n");
      }
    getchar ();
    return 0;
}
```

Figura 82.14. Schematizzazione dell'operato del programma di esempio, per determinare l'ordine dei byte usato nella propria architettura.



82.5.1 Esercizio

La figura successiva mostra una mappa ipotetica di memoria, con indirizzi che vanno da 00_{16} a FF_{16}, in cui sono evidenziate delle variabili scalari comuni e delle variabili puntatore. Ogni cella di memoria corrisponde a un byte e si presume che l'architettura del microprocessore preveda un accesso in modalità *big endian* (quello più semplice dal punto di vista umano). Si vuole conoscere il contenuto delle variabili scalari normali e quello rappresentato dai puntatori dereferenziati, con l'aiuto di alcuni suggerimenti.



Variabile o puntatore dereferenziato	Contenuto
a	$4369616F_{16}$
b	

Variabile o puntatore dereferenziato	Contenuto
c	
d	
****i***	$4369616F_{16}$
****j***	
****k***	$72652E00_{16}$
****l***	
****m***	
****n***	
****o***	
****p***	

82.6 Chiamata di funzione con puntatori

Il linguaggio C utilizza il passaggio degli argomenti alle funzioni per valore, per cui, anche se gli argomenti sono indicati in qualità di variabili, le modifiche ai valori rispettivi apportati nel codice delle

funzioni non si riflettono sul contenuto delle variabili originali; per farlo, occorre usare invece argomenti costituiti da puntatori.

Si immagini di volere realizzare una funzione banale che modifica la variabile utilizzata nella chiamata, sommandovi una quantità fissa. Invece di passare il valore della variabile da modificare, si può passare il suo puntatore; in questo modo la funzione (che comunque deve essere stata realizzata appositamente per questo scopo) agisce nell'area di memoria a cui punta il proprio parametro.

Listato 82.17. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/eEWeJvo2* , *http://ideone.com/bKTQx* .

```
#include <stdio.h>
void funzione (int *x)
{
    (*x)++;
}
int main (void)
{
    int y = 10;
    funzione (&y);
    printf ("y = %i\n", y);
    getchar ();
    return 0;
}
```

L'esempio mostra la dichiarazione e descrizione di una funzione che non restituisce alcun valore e ha un parametro costituito da un puntatore a un intero. Il lavoro della funzione è solo quello di incrementare il valore contenuto nell'area di memoria a cui si riferisce tale puntatore.

Poco dopo, nella funzione ***main()*** inizia il programma vero e proprio; viene dichiarata la variabile ***y*** corrispondente a un intero normale inizializzato a 10, poi viene chiamata la funzione vista prima, passando il puntatore a ***y***.

Il risultato è che dopo la chiamata, la variabile ***y*** contiene il valore precedente incrementato di un'unità, ovvero 11.

82.6.1 Esercizio

Si prenda in considerazione il programma successivo e si scriva il valore contenuto nelle tre variabili ***i***, ***j*** e ***k***, così come rappresentato dalla funzione ***printf()***.

```
#include <stdio.h>
int f (int *x, int y)
{
    return ((*x)++ + y);
}
int main (void)
{
    int i = 1;
    int j = 2;
    int k;
    k = f (&i, j);
    printf ("i=%i, j=%i, k=%i\n", i, j, k);
    getchar ();
    return 0;
}
```

82.6.2 Esercizio

«

Si modifichi il programma dell'esercizio precedente, creando nella funzione ***main()*** la variabile ***l***, in qualità di puntatore a un intero, assegnando a questa variabile il puntatore dell'area di memoria rappresentata da ***i***, usando poi la variabile ***l*** nella chiamata della funzione ***f()***.

82.7 Array

«

Nel linguaggio C, l'array è una sequenza ordinata di elementi dello stesso tipo nella rappresentazione ideale di memoria di cui si dispone. Quando si dichiara un array, quello che il programmatore ottiene in pratica è il riferimento alla posizione iniziale di questo, mentre gli elementi successivi si raggiungono tenendo conto della lunghezza di ogni elemento.

È compito del programmatore ricordare la quantità di elementi che compone l'array, perché determinarlo diversamente è complicato e a volte non è possibile. Inoltre, quando un programma tenta di accedere a una posizione oltre il limite degli elementi esistenti, c'è il rischio che non si manifesti alcun errore, arrivando però a dei risultati imprevedibili.

82.8 Array a una dimensione

«

La dichiarazione di un array avviene in modo intuitivo, definendo il tipo degli elementi e la loro quantità. L'esempio seguente mostra la dichiarazione dell'array ***a*** di sette elementi di tipo '**`int`**':

```
int a[7];
```

Per accedere agli elementi dell'array si utilizza un indice, il cui valore iniziale è sempre zero e, di conseguenza, quello con cui si raggiunge l'elemento ***n***-esimo deve avere il valore ***n***−1. L'esempio seguente mostra l'assegnamento del valore 123 al **secondo** elemento:

```
a[1] = 123;
```

In presenza di array monodimensionali che hanno una quantità ridotta di elementi, può essere sensato attribuire un insieme di valori iniziale all'atto della dichiarazione.

```
int a[] = {123, 453, 2, 67};
```

L'esempio mostrato dovrebbe chiarire in che modo si possono dichiarare gli elementi dell'array, tra parentesi graffe, togliendo così la necessità di specificare la quantità di elementi. Tuttavia, le due cose possono coesistere, purché siano compatibili:

```
int a[10] = {123, 453, 2, 67};
```

In tal caso, l'array si compone di 10 elementi, di cui i primi quattro con valori prestabiliti, mentre gli altri ottengono il valore zero. Si osservi però che il contrario non può essere fatto:

```
int a[5] = {123, 453, 2, 67, 32, 56, 78};      // Non si può!
```

La scansione di un array avviene generalmente attraverso un'iterazione enumerativa, in pratica con un ciclo '**`for`**' che si presta particolarmente per questo scopo. Si osservi l'esempio seguente:

```
int a[7];
int i;
…
for (i = 0; i < 7; i++)
  {
    …
    a[i] = …;
    …
  }
```

L'indice ***i*** viene inizializzato a zero, in modo da cominciare dal primo elemento dell'array; il ciclo può continuare fino a che ***i*** continua a essere inferiore a sette, infatti l'ultimo elemento dell'array ha indice sei; alla fine di ogni ciclo, prima che riprenda il successivo, viene incrementato l'indice di un'unità.

Per scandire un array in senso opposto, si può agire in modo analogo, come nell'esempio seguente:

```
int a[7];
int i;
…
for (i = 6; i >= 0; i--)
  {
    …
    a[i] = …;
    …
  }
```

Questa volta l'indice viene inizializzato in modo da puntare alla posizione finale; il ciclo viene ripetuto fino a che l'indice è maggiore o uguale a zero; alla fine di ogni ciclo, l'indice viene decrementato di un'unità.

82.8.1 Esercizio

«

Si completi la tabella successiva con il codice necessario a creare gli array richiesti.

Richiesta	Codice
Si vuole creare l'array ***a[]*** di 11 elementi di tipo intero senza segno.	
Si vuole creare l'array ***b[]*** di 3 elementi di tipo intero normale, contenente i valori 2, 7 e 123.	
Si vuole creare l'array ***c[]*** di 7 elementi di tipo intero normale, contenente inizialmente i valori 2, 7 e 123.	

82.8.2 Esercizio

«

Completare il codice successivo, in cui si dichiara un array e lo si popola successivamente con i primi valori numerici interi, a partire da uno.

```
…
int a[5];
int i;
…
for (i =          ; i          ; i          )
  {
    a[i] =               ;
  }
…
```

Dopo il ciclo '**for**', si vuole che l'array contenga la sequenza dei numeri: 1, 2, 3,... 5.

82.8.3 Esercizio

«

Si vuole ottenere lo stesso risultato dell'esercizio precedente, ma in questo caso viene posto un vincolo nel codice, in cui si vede che l'indice ***i*** viene decrementato nel ciclo '**for**'.

```
…
int a[5];
int i;
…
for (i =          ; i          ; i--)
  {
    a[i] =               ;
  }
…
```

Che valore ha la variabile ***i***, al termine del ciclo '**for**'?

82.9 Array multidimensionali

Gli array in C sono monodimensionali, però nulla vieta di creare un array i cui elementi siano array tutti uguali. Per esempio, nel modo seguente, si dichiara un array di cinque elementi che a loro volta sono insiemi di sette elementi di tipo '**int**'. Nello stesso modo si possono definire array con più di due dimensioni.

```
int a[5][7];
```

L'esempio seguente mostra il modo normale di scandire un array a due dimensioni:

```
int a[5][7];
int i;
int j;
…
for (i = 0; i < 5; i++)
  {
    …
    for  (j = 0; j < 7; j++)
      {
        …
        a[i][j] = …;
        …
      }
    …
  }
```

Anche se in pratica un array a più dimensioni è solo un array «normale» in cui si individuano dei sottogruppi di elementi, la scansione deve avvenire sempre indicando formalmente lo stesso numero di elementi prestabiliti per le dimensioni rispettive, anche se dovrebbe essere possibile attuare qualche trucco. Per esempio, tornando al

listato mostrato, se si vuole scandire in modo continuo l'array, ma usando un solo indice, bisogna farlo gestendo l'ultimo:

```
int a[5][7][9];
int j;
…
for (j = 0; j < (5 * 7 * 9); j++)
  {
    …
    a[0][0][j] = …;
    …
  }
```

Rimane comunque da osservare il fatto che questo non sia un bel modo di programmare.

Anche gli array a più dimensioni possono essere inizializzati, secondo una modalità analoga a quella usata per una sola dimensione, con la differenza che l'informazione sulla quantità di elementi per dimensione non può essere omessa. L'esempio seguente è un programma completo, in cui si dichiara e inizializza un array a due dimensioni, per poi mostrarne il contenuto.

Listato 82.32. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/d60HA60Fgn* , *http://ideone.com/4VFM9* .

```
#include <stdio.h>

int main (void)
{
  int a[3][4] = {{1,  2,  3,  4},
                 {5,  6,  7,  8},
                 {9, 10, 11, 12}};
  int i, j;
```

```
  for (i = 0; i < 3; i++)
    {
      for (j = 0; j < 4; j++)
        {
          printf ("a[%i][%i]=%i\t", i, j, a[i][j]);
        }
      printf ("\n");
    }

  getchar ();
  return 0;
}
```

Il programma dovrebbe mostrare il testo seguente:

```
a[0][0]=1       a[0][1]=2       a[0][2]=3       a[0][3]=4
a[1][0]=5       a[1][1]=6       a[1][2]=7       a[1][3]=8
a[2][0]=9       a[2][1]=10      a[2][2]=11      a[2][3]=12
```

Anche nell'inizializzazione di un array a più dimensioni si possono omettere degli elementi, come nell'estratto seguente:

```
…
  int a[3][4] = {{1, 2},
                 {5, 6, 7, 8}};
…
```

In tal caso, il programma si mostrerebbe così:

```
a[0][0]=1       a[0][1]=2       a[0][2]=0       a[0][3]=0
a[1][0]=5       a[1][1]=6       a[1][2]=7       a[1][3]=8
a[2][0]=0       a[2][1]=0       a[2][2]=0       a[2][3]=0
```

Di certo, pur sapendo di voler utilizzare un array a più dimensioni, si potrebbe pretendere di inizializzarlo come se fosse a una sola,

come nell'esempio seguente, ma il compilatore dovrebbe avvisare del fatto:

```
…
  int a[3][4] = {1, 2, 3, 4, 5, 6,              // Così non è
                 7, 8, 9, 10, 11, 12};          // grazioso.
…
```

82.9.1 Esercizio

Si completi la tabella successiva con il codice necessario a creare gli array richiesti.

Richiesta	Codice
Si vuole creare l'array ***a[]*** di 11×7 elementi di tipo intero senza segno.	
Si vuole creare l'array ***b[]*** di 3×2 elementi di tipo intero normale, contenente i valori {2, 7}, {5, 11} e {100, 123}.	
Si vuole creare l'array ***c[]*** di 7×2 elementi di tipo intero normale, contenente i valori {2, 7} e {5, 11}.	

82.9.2 Esercizio

Completare il codice successivo, in cui si dichiara un array e lo si popola successivamente con i primi valori numerici interi, a partire da uno.

```
…
int a[5][7];
int i;
int j;
…
for (i =          ; i          ; i           )
  {
    for (j =          ; j          ; j           )
      {
        a[i][j] =               ;
      }
  }
…
```

Dopo il ciclo '**for**', si vuole che l'array contenga la sequenza dei numeri: 1, 2, 3,... 35, cominciando dall'elemento ***a[0][0]***, per finire con l'elemento ***a[4][6]***.

82.9.3 Esercizio

Si vuole ottenere lo stesso risultato dell'esercizio precedente, ma in questo caso viene posto un vincolo nel codice, in cui si vede che gli indici ***i*** e ***j*** vengono decrementati nel ciclo '**for**' rispettivo.

```
…
int a[5][7];
int i;
int j;
…
for (i =          ; i          ; i--)
  {
    for (j =          ; j          ; j--)
      {
        a[i][j] =               ;
      }
  }
…
```

82.10 Natura dell'array

«

Quando si crea un array, quello che viene restituito in pratica è un puntatore alla sua posizione iniziale, ovvero all'indirizzo del primo elemento di questo. Si può intuire che non sia possibile assegnare a un array un altro array, anche se ciò potrebbe avere significato. Al massimo si può copiare il contenuto, elemento per elemento.

Per evitare errori del programmatore, la variabile che contiene l'indirizzo iniziale dell'array, quella che in pratica rappresenta l'array stesso, è in **sola lettura**. Quindi, nel caso dell'array già visto, la variabile ***a*** non può essere modificata, mentre i singoli elementi ***a[i]*** sì:

```
int a[7];
```

Data la filosofia del linguaggio C, se fosse possibile assegnare un valore alla variabile ***a***, si modificherebbe il puntatore, facendo in modo che questo punti a un array differente. Ma per raggiungere

questo risultato vanno usati i puntatori in modo esplicito. Si osservi l'esempio seguente.

Listato 82.41. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/MPcyb9yQ* , *http://ideone.com/j7lVY* .

```
#include <stdio.h>

int main (void)
{
  int a[3];
  int *p;

  p = a;          // «p» diventa un alias dell'array «a».

  p[0] = 10;      // Si può fare solo con gli array
  p[1] = 100;     // a una sola dimensione.
  p[2] = 1000;    //

  printf ("%i %i %i \n",  a[0], a[1], a[2]);

  getchar ();
  return 0;
}
```

Viene creato un array, ***a***, di tre elementi di tipo '**int**', e subito dopo una variabile puntatore, ***p***, al tipo '**int**'. Si assegna quindi alla variabile ***p*** il puntatore rappresentato da ***a***; da quel momento si può fare riferimento all'array indifferentemente con il nome ***a*** o ***p***.

Si può osservare anche che l'operatore '**&**', seguito dal nome di un array, produce ugualmente l'indirizzo dell'array che è equivalente a quello fornito senza l'operatore stesso, con la differenza che riguarda

l'array nel suo complesso:

```
…
p = &a; // I due puntatori non sono dello stesso tipo!
…
```

Pertanto, in questo caso si pone il problema di compatibilità del tipo di puntatore che si può risolvere con un cast esplicito:

```
…
p = (int *) &a; // «p» diventa un alias dell'array «a».
…
```

In modo analogo, si può estrapolare l'indice che rappresenta l'array dal primo elemento, cosa che si ottiene senza incorrere in problemi di compatibilità tra i puntatori. Si veda la trasformazione dell'esempio nel modo seguente.

Listato 82.44. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/LTyTlzk1* , *http://ideone.com/ndTqs* .

```
#include <stdio.h>

int main (void)
{
  int a[3];
  int *p;

  p = &a[0];     // «p» diventa un alias dell'array «a».

  p[0] = 10;     // Si può fare solo con gli array
  p[1] = 100;    // a una sola dimensione.
  p[2] = 1000;   //

  printf ("%i %i %i \n",  a[0], a[1], a[2]);
```

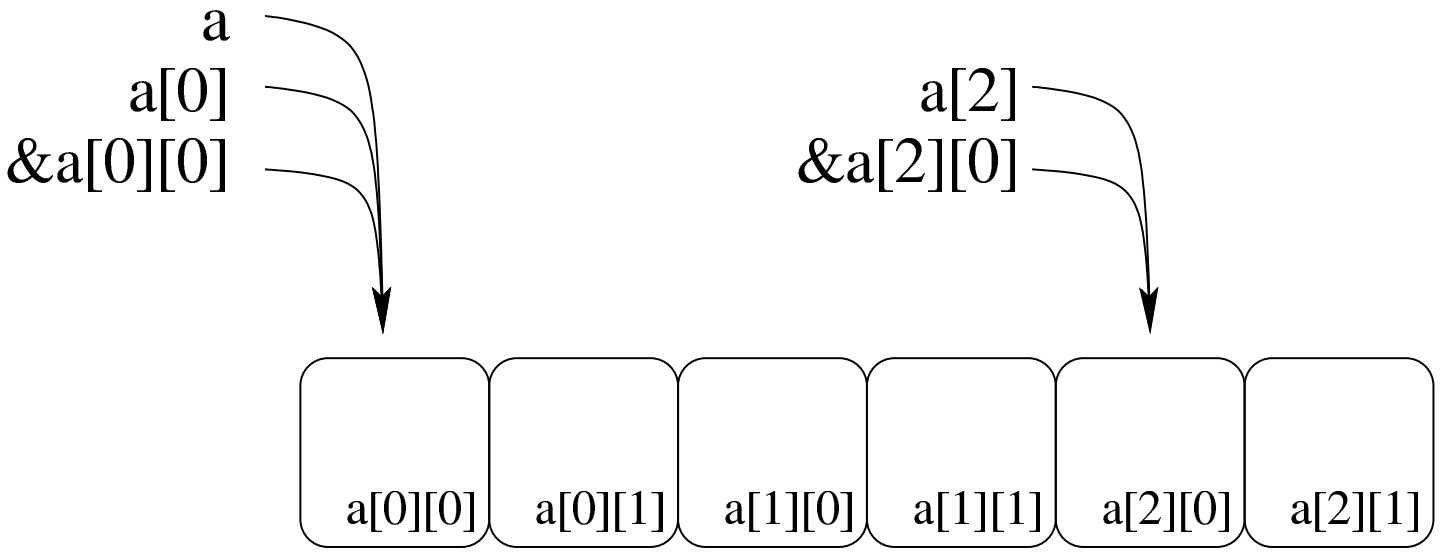
```
  getchar ();
  return 0;
}
```

Anche se si può usare un puntatore come se fosse un array, va osservato che la variabile ***p***, in quanto dichiarata come puntatore, viene considerata in modo differente dal compilatore.

Quando si opera con array a più dimensioni, il riferimento a una porzione di array restituisce l'indirizzo della porzione considerata. Per esempio, si supponga di avere dichiarato un array a due dimensioni, nel modo seguente:

```
int a[3][2];
```

Se a un certo punto, in riferimento allo stesso array, si scrivesse '**`a[2]`**', si otterrebbe l'indirizzo del terzo gruppo di due interi:



Tenendo d'occhio lo schema appena mostrato, considerato che si sta facendo riferimento all'array ***a*** di 3×2 elementi di tipo '**`int`**', va osservato che:

- in condizioni normali '**a**' si traduce nel puntatore a un array di due elementi di tipo '**int**';
- '**a[0]**' e '**&a[0][0]**' si traducono nel puntatore a un elemento di tipo '**int**' (precisamente il primo);
- '**&a**' si traduce nel puntatore a un array composto da 3×2 elementi di tipo '**int**'.

Pertanto, se questa volta si volesse assegnare a una variabile puntatore di tipo '**int ***' l'indirizzo iniziale dell'array, nell'esempio seguente si creerebbe un problema di compatibilità:

```
…
int a[3][2];
int *p;
p = a;    // I due puntatori non sono dello stesso tipo!
…
```

Pertanto, occorrerebbe riferirsi all'inizio dell'array in modo differente oppure attraverso un cast.

82.10.1 Esercizio

«

Il codice che appare nella tabella successiva, contiene dei problemi. Si spieghi perché.

Codice problematico	Spiegazione
`signed   int a[7];` `unsigned int b[8];` `…` `a = b;`	

Codice problematico	Spiegazione
`int a[7][5];` `long int *b;` … `b = a;`	
`int a[7][5];` `int *b;` … `b = &a[3];`	

82.10.2 Esercizio

Da un array viene estrapolato il puntatore, del suo inizio o di una posizione interna, e con quello si fanno delle modifiche al contenuto. Indicare dove avvengono le modifiche.

Codice	Richiesta
`int a[7][5];` `int *p;` … `p = (int *) a;` **`p[7] = 123;`**	L'ultima istruzione evidenziata, modifica un elemento dell'array ***a[][]***; quale?
`int a[7][5];` `int *p;` … `p = (int *) &a[1];` **`*p = 123;`**	L'ultima istruzione evidenziata, modifica un elemento dell'array ***a[][]***; quale?
`int a[7][5];` `int *p;` … `p = (int *) a;` **`p[35] = 123;`**	L'ultima istruzione evidenziata, modifica il contenuto dell'array ***a[][]***? Cosa fa invece?

82.11 Array e funzioni

«

Le funzioni possono accettare solo parametri composti da tipi di dati elementari, compresi i puntatori. In questa situazione, l'unico modo per trasmettere a una funzione un array attraverso i parametri, è quello di inviarne il puntatore iniziale. Di conseguenza, le modifiche che vengono poi apportate da parte della funzione si riflettono nell'array di origine. Si osservi l'esempio seguente.

Listato 82.50. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/GmqgyheC* , *http://ideone.com/59ix59q* .

```
#include <stdio.h>

void elabora (int *p)
{
  p[0] = 10;
  p[1] = 100;
  p[2] = 1000;
}

int main (void)
{
  int a[3];

  elabora (a);
  printf ("%i %i %i \n",  a[0], a[1], a[2]);

  getchar ();
  return 0;
}
```

La funzione ***elabora()*** utilizza un solo parametro, rappresentato da

un puntatore a un tipo '**int**'. La funzione **presume** che il puntatore si riferisca all'inizio di un array di interi e così assegna alcuni valori ai primi tre elementi.

All'interno della funzione ***main()*** viene dichiarato l'array ***a*** di tre elementi interi e subito dopo viene passato come argomento alla funzione ***elabora()***. Così facendo, in realtà si passa il puntatore al primo elemento dell'array.

Infine, la funzione altera gli elementi come è già stato descritto e gli effetti si possono osservare così:

```
10 100 1000
```

L'esempio potrebbe essere modificato per presentare la gestione dell'array in modo più elegante. Per la precisione si tratta di ritoccare la funzione '**elabora**':

```
void elabora (int a[])
{
    a[0] = 10;
    a[1] = 100;
    a[2] = 1000;
}
```

Si tratta sostanzialmente della stessa cosa, solo che si pone l'accento sul fatto che l'argomento è un array di interi, benché di tipo incompleto.

82.12 Aritmetica dei puntatori

Con le variabili puntatore è possibile eseguire delle operazioni elementari: possono essere incrementate e decrementate. Il risultato che si ottiene è il riferimento a una zona di memoria adiacente, in fun-

zione della dimensione del tipo di dati per il quale è stato creato il puntatore. Si osservi l'esempio seguente:

```
int i = 10;
int j;
int *p = &i;
p++;
j = *p;          // Attenzione!
```

In questo caso viene creato un puntatore al tipo '**int**' che inizialmente contiene l'indirizzo della variabile ***i***. Subito dopo questo puntatore viene incrementato di una unità e ciò comporta che si riferisca a un'area di memoria adiacente, immediatamente successiva a quella occupata dalla variabile ***i*** (molto probabilmente si tratta dell'area occupata dalla variabile ***j***). Quindi si tenta di copiare il valore di tale area di memoria, interpretato come '**int**', all'interno della variabile ***j***.

Se un programma del genere funziona nell'ambito di un sistema operativo che controlla l'utilizzo della memoria, se l'area che si tenta di raggiungere incrementando il puntatore non è stata allocata, si ottiene un «errore di segmentazione» e l'arresto del programma stesso. L'errore si verifica quando si tenta l'accesso, mentre la modifica del puntatore è sempre lecita.

Lo stesso meccanismo riguarda tutti i tipi di dati che non sono array, perché per gli array, l'incremento o il decremento di un puntatore riguarda i componenti dell'array stesso. In pratica, quando si gestiscono tramite puntatori, gli array sono da intendere come una serie di elementi dello stesso tipo e dimensione, dove, nella maggior parte dei casi, il nome dell'array si traduce nell'indirizzo del primo elemento:

```
int i[3] = { 1, 3, 5 };
int *p;
…
p = i;
```

Nell'esempio si vede che il puntatore ***p*** punta all'inizio dell'array di interi ***i[]***.

```
*p = 10; // Equivale a:  i[0] = 10.
p++;
*p = 30; // Equivale a:  i[1] = 30.
p++;
*p = 50; // Equivale a:  i[2] = 50.
```

Ecco che, incrementando il puntatore, si accede all'elemento adiacente successivo, in funzione della dimensione del tipo di dati. Decrementando il puntatore si ottiene l'effetto opposto, di accedere all'elemento precedente. La stessa cosa avrebbe potuto essere ottenuta così, senza alterare il valore contenuto nella variabile ***p***:

```
*(p + 0) = 10; // Equivale a:  i[0] = 10.
*(p + 1) = 30; // Equivale a:  i[1] = 30.
*(p + 2) = 50; // Equivale a:  i[2] = 50.
```

Inoltre, come già visto in altre sezioni, si potrebbe usare il puntatore con la stessa notazione propria dell'array, ma ciò solo perché si opera a una sola dimensione:

```
p[0] = 10; // Equivale a:  i[0] = 10.
p[1] = 30; // Equivale a:  i[1] = 30.
p[2] = 50; // Equivale a:  i[2] = 50.
```

82.12.1 Esercizio

«

Da un array viene estrapolato il puntatore, del suo inizio o di una posizione interna, e con quello si fanno delle cose. Rispondere alle domande a fianco del codice mostrato.

Codice	Richiesta
`int a[7][5];` `int *p;` … `p = (int *) a;` `p += 7;` `*p = 123;`	Attraverso la variabile puntatore ***p*** viene modificato un elemento dell'array ***a[][]***; quale?
`int a[7][5];` `int *p;` … `p = (int *) &a[1];` `*(p+7) = 123;`	Attraverso la variabile puntatore ***p*** viene modificato un elemento dell'array ***a[][]***; quale? Che differenza c'è rispetto al caso precedente?
`int a[7][5];` `int *p;` `int i;` … `p = (int *) a;` `for (i = 0 ; i < 35 ; i++, p++)` `  {` `  *p = i;` `  }`	Cosa succede al contenuto dell'array ***a[][]***? Al termine del ciclo '**`for`**', a cosa punta la variabile puntatore ***p***?

82.13 Stringhe

Le stringhe, nel linguaggio C, non sono un tipo di dati a sé stante; si tratta solo di array di caratteri con una particolarità: l'ultimo carattere è sempre zero, ovvero una sequenza di bit a zero, che si rappresenta simbolicamente come carattere con '`\0`'. In questo modo, si evita di dover accompagnare le stringhe con l'informazione della loro lunghezza.

Pertanto, va osservato che una stringa è sempre un array di caratteri, ma un array di caratteri non è necessariamente una stringa, in quanto per esserlo occorre che l'ultimo elemento sia il carattere '`\0`'. Seguono alcuni esempi che servono a comprendere questa distinzione.

```
char c[20];
```

L'esempio mostra la dichiarazione di un array di caratteri, senza specificare il suo contenuto. Per il momento non si può parlare di stringa, soprattutto perché per essere tale, la stringa deve contenere dei caratteri.

```
char c[] = {'c', 'i', 'a', 'o'};
```

Questo esempio mostra la dichiarazione di un array di quattro caratteri. All'interno delle parentesi quadre non è stata specificata la dimensione perché questa si determina dall'inizializzazione. Anche in questo caso non si può ancora parlare di stringa, perché manca la terminazione.

```
char z[] = {'c', 'i', 'a', 'o', '\0'};
```

Questo esempio mostra la dichiarazione di un array di cinque caratteri corrispondente a una stringa vera e propria. L'esempio seguente

è tecnicamente equivalente, solo che utilizza una rappresentazione più semplice:

```
char z[] = "ciao";
```

Pertanto, la stringa rappresentata dalla costante '**"ciao"**' è un array di cinque caratteri, perché, pur senza mostrarlo, include implicitamente anche la terminazione.

L'indicazione letterale di una stringa può avvenire attraverso sequenze separate, senza l'indicazione di alcun operatore di concatenamento. Per esempio, '**"ciao amore\n"**' è perfettamente uguale a '**"ciao " "amore" "\n"**' che viene inteso come una costante unica.

In un sorgente C ci sono varie occasioni di utilizzare delle stringhe letterali (delimitate attraverso gli apici doppi), senza la necessità di dichiarare l'array corrispondente. Però è importante tenere presente la natura delle stringhe per sapere come comportarsi con loro. Per prima cosa, bisogna rammentare che la stringa, anche se espressa in forma letterale, è un array di caratteri; come tale restituisce semplicemente il puntatore del primo di questi caratteri (salvo le stesse eccezioni che riguardano tutti i tipi di array).

```
char *p;
…
p = "ciao";
…
```

L'esempio mostra il senso di quanto affermato: non esistendo un tipo di dati «stringa», si può assegnare una stringa solo a un puntatore al tipo '**char**' (ovvero a una variabile di tipo '**char ***'). L'esem-

pio seguente non è valido, perché non si può assegnare un valore alla variabile che rappresenta un array, dal momento che il puntatore relativo è un valore costante:

```
char z[];
…
z = "ciao";      // Non si può.
…
```

Quando si utilizza una stringa tra gli argomenti della chiamata di una funzione, questa riceve il puntatore all'inizio della stringa. In pratica, si ripete la stessa situazione già vista per gli array in generale.

Listato 82.65. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/9Id0fIdf* , *http://ideone.com/CCkFd* .

```
#include <stdio.h>

void elabora (char *z)
{
  printf (z);
}

int main (void)
{
  elabora ("ciao\n");
  getchar ();
  return 0;
}
```

L'esempio mostra una funzione banale che si occupa semplicemente di emettere la stringa ricevuta come parametro, utilizzando ***printf()***. La variabile utilizzata per ricevere la stringa è stata dichiarata come

puntatore al tipo '**char**' (ovvero come puntatore di tipo '**char ***'), poi tale puntatore è stato utilizzato come argomento per la chiamata della funzione ***printf()***. Volendo scrivere il codice in modo più elegante si potrebbe dichiarare apertamente la variabile ricevente come array di caratteri di dimensione indefinita. Il risultato è lo stesso.

Listato 82.66. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/ksRqufBV* , *http://ideone.com/jmtac* .

```
#include <stdio.h>

void elabora (char z[])
{
  printf (z);
}

int main (void)
{
  elabora ("ciao\n");
  getchar ();
  return 0;
}
```

Tabella 82.67. Elenco dei modi di rappresentazione delle costanti carattere attraverso codici di escape.

Codice di escape	Descrizione
\ooo	Notazione ottale.
\xhh	Notazione esadecimale.

Codice di escape	Descrizione
`\\`	Una singola barra obliqua inversa (‘****’).
`\'`	Un apice singolo destro.
`\"`	Un apice doppio.
`\?`	Un punto interrogativo. Si usa in quanto le sequenze *trigraph* sono formate da un prefisso di due punti interrogativi.
`\0`	Il codice <*NUL*>.
`\a`	Il codice <*BEL*> (*bell*).
`\b`	Il codice <*BS*> (*backspace*).
`\f`	Il codice <*FF*> (*formfeed*).
`\n`	Il codice <*LF*> (*linefeed*).
`\r`	Il codice <*CR*> (*carriage return*).
`\t`	Una tabulazione orizzontale (<*HT*>).
`\v`	Una tabulazione verticale (<*VT*>).

82.13.1 Esercizio

«

Cosa contengono gli array rappresentati nella tabella successiva? Sono stringhe?

Codice
`int a[] = {'a', 'm', 'o', 'r', 'e', '\n', '\0'};`
`int b[] = {'a', 'm', 'o', 'r', 'e', '\0'};`
`int c[] = {'a', 'm', 'o', 'r', 'e', '\n'};`
`int d[] = {'a', 'm', 'o', 'r', 'e'};`
`char e[] = {'a', 'm', 'o', 'r', 'e', '\n', '\0'};`
`char f[] = {'a', 'm', 'o', 'r', 'e', '\0'};`
`char g[] = {'a', 'm', 'o', 'r', 'e', '\n'};`
`char h[] = {'a', 'm', 'o', 'r', 'e'};`
`char i[] = {'a', 'm', 'o', 'r', 'e', '\0', 'm', 'i', 'o'};`

82.13.2 Esercizio

«

Rispondere alle domande a fianco del codice contenuto nella tabella successiva.

Codice	Richiesta
… `char a[15];` …	Di cosa si tratta? Può essere una stringa?
… `char b[15] = "ciao";` …	Di cosa si tratta? Può essere una stringa?
… `char c[15] = "ciao";` … **`c = "amore";`**	È lecito l'assegnamento evidenziato? Perché?
… `char d[15] = "ciao";` `char *`**`e = d;`** …	È lecito l'assegnamento evidenziato? Perché?

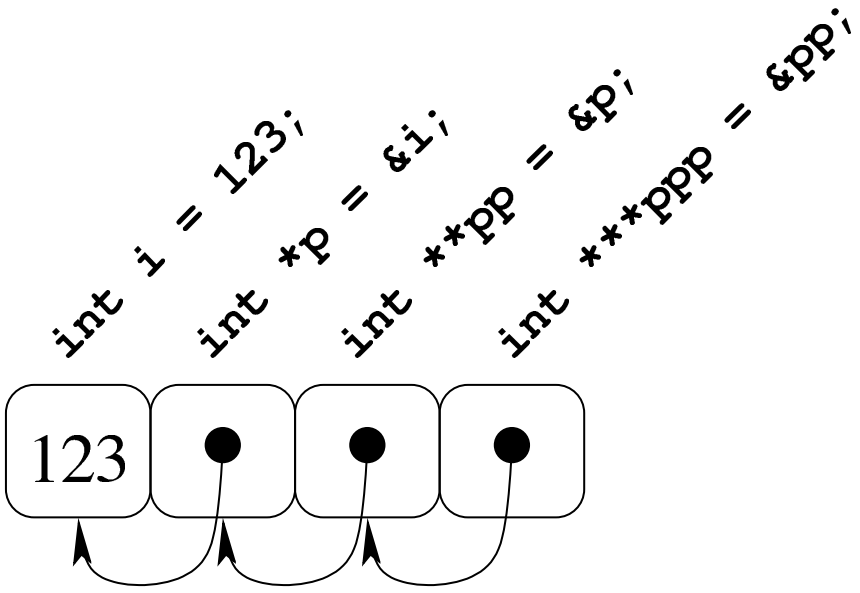
Codice	Richiesta
… `char *f;` … `f = "ciao";`	Dopo l'assegnamento, cos'è ***f***?
… `char *g = "ciao"` … `g = "amore";`	Dopo l'assegnamento, si può ancora fare riferimento alla stringa contenente la parola «ciao»? Che fine fa la memoria che la contiene?
… `char *h = "ciao"` … `*h = 'C';`	A cosa serve l'assegnamento finale? È possibile attuarlo?
… `char *i = "ciao"` … `i++;`	Al termine, cosa rappresenta ***i***?

82.14 Puntatori a puntatori

Una variabile puntatore potrebbe fare riferimento a un'area di memoria contenente a sua volta un puntatore per un'altra area. Per dichiarare una cosa del genere, si possono usare più asterischi, come nell'esempio seguente:

```
int i = 123;
int *p = &i;         // Puntatore al tipo "int".
int **pp = &p;       // Puntatore di puntatore al tipo "int".
int ***ppp = &pp;    // Puntatore di puntatore di puntatore
                     // al tipo "int".
```

Il risultato si potrebbe rappresentare graficamente come nello schema seguente:



Per dimostrare in pratica il funzionamento di questo meccanismo di riferimenti successivi, si può provare con il programma seguente.

Listato 82.72. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/6BXKTQeS* , *http://ideone.com/1FLV9* .

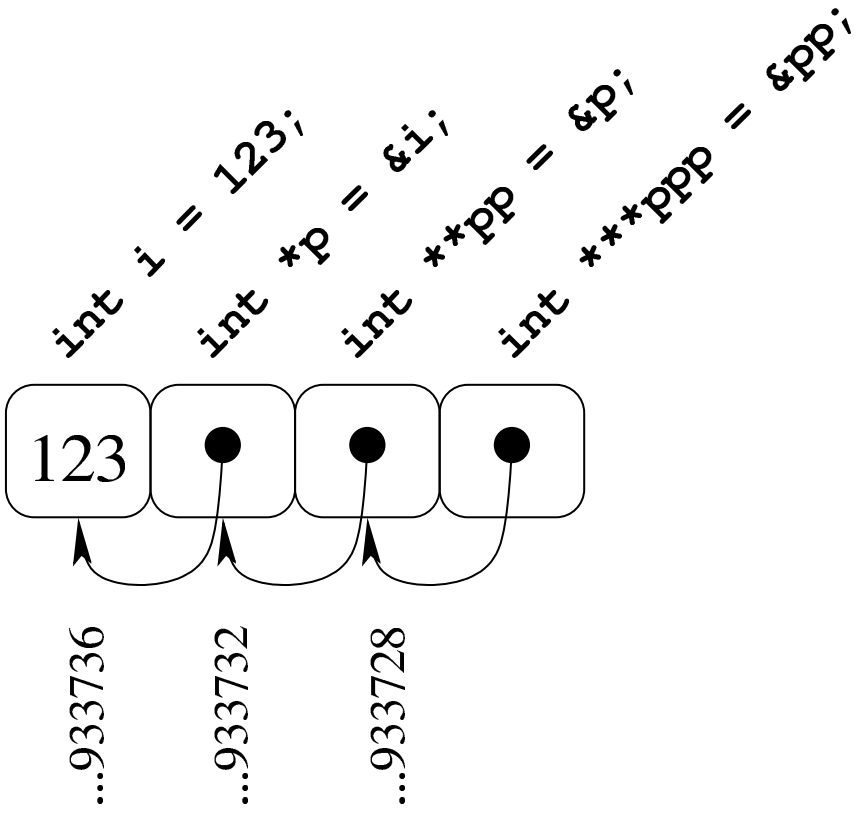
```
#include <stdio.h>
int main (void)
{
  int i = 123;
  int *p = &i;       // Puntatore al tipo "int".
  int **pp = &p;     // Puntatore di puntatore al tipo "int".
  int ***ppp = &pp; // Puntatore di puntatore di puntatore
                     // al tipo "int".

  printf ("i, p, pp, ppp: %i, %u, %u, %u\n",
           i, (unsigned int) p, (unsigned int) pp,
           (unsigned int) ppp);

  printf ("i, p, pp, *ppp: %i, %u, %u, %u\n",
           i, (unsigned int) p, (unsigned int) pp,
           (unsigned int) *ppp);
```

```
  printf ("i, p, *pp, **ppp: %i, %u, %u, %u\n",
           i, (unsigned int) p, (unsigned int) *pp,
           (unsigned int) **ppp);

  printf ("i, *p, **pp, ***ppp: %i, %i, %i, %i\n",
           i, *p, **pp, ***ppp);

  getchar ();
  return 0;
}
```

Eseguendo il programma si dovrebbe ottenere un risultato simile a quello seguente, dove si può verificare l'effetto delle dereferenziazioni applicate alle variabili puntatore:

```
i, p, pp, ppp: 123, 3217933736, 3217933732, 3217933728
i, p, pp, *ppp: 123, 3217933736, 3217933732, 3217933732
i, p, *pp, **ppp: 123, 3217933736, 3217933736, 3217933736
i, *p, **pp, ***ppp: 123, 123, 123, 123
```

Pertanto si può ricostruire la disposizione in memoria delle variabili:



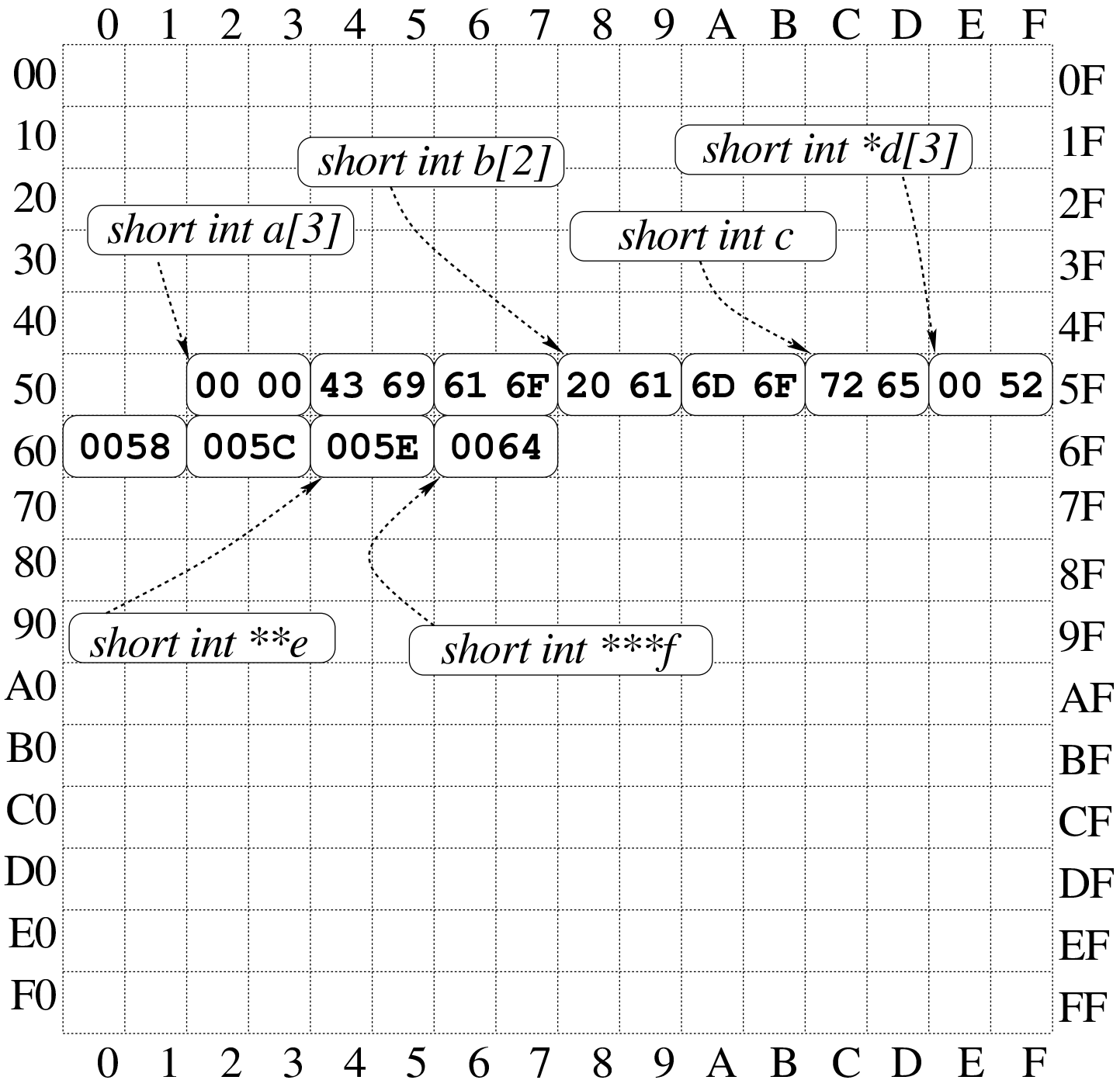

```
i == *p

i == **pp

i == ***ppp
```

Come si può comprendere facilmente, la gestione di puntatori a puntatore è difficile e va usata con prudenza e solo quando ne esiste effettivamente l'utilità. Va notato anche che si ottiene la dereferenziazione (la traduzione di un puntatore nel contenuto di ciò a cui punta) usando la notazione tipica degli array, ma questo fatto viene descritto nella sezione successiva.

82.14.1 Esercizio

«

La figura successiva mostra una mappa ipotetica di memoria, con indirizzi che vanno da 00_{16} a FF_{16}, in cui sono evidenziate delle variabili di vario tipo, inclusi i puntatori. Ogni cella di memoria corrisponde a un byte e si presume che l'architettura del microprocessore preveda un accesso in modalità *big endian* (quello più semplice dal punto di vista umano). Si vuole conoscere il risultato della dereferenziazione dei puntatori, secondo quanto richiesto espressamente nella tabella che segue la figura.



Variabile o puntatore dereferenziato	Contenuto
a[1]	4369_{16}
b[0]	

Variabile o puntatore dereferenziato	Contenuto
c	
d[0]	0052_{16}
****d[0]***	
****e***	0052_{16}
*****e***	
****f***	$005E_{16}$
*****f***	
******f***	

82.15 Puntatori a più dimensioni

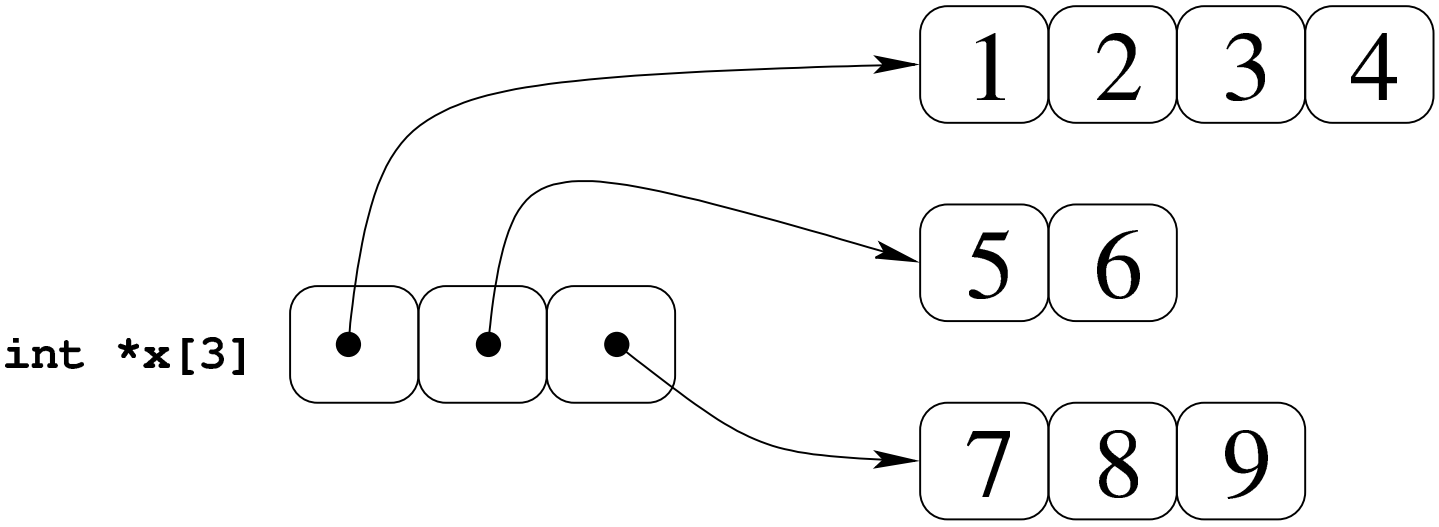
Un array di puntatori consente di realizzare delle strutture di dati ad albero, non più uniformi come invece devono essere gli array a più dimensioni consueti. L'esempio seguente mostra la dichiarazione di tre array di interi, con una quantità di elementi disomogenea, e la successiva dichiarazione di un array di puntatori di tipo '`int *`', a cui si assegnano i riferimenti ai tre array precedenti. Nell'esempio appare poi un tipo di notazione per accedere ai dati terminali che dovrebbe risultare intuitiva, ma se ne possono usare delle altre.

Listato 82.77. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/0hJZbbZ5* , *http://ideone.com/WelMI* .

```
#include <stdio.h>

int main (void)
{
  int a[] = {1, 2, 3, 4};
  int b[] = {5, 6,};
  int c[] = {7, 8, 9};
  int *x[] = {a, b, c};

  printf ("*x[0] = {%i, %i, %i, %i}\n",
          *x[0], *(x[0]+1), *(x[0]+2), *(x[0]+3));
  printf ("*x[1] = {%i, %i}\n", *x[1], *(x[1]+1));
  printf ("*x[2] = {%i, %i, %i}\n",
           *x[2], *(x[2]+1), *(x[2]+2));

  getchar ();
  return 0;
}
```

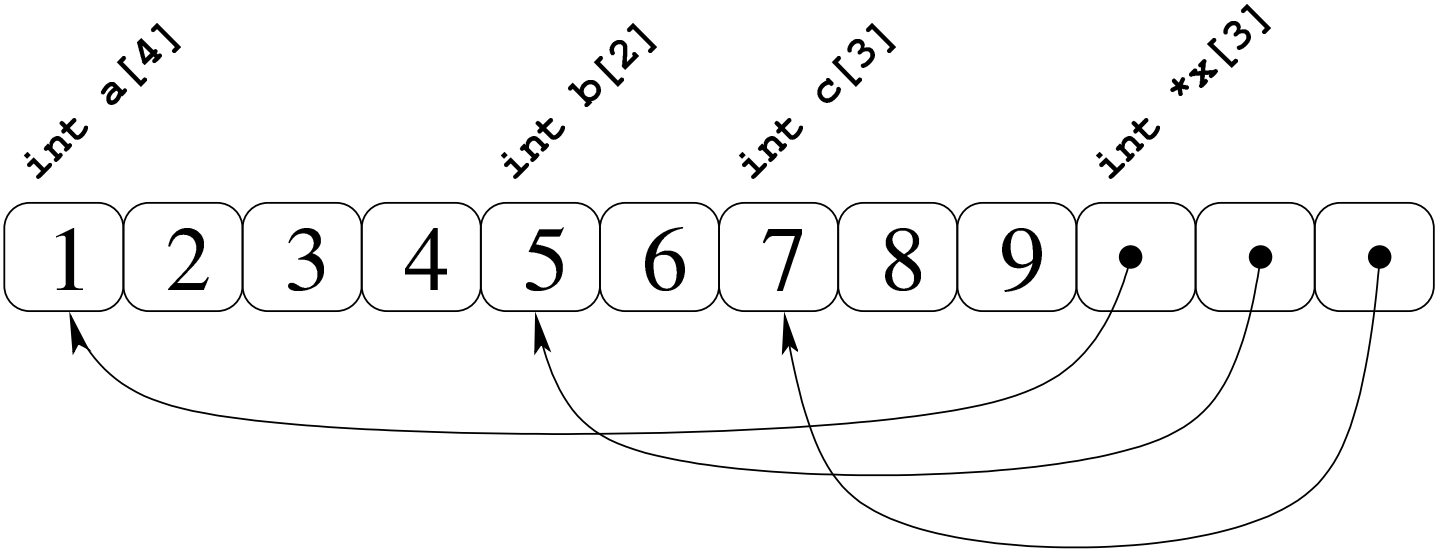
La figura successiva dovrebbe facilitare la comprensione del senso dell'array di puntatori. Come si può osservare, per accedere agli elementi degli array a cui puntano quelli di ***x*** è necessario dereferenziare gli elementi. Pertanto, '***x*`[0]`' corrisponde al contenuto del primo elemento del primo sotto-array, '`*(`***x***`[0]+1)`' corrisponde al contenuto del secondo elemento del primo sotto-array e così di seguito. Dal momento che i sotto-array non hanno una quantità uniforme di elementi, non è semplice la loro scansione.

Figura 82.78. Schematizzazione semplificata del significato dell'array di puntatori definito nell'esempio.



Si potrebbe obbiettare che la scansione di questo array di puntatori a array può avvenire ugualmente in modo sequenziale, come se fosse un array «normale» a una sola dimensione. Molto probabilmente ciò è possibile effettivamente, dal momento che è probabile che il compilatore disponga le variabili in memoria in sequenza, come si vede nella figura successiva, ma ciò non può essere garantito.

Figura 82.79. La disposizione più probabile delle variabili dell'esempio.



Se invece di un array di puntatori si ha un puntatore di puntatori, il meccanismo per l'accesso agli elementi terminali è lo stesso. L'esempio seguente contiene la dichiarazione di un puntatore a puntatori di tipo intero, a cui viene assegnato l'indirizzo dell'array già descritto. La scansione può avvenire nello stesso modo, ma ne viene proposto uno alternativo e più chiaro, con il quale si comprende cosa si intende per puntatore a più dimensioni.

Listato 82.80. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/D002Bp02rL* , *http://ideone.com/ozEKK* .

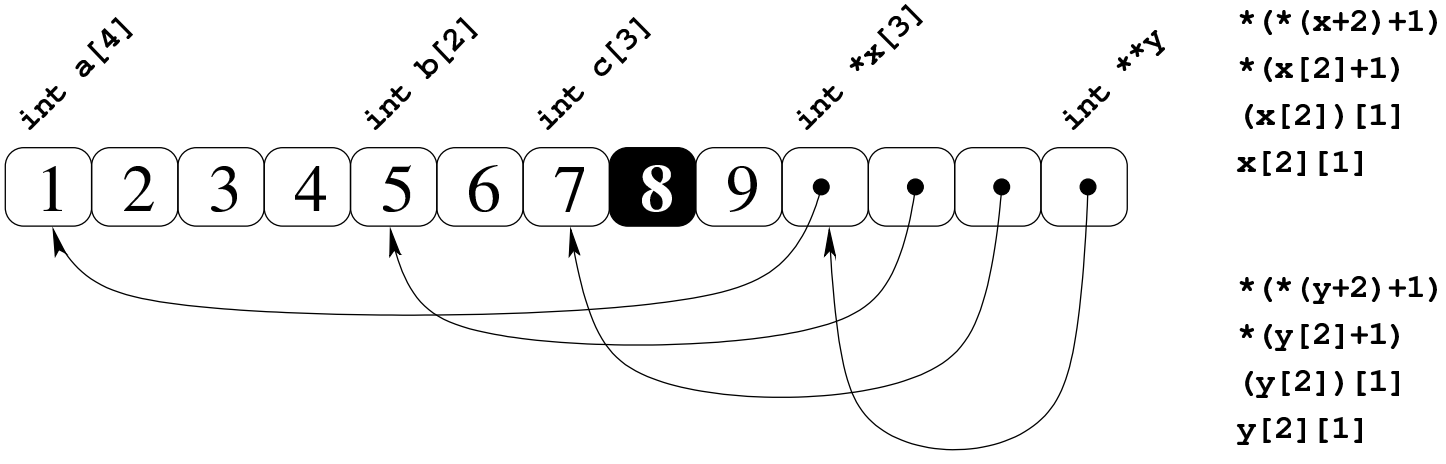
```
#include <stdio.h>

int main (void)
{
  int a[] = {1, 2, 3, 4};
  int b[] = {5, 6,};
  int c[] = {7, 8, 9};
  int *x[] = {a, b, c};
  int **y = x;

  printf ("*x[0] = {%i, %i, %i, %i}\n", y[0][0], y[0][1],
                                        y[0][2], y[0][3]);
  printf ("*x[1] = {%i, %i}\n", y[1][0], y[1][1]);
  printf ("*x[2] = {%i, %i, %i}\n", y[2][0], y[2][1],
                                    y[2][2]);

  getchar ();
  return 0;
}
```

Come si vede, la variabile ***y*** viene usata come se fosse un array a due

dimensioni, ma lo stesso sarebbe valso per la variabile ***x***, in qualità di array di puntatori.

Per capire cosa succede, occorre fare mente locale al fatto che il nome di una variabile puntatore seguito da un numero tra parentesi quadre corrisponde alla dereferenziazione dell'***n***-esimo elemento successivo alla posizione a cui punta tale variabile, mentre il valore puntato in sé corrisponde all'elemento zero (ciò è come dire che ***p** equivale a '`p[0]`'). Quindi, scrivere '`*(p+n)`' è esattamente uguale a scrivere '`p[n]`'. Se il valore a cui punta una variabile puntatore è a sua volta un puntatore, per dereferenziarlo occorrono due fasi: per esempio ***p* è il valore che si ottiene dereferenziando il primo puntatore e quello che si trova nella prima destinazione (quindi ***p* equivale a '`*p[0]`' e a '`p[0][0]`'). Volendo gestire gli indici si possono considerare equivalenti i puntatori: '`*(*(p+m)+n)`', '`*(p[m]+n)`', '`(p[m])[n]`' e '`p[m][n]`'.

Figura 82.81. Tanti modi alternativi per raggiungere lo stesso elemento.



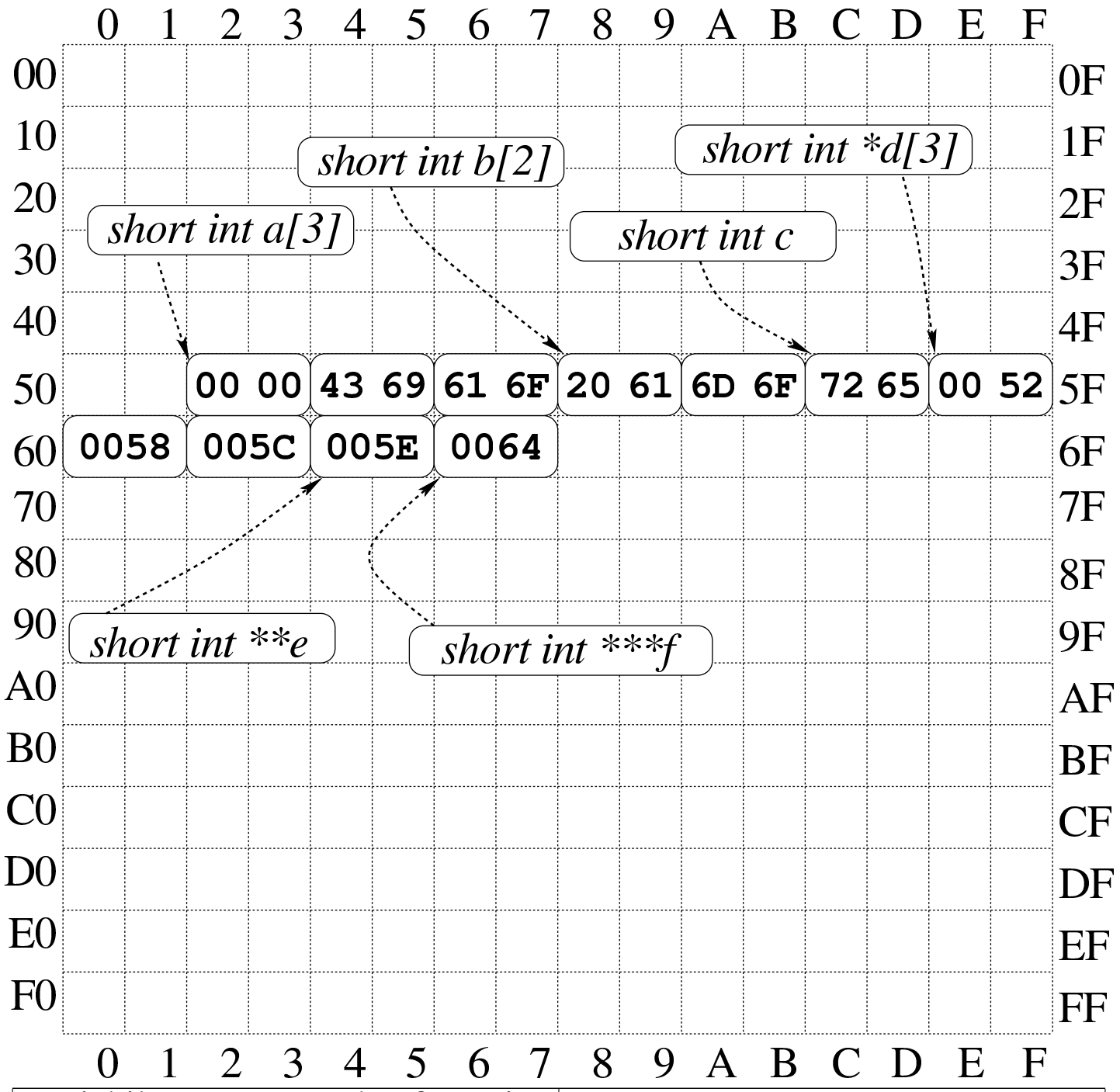
Seguendo lo stesso ragionamento si possono gestire strutture ad albero più complesse, con più livelli di puntatori, ma qui non vengono proposti esempi di questo tipo.

Sia l'array di puntatori, sia il puntatore a puntatori, possono essere gestiti con gli indici come se si trattasse di un array a più dimensioni. Pertanto, la notazione '***a*`[`*m*`][`*n*`]`' può rappresentare l'elemento ***m***,***n*** di un array ***a*** ottenuto secondo la rappresentazione «normale» a matrice, oppure secondo uno schema ad albero attraverso dei puntatori: la differenza sta solo nella presenza o meno di elementi costituiti da puntatori.

82.15.1 Esercizio

«

La figura successiva mostra una mappa ipotetica di memoria, con indirizzi che vanno da 00_{16} a FF_{16}, in cui sono evidenziate delle variabili di vario tipo, inclusi i puntatori. Ogni cella di memoria corrisponde a un byte e si presume che l'architettura del microprocessore preveda un accesso in modalità *big endian* (quello più semplice dal punto di vista umano). Si vuole conoscere il risultato della dereferenziazione dei puntatori, secondo quanto richiesto espressamente nella tabella che segue la figura.



Variabile o puntatore dereferenziato	Contenuto
a[1]	4369_{16}
b[0]	

Variabile o puntatore dereferenziato	Contenuto
c	
d[0]	0052_{16}
d[0][0]	
d[0][1]	4369_{16}
d[1][0]	
d[1][1]	
d[2][0]	
e[0]	
e[0][0]	
e[0][1]	4369_{16}
e[1][0]	
e[1][1]	
e[2][0]	

Variabile o puntatore dereferenziato	Contenuto
f[0]	
f[0][0]	
f[0][0][0]	
f[0][0][1]	4369_{16}
f[0][1][0]	
f[0][1][1]	
f[0][2][0]	

82.16 Parametri della funzione main()

«

La funzione ***main()***, se viene dichiarata con i suoi parametri tradizionali, permette di acquisire la riga di comando utilizzata per avviare il programma. La dichiarazione completa è la seguente:

```
int main (int argc, char *argv[])
{
  ...
}
```

Gli argomenti della riga di comando vengono convertiti in un array di stringhe (cioè di puntatori a '**char**'), in cui il primo elemento è il nome utilizzato per avviare il programma e gli elementi successivi

sono gli altri argomenti. Il primo parametro, ***argc***, serve a contenere la quantità di elementi del secondo, ***argv[]***, il quale è l'array di stringhe da scandire. È il caso di annotare che questo array dovrebbe avere sempre almeno un elemento: il nome utilizzato per avviare il programma e, di conseguenza, ***argc*** è sempre maggiore o uguale a uno.[1]

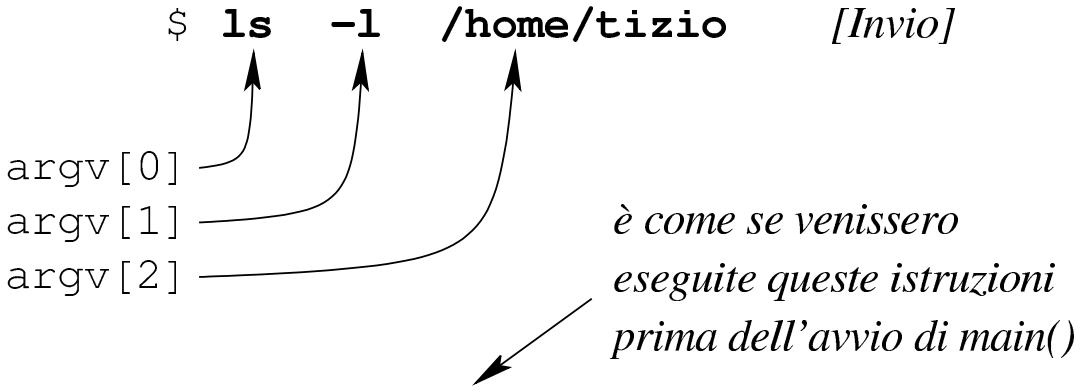
L'esempio seguente mostra in che modo gestire tale array, con la semplice riemissione degli argomenti attraverso lo standard output.

```
#include <stdio.h>

int main (int argc, char *argv[])
{
  int i;

  printf ("Il programma si chiama %s\n", argv[0]);

  for (i = 1; i < argc; i++)
    {
      printf ("argomento n. %i: %s\n", i, argv[i]);
    }
}
```

In alternativa, ma con lo stesso effetto, l'array di puntatori a stringhe può essere definito nel modo seguente, come puntatore di puntatori a caratteri:

```
int main (int argc, char **argv)
{
  ...
}
```

Figura 82.87. Schematizzazione di ciò che accade alla chiamata della funzione ***main()***, con un esempio.




```
int argc = 3;
char *argv[argc];
argv[0] = "/bin/ls";
argv[1] = "-l";
argv[2] = "/home/tizio";
main (argc, argv);
```

Chi è abituato a utilizzare linguaggi di programmazione più evoluti del C, può trovare strano che non si possa scrivere '**`main (int argc, char argv[][])`**' e usare di conseguenza l'array. Il motivo per cui ciò non è possibile dipende dal fatto che gli array a più dimensioni sono ottenuti attraverso sottoinsiemi uniformi del tipo dichiarato, così, in questo caso le stringhe dovrebbero essere della stessa dimensione, ma evidentemente ciò non corrisponde alla realtà. Inoltre, la dichiarazione della funzione dovrebbe contenere le dimensioni dell'array che non possono essere note. Pertanto, un array formato da stringhe diseguali, può essere ottenuto solo come array di puntatori al tipo '**`char`**'.

82.17 Puntatori a variabili distrutte

L'esempio seguente potrebbe funzionare, ma contiene un errore di principio.

Listato 82.88. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/vO5J8vzi* , *http://ideone.com/30i0s* .

```
#include <stdio.h>

double *f (void)
{
  double x = 1234.5678;
  return &x;                    // Orrore!
}

int main (int argc, char *argv[])
{
  double *p;
  p = f ();
  printf ("x = %f\n", *p);
  return 0;
}
```

La funzione *f()* dichiara localmente una variabile che inizializza al valore 1234,5678, quindi restituisce il puntatore a questa variabile. A parte il fatto che il compilatore possa segnalare o meno la cosa, non si può utilizzare un puntatore rivolto a un'area di memoria che, almeno teoricamente, non è più allocata. In altri termini, se si costruisce un puntatore a qualcosa, occorre tenere sempre presente il ciclo di vita della sua destinazione e non solo della variabile che contiene tale riferimento.

Purtroppo questa attenzione non viene imposta e, generalmente, il compilatore consente di usare un puntatore a variabili che, formalmente, sono già state distrutte.

82.18 Soluzioni agli esercizi proposti

«

Esercizio	Soluzione
82.1.1	L'espressione multipla '`x = 4, y = 3 * 2`' ha come *lvalue* le variabili ***x*** e ***y***. L'espressione '`y = 3 * x`' ha come *lvalue* la variabile ***y***. L'espressione '`z += 3 * x`' ha come *lvalue* la variabile *z*. L'espressione '`j=i++ * 5`' ha come *lvalue* le variabili ***j*** e ***i*** (la seconda viene incrementata di una unità dopo aver assegnato il prodotto di ***i*** per 5 alla variabile ***j***).
82.3.1	Per contenere il puntatore alla variabile ***a***, la quale è di tipo '`int`', la variabile ***b*** deve essere di tipo '`int *`'. La variabile ***x***, per essere un puntatore al tipo '`long long int`' si dichiara di tipo '`long long int *`'. La variabile *z*, essendo un puntatore al tipo '`long int`', può contenere il valore che esprime un indirizzo di memoria, all'interno del quale ci si attende di trovare un dato che si estende quanto richiederebbe un intero di tipo '`long int`'.
82.4.1	1) L'assegnamento corretto potrebbe essere '`i = &j`' oppure '`*i = j`', ma non si può sapere quale dei due fosse l'intenzione del programmatore. 2) La variabile ***j*** contiene alla fine il valore 11. 3) L'assegnamento richiederebbe un cast, perché il puntatore dereferenziato ****i*** è equivalente a una variabile di tipo '`long`', mentre ciò che gli viene assegnato è di tipo '`int`': '`*i = (long) j`'.

<table>
<tr><th>Esercizio</th><th>Soluzione</th></tr>
<tr><td>82.5.1</td><td>***a*** contiene $4369616F_{16}$.
b contiene $20616D6F_{16}$.
c contiene 7265_{16}.
d contiene $2E00_{16}$.
****i*** contiene $4369616F_{16}$.
****j*** contiene $20616D6F_{16}$.
****k*** contiene $72652E00_{16}$, perché ***k*** punta alla variabile ***c*** estendendosi fino a tutto il contenuto di ***d***.
****l*** contiene $2E000054_{16}$, perché ***l*** punta alla variabile ***d*** estendendosi fino a tutto il contenuto di ***j***.
****m*** contiene 4369_{16}, perché ***m*** punta alla variabile ***a*** estendendosi però solo fino alla sua metà.
****n*** contiene 2061_{16}, perché ***n*** punta alla variabile ***b*** estendendosi però solo fino alla sua metà.
****o*** contiene 43_{16}, perché ***o*** punta al primo byte della variabile ***a***.
****p*** contiene 20_{16}, perché ***p*** punta al primo byte della variabile ***b***.</td></tr>
<tr><td>82.6.1</td><td>***i***=2, ***j***=2, ***k***=3
Nella funzione ***f()***, il contenuto dell'area di memoria a cui punta ****x***, corrispondente a ***j***, viene incrementato di una unità dopo che si è svolta la somma; pertanto, il valore restituito dalla funzione è tre (uno+due).</td></tr>
<tr><td>82.6.2</td><td><pre>
#include <stdio.h>
int f (int *x, int y)
{
    return ((*x)++ + y);
}
int main (void)
{
    int i = 1;
    int j = 2;
    int k;
    int *l;
    l = &i;
    k = f (l, j);
    printf ("i=%i, j=%i, k=%i\n", i, j, k);
    getchar ();
    return 0;
}
</pre></td></tr>
</table>

Esercizio	Soluzione
82.8.1	`unsigned int a[11];` `int b[] = { 2, 7, 123 };` `int c[7] = { 2, 7, 123 };`
82.8.2	`int a[5];` `int i;` … `for (i = 0 ; i < 5 ; i++)` `  {` `  a[i] = i + 1;` `  }`
82.8.3	`int a[5];` `int i;` … `for (i = 4 ; i >= 0 ; i--)` `  {` `  a[i] = i + 1;` `  }` Al termine, la variabile ***i*** ha il valore -1.
82.9.1	`unsigned int a[11][7];` `int b[3][2] = {{2, 7}, {5, 11}, {100, 123}};` `int c[7][2] = {{2, 7}, {5, 11}};`
82.9.2	`int a[5][7];` `int i;` `int j;` … `for (i = 0 ; i < 5 ; i++)` `  {` `  for (j = 0 ; j < 6 ; j++)` `  {` `  a[i][j] = (i * 7) + j + 1;` `  }` `  }`

Esercizio	Soluzione
82.9.3	`int a[5][7];` `int i;` `int j;` `…` `for (i = 4 ; i >= 0 ; i--)` `  {` `  for (j = 7 ; j >= 0 ; j--)` `  {` `  a[i][j] = (i * 7) + j + 1;` `  }` `  }`
82.10.1	1) La variabile che rappresenta un array è in sola lettura, perciò non le si può assegnare alcunché. 2) La variabile puntatore ***b*** riguarda il tipo '**`long int`**', mentre l'array ***a*** si compone di elementi di tipo '**`int`**', pertanto i puntatori non possono essere dello stesso tipo; tuttavia, anche se non ci fosse questo problema, c'è da osservare che l'array ***a[][]*** è a due dimensioni, restituendo, in questo caso, il puntatore a un'area di memoria lunga cinque volte un intero normale, rendendo comunque incompatibile l'assegnamento alla variabile ***b***; pertanto, si richiede un cast. 3) Il puntatore che si ottiene da '**`&a[3]`**' si riferisce a un array di cinque elementi di tipo '**`int`**', pertanto è incompatibile con ***p*** e si richiederebbe eventualmente un cast, oppure si potrebbe togliere l'operatore '**`&`**', rendendo in questo caso compatibili i puntatori.
82.10.2	1) Viene modificato l'elemento ***a[1][1]***. 2) Viene modificato l'elemento ***a[1][0]***. 3) L'area di memoria a cui si riferisce ***p[35]*** è immediatamente successiva allo spazio occupato dall'array ***a[][]***; infatti, essendo questo composto da 35 elementi, ***p[35]*** si riferisce a un 36-esimo elemento non esistente.
82.12.1	1) Viene modificato l'elemento ***a[1][1]***. 2) Viene modificato l'elemento ***a[1][1]***. In questo caso, il puntatore corrispondente al contenuto della variabile ***p*** non viene modificato, continuando a riferirsi all'inizio dell'array ***a[][]***. 3) Le celle dell'array ***a[][]*** vengono inizializzate con un valore intero da uno a 34. Al termine, il puntatore contenuto nella variabile ***p*** si riferisce all'area di memoria immediatamente successiva all'array ***a[][]***.

Esercizio	Soluzione
82.13.1	***a[]*** è un array di interi, di sette elementi, contenenti i valori numerici corrispondenti alle lettere della parola «amore», oltre al codice *<LF>* e allo zero finale. ***b[]*** è un array di interi, di sei elementi, contenenti i valori numerici corrispondenti alle lettere della parola «amore», oltre allo zero finale. ***c[]*** è un array di interi, di sei elementi, contenenti i valori numerici corrispondenti alle lettere della parola «amore», oltre al codice *<LF>* finale. ***d[]*** è un array di interi, di cinque elementi, contenenti i valori numerici corrispondenti alle lettere della parola «amore» e nulla altro. ***e[]*** è un array di caratteri, di sette elementi, contenenti i valori numerici corrispondenti alle lettere della parola «amore», oltre al codice *<LF>* e allo zero finale: si tratta di una stringa che se visualizzata porta anche a capo il cursore al termine. ***f[]*** è un array di caratteri, di sette elementi, contenenti i valori numerici corrispondenti alle lettere della parola «amore», oltre allo zero finale: si tratta di una stringa che se visualizzata non porta a capo il cursore al termine. ***g[]*** è un array di caratteri, di sette elementi, contenenti i valori numerici corrispondenti alle lettere della parola «amore», oltre allo al codice *<LF>* finale: non si tratta di una stringa, perché manca lo zero finale. ***g[]*** è un array di caratteri, di sei elementi, contenenti i valori numerici corrispondenti alle lettere della parola «amore» e nulla altro: non si tratta di una stringa, perché manca lo zero finale. ***i[]*** è un array di caratteri, di nove elementi, contenenti i valori numerici corrispondenti alle lettere della parola «amore», uno zero e poi le lettere della parola «mio»: può valere come stringa, ma in tal caso si ignora il testo successivo allo zero.

Esercizio	Soluzione
82.13.2	***a[]*** è un array di caratteri che nel corso del programma potrebbe anche contenere una stringa. ***b[]*** è un array di caratteri contenente inizialmente una stringa. Non è possibile assegnare qualcosa direttamente a ***c***, perché si tratta di un puntatore in sola lettura; per cambiare il contenuto dell'array ***c[]*** bisogna invece intervenire cella per cella. ***e*** è un puntatore che può ricevere l'indirizzo iniziale di un array di caratteri; pertanto l'assegnamento è valido ed ***e*** diventa un modo alternativo per fare riferimento alla stringa contenuto nell'array ***d[]***. Dopo l'assegnamento, ***f*** è un puntatore a un carattere che contiene il valore corrispondente alla lettera «c»; tuttavia può essere usato in qualità di stringa, contenente la parola «ciao». Dopo l'assegnamento, ***g*** punta all'inizio di una stringa che rappresenta la parola «amore»; per converso, la stringa che rappresentava la parola «ciao» continua a occupare spazio in memoria, ma risulta irraggiungibile. Con l'assegnamento di ****h***, si vorrebbe sostituire l'iniziale della parola «ciao» con una maiuscola; tuttavia, ciò non è ammissibile, perché l'area di memoria che contiene inizialmente la stringa «ciao» dovrebbe essere in sola lettura. Con l'incremento di ***i***, questo puntatore rappresenta una stringa, contenente però solo la parola «iao».
82.14.1	***a[1]*** contiene 4369_{16}. ***b[0]*** contiene 2061_{16}. ***c*** contiene 7265_{16}. ***d[0]*** contiene 0052_{16}. ****d[0]*** contiene 0000_{16} e corrisponde a ***a[0]***. ****e*** contiene 0052_{16} e corrisponde a ***d[0]***. *****e*** contiene 0000_{16} e corrisponde a ***a[0]***, così come a ****d[0]***. ****f*** contiene $005E_{16}$ e corrisponde a ***e***. *****f*** contiene 0052_{16} e corrisponde a ***d[0]***, così come a ****e***. ******f*** contiene 0000_{16} e corrisponde a ***a[0]***, così come a ****d[0]*** e a *****e***.

Esercizio	Soluzione
82.15.1	***a[1]*** contiene 4369_{16}. ***b[0]*** contiene 2061_{16}. ***c*** contiene 7265_{16}. ***d[0]*** contiene 0052_{16}. ***d[0][0]***, ovvero ****d[0]***, contiene 0000_{16} e corrisponde a ***a[0]***. ***d[0][1]*** contiene 4369_{16} e corrisponde a ***a[1]***. ***d[1][0]*** contiene 2061_{16} e corrisponde a ***b[0]***. ***d[1][1]*** contiene $6D6F_{16}$ e corrisponde a ***b[1]***. ***d[2][0]*** contiene 7265_{16} e corrisponde a ***c***. ***e[0]*** contiene 0052_{16} e corrisponde a ***d[0]***. ***e[0][0]***, ovvero ****e[0]***, contiene 0000_{16} e corrisponde a ***a[0]***, così come a ***d[0][0]***. ***e[0][1]*** contiene 4369_{16} e corrisponde a ***a[1]***, così come a ***d[0][1]***. ***e[1][0]*** contiene 2061_{16} e corrisponde a ***b[0]***, così come a ***d[1][0]***. ***e[1][1]*** contiene $6D6F_{16}$ e corrisponde a ***b[1]***, così come a ***d[1][1]***. ***e[2][0]*** contiene 7265_{16} e corrisponde a ***c***, così come a ***d[2][0]***. ***f[0]*** contiene $005E_{16}$ e corrisponde a ***e***. ***f[0][0]*** contiene 0052_{16} e corrisponde a ***d[0]*** e a ***e[0]***. ***f[0][0][0]***, ovvero ******f***, contiene 0000_{16} e corrisponde a ***a[0]***, così come a ***d[0][0]*** e a ***e[0][0]***. ***f[0][0][1]*** contiene 4369_{16} e corrisponde a ***a[1]***, così come a ***d[0][1]*** e a ***e[0][1]***. ***f[0][1][0]*** contiene 2061_{16} e corrisponde a ***b[0]***, così come a ***d[1][0]*** e a ***e[1][0]***. ***f[0][1][1]*** contiene $6D6F_{16}$ e corrisponde a ***b[1]***, così come a ***d[1][1]*** e a ***e[1][1]***. ***f[0][2][0]*** contiene 7265_{16} e corrisponde a ***c***, così come a ***d[2][0]*** e a ***e[2][0]***.

[1] In contesti particolari è ammissibile che ***argc*** sia pari a zero, a indicare che non viene fornita alcuna informazione; oppure, se gli argomenti vengono forniti ma il nome del programma è assente, ***argv[0][0]*** deve essere pari a *<NUL>*, ovvero al carattere nullo.

Indice analitico del volume

! 586 590 2402 2413 != 586 590 2402 2410 * 586 588 679 2402 2406 2478 ** 723 725 2517 2522 *** 723 2517 *...const 701 *= 586 588 2402 2406 *& 709 + 586 588 2402 2406 ++ 586 588 2402 2406 += 586 588 2402 2406 . 759 761 .ascii 239 .bss 241 .byte 239 .data 241 .equ 238 .int 239 .lcomm 239 .odbc.ini 2121 .text 241 / 586 588 2402 2406 /*...*/ 564 2375 // 564 2375 /= 586 588 2402 2406 0... 575 2391 01 1548 0x... 575 2391 2421 134 5211 134 631-1 134 66 1589 732-1 134 8421 134 88 1586 ; 564 2375 = 586 588 2402 2406 == 586 590 2402 2410 ? : 586 590 2402 2413 a.out 132 567 2380 abort 1379 abort() 1141 abs() 1146 1379 ACCEPT 1628 access() 1331 1333 *actual argument* 670 ADC 189 255 ADD 189 235 248 1633 addizione binaria 2312 AH 183 AL 183 alarm() 1331 allineamento della memoria 442 ALTER TABLE 1914 ALTER USER 1938 AND 170 2345 and 1167 AND 193 and_eq 1167 argc 720 2530 argomento attuale 670 argv 720 2530 *array* 110 375 687 2351 2490 *array* di puntatori 725 2522 asctime() 1247 1399 assemblatore 124 213 *assembler* 213 assembler 124 assembly 124 *assembly* 213 assert() 1069 1435 assert.h 1069 1435 atexit() 1141 1379 atof() 1130 1379 atoi() 1130 1379 atol() 1130 1379 atoll() 1130 1379 auto 657 AX 183 basi di dati 1859 BCD 134 BH 183 *big endian* 119 2358 big endian 146 bit 570 2384 bitand 1167 BL 183 BLANK WHEN ZERO 1569 blkcnt_t 1312 blksize_t 1312 BLOCK CONTAINS 1543 bool 778 1168 *borrow* 152 160 2330 BP 183 break 599 602 604 2428 2433 2437 BRKINT 1347 bsearch() 1143 1379 BSWAP 187 Bubblesort 37 896 BUFSIZ 1256 1406 BX

183 byte 570 671 2384 byte order 146 *byte order* 119 2358 C 495 563 2373 2475 CALL 196 322 350 calloc() 737 1140 1379 campo 770 carattere 570 671 2384 carattere esteso 792 caratteristica 142 caricamento di un programma 432 *carry* 152 160 164 164 165 169 183 254 2330 2340 2341 case 599 2428 *cast* 594 2418 CBW 187 cc 509 cc_t 1345 CDQ 187 CGI 2041 CH 183 char 571 2385 CHAR_BIT 1071 1360 CHAR_MAX 1071 1360 CHAR_MIN 1071 1360 chdir() 1331 1338 chmod() 1320 1442 chown() 1331 CL 183 CLC 200 clearerr() 1300 1406 clock() 1239 1399 CLOCKS_PER_SEC 1239 clock_t 1239 1312 CLOSE 1635 1938 close() 980 1331 closedir() 1048 1344 CMC 200 CMP 200 290 301 306 COBOL 1733 1733 CODE-SET 1547 codice di interruzione di riga 826 codice pesato 134 collegamento 650 COMMIT 1944 comparazione binaria 174 compilazione di un programma 432 compl 1167 complemento alla base 2308 complemento a due 138 2311 2316 complemento a uno 136 2311 COMPUTE 1636 condotto 1034 CONFIGURATION SECTION 1503 confstr() 1331 const 583 584 2400 const...* 701 const volatile 584 continue 602 604 2433 2437 convenzione di chiamata 348 conversione di tipo 594 2418 *conversion specifier* 568 2382 costante letterale composta 773 cpp 511 619 creat() 980 1327 CREATE DATABASE 1941 CREATE TABLE 1908 CREATE USER 1938 CREATE VIEW 1934 ctermid() 1299 ctime() 1248 1399 ctype.h 1101 1375 CWDE 187 CX 183 data 1897 DATA DIVISION 1538 DATA RECORD 1545 db 239 DBA 1866 DBMS 1859 DCL 1938 dd 239 DDD 231 499 DDL 1866 1893 DEC 189 236 306 DECLARATIVES 1615 DECLARE 1935 default 599 2428 DELETE 1636 DELETE FROM 1919 DEPENDING ON 1576 descrittore 979 Dev86 815

dev_t 1312 DH 183 DI 183 difftime() 1244 1399 *digraph* 580 676 2397 DIR 1048 1048 1343 *directory* 1047 directory 1001 dirent.h 1047 1342 dislocamento 298 *displacement* 298 DISPLAY 1638 DIV 189 272 div() 1146 1379 DIVIDE 1639 divisione binaria 2315 div_t 1128 1128 DL 183 DML 1866 1915 do 603 2436 double 571 2385 DROP TABLE 1915 DROP USER 1938 DROP VIEW 1934 DSN 2121 dup() 1331 dup2() 1331 durata di memorizzazione 673 DX 183 EAX 183 EBP 183 334 EBX 183 eccesso 3 134 ECHO 1349 ECHOE 1349 ECHOK 1349 ECHONL 1349 ECX 183 EDI 183 EDOM 1087 EDX 183 EFLAGS 183 EILSEQ 1087 EIP 183 ELF 453 else 598 2423 *endianess* 119 2358 endianess 146 ENTER 341 350 enum 755 enumerazione 755 environ 950 ENVIRONMENT DIVISION 1502 EOF 828 1256 1406 equ 238 ERANGE 1087 errno 847 995 1087 errno.h 1087 errore di segmentazione 439 esecuzione di un programma 432 ESI 183 ESP 183 esponente 142 espressione multipla 596 2420 espressione regolare 916 execl() 943 1331 execle() 1331 execlp() 1331 execv() 1331 execve() 951 1331 execvp() 1331 EXIT 1642 exit() 615 1141 1379 2450 EXIT_FAILURE 1128 EXIT_SUCCESS 1128 extern 651 657 *external linkage* 650 extern const volatile 584 F 575 2391 false 1168 fchmod() 1320 1442 fchown() 1331 fclose() 832 1264 1406 fcntl() 1327 fcntl.h 979 1322 FD 1541 fdopen() 1264 FD_CLOEXEC 1049 1323 feof() 1300 1406 ferror() 1300 1406 FETCH 1937 fflush() 1268 1406 ffs() 1321 fgetc() 1289 1406 fgetpos() 1295 1406 fgets() 843 1292 1406 Fibonacci 32 887 FIFO 1034 file 979 FILE 788 822 832 1255 *file* 1511 FILENAME_MAX 1256 1406 FILE-CONTROL 1509 file di intestazione 621 file eseguibile 435

file oggetto 433 FILE SECTION 1540 file speciale 1022 *file system* 996 1008 FILLER 1560 *flag* 126 160 FLAGS 183 float 571 2385 flockfile() 1297 flusso di controllo 953 flusso di file 822 fopen() 832 1264 1406 FOPEN_MAX 1256 1406 for 604 2437 fork() 940 1331 *formal parameter* 670 formato a.out 132 fpathconf() 1331 fpos_t 788 1255 fprintf() 1278 1421 fputc() 1289 1406 fputs() 843 1292 1406 fread() 835 1294 1406 free() 737 1140 1379 freopen() 1264 1406 fscanf() 1287 1428 fseek() 840 1295 1406 fseeko() 1295 fsetpos() 1295 1406 fstat() 1320 1442 ftell() 840 1295 1406 ftello() 1295 ftruncate() 1331 ftrylockfile() 1297 *function-like macro* 623 funlockfile() 1297 fusione 56 fwrite() 835 1294 1406 F_DUPFD 1323 F_GETFD 1323 F_GETFL 1323 F_GETLK 1323 F_GETOWN 1323 F_OK 1330 F_RDLCK 1326 F_SETFD 1323 F_SETFL 1323 F_SETLK 1323 F_SETLKW 1323 F_SETOWN 1323 F_UNLCK 1326 F_WRLCK 1326 *garbage collector* 737 GAS 213 GCC 509 gcc 509 GDB 217 499 getc() 1289 getchar() 1289 getchar_unlocked() 1297 getcwd() 1331 1337 getc_unlocked() 1297 getegid() 1331 getenv() 1142 geteuid() 1331 getgid() 1331 getgroups() 1331 gethostname() 1331 getlogin() 1331 getlogin_r() 1331 getopt() 1331 getpgrp() 1331 getpid() 1331 getppid() 1331 gets() 1292 1406 Gettext 1449 getuid() 1331 gid_t 1312 gmtime() 1246 1399 GNU AS 213 GO TO 1644 GRANT 1942 Hanoi 41 899 *header file* 621 ICANON 1349 ICRNL 1347 IDENTIFICATION DIVISION 1500 IDIV 189 272 id_t 1312 IEEE 754 142 IEXTEN 1349 if 598 2423 IF 1645 IGNBRK 1347 IGNCR 1347 IGNPAR 1347 imaxabs()

1165 imaxdiv() 1157 imaxdiv_t 1157 immagine 472
immagine di un processo elaborativo 439 IMUL 189 269 INC 189
236 306 indicatore 126 160 indirizzamento 358 init 946
initdb 1954 INLCR 1347 inode 998 ino_t 1312 INPCK 1347
INPUT-OUTPUT SECTION 1508 INSERT INTO 1916 1933
INSPECT 1646 int 571 2385 INT 196 213 INT16_C() 1079
1361 INT16_MAX 1078 1361 INT16_MIN 1078 1361 int16_t
1078 1361 INT32_C() 1079 1361 INT32_MAX 1078 1361
INT32_MIN 1078 1361 int32_t 1078 1361 INT64_C() 1079
1361 INT64_MAX 1078 1361 INT64_MIN 1078 1361 int64_t
1078 1361 INT8_C() 1079 1361 INT8_MAX 1078 1361
INT8_MIN 1078 1361 int8_t 1078 1361 *internal linkage* 650
intero con segno 138 2316 intero senza segno 136 2315
interruzione di riga 826 INTMAX_C() 1085 1361 INTMAX_MAX
1085 1361 INTMAX_MIN 1085 1361 intmax_t 1085 1361
INTPTR_MAX 1084 1361 INTPTR_MIN 1084 1361 intptr_t
1084 1361 inttypes.h 1156 1365 INT_FAST16_MAX 1082
1361 INT_FAST16_MIN 1082 1361 int_fast16_t 1082 1361
INT_FAST32_MAX 1082 1361 INT_FAST32_MIN 1082 1361
int_fast32_t 1082 1361 INT_FAST64_MAX 1082 1361
INT_FAST64_MIN 1082 1361 int_fast64_t 1082 1361
INT_FAST8_MAX 1082 1361 INT_FAST8_MIN 1082 1361
int_fast8_t 1082 1361 INT_LEAST16_MAX 1079 1361
INT_LEAST16_MIN 1079 1361 int_least16_t 1079 1361
INT_LEAST32_MAX 1079 1361 INT_LEAST32_MIN 1079
1361 int_least32_t 1079 1361 INT_LEAST64_MAX 1079
1361 INT_LEAST64_MIN 1079 1361 int_least64_t 1079
1361 INT_LEAST8_MAX 1079 1361 INT_LEAST8_MIN 1079
1361 int_least8_t 1079 1361 INT_MAX 1071 1360

INT_MIN 1071 1360 IP 183 isalnum() 1102 1375 isalpha() 1103 1375 isascii() 1114 isatty() 1331 isblank() 1104 1375 iscntrl() 1105 1375 isdigit() 1106 1375 isgraph() 1107 1375 ISIG 1349 islower() 1108 1375 iso646.h 1167 ISO 10646 1611 ISO 8601 1897 isprint() 1109 1375 ispunct() 1110 1375 isql 2129 isspace() 1111 1375 ISTRIP 1347 isupper() 1112 1375 isxdigit() 1113 1375 iusql 2129 IXOFF 1347 IXON 1347 I-O-CONTROL 1528 JA 201 301 JAE 201 301 JB 201 301 312 JBE 201 301 JC 201 300 JCXZ 201 JE 201 301 306 JG 201 301 JGE 201 301 JL 201 301 JLE 201 301 JMP 201 299 309 JNA 201 301 JNAE 201 301 JNB 201 JNBE 201 301 JNC 201 300 322 JNE 201 301 JNG 201 301 JNGE 201 301 JNL 201 301 JNLE 301 JNO 201 300 JNP 201 300 JNS 201 300 JNZ 201 300 306 JO 201 300 JP 201 300 JS 201 300 JUSTIFIED RIGHT 1569 JZ 201 300 314 L 575 575 2391 2391 LABEL RECORD 1545 labs() 1146 1379 LC_TIME 1249 ldiv() 1146 1379 ldiv_t 1128 LEA 187 366 LEAVE 341 350 LibPQ 1973 libreria dinamica 419 libreria statica 432 LIFO 99 2347 limits.h 1071 1360 *link* 132 650 link() 1331 1340 *linkage* esterno 672 *linkage* interno 672 *link script* 443 little endian 146 *little endian* 119 2358 LL 575 2391 llabs() 1146 1379 lldiv() 1146 1379 lldiv_t 1128 LLONG_MAX 1071 1360 LLONG_MIN 1071 1360 locale.h 790 1092 1436 localtime() 1246 1399 long 571 2385 LONG_BIT 1075 LONG_MAX 1071 1360 LONG_MIN 1071 1360 long long 571 2385 LOOP 211 304 306 LOOPE 211 304 LOOPNE 211 304 LOOPNZ 211 304 LOOPZ 211 304 lseek() 992 1331 lstat() 1320 1442 *lvalue* 674 678 2477 L"..." 792 L_ctermid 1256 L_tmpnam 1256 1406 L'...' 792 main()

720 2530 major() 1028 Make 523 makedev() 1028 Makefile 523 malloc() 737 1140 1379 mantissa 142 mblen() 1149 1379 mbstowcs() 1153 1379 mbtowc() 1151 1379 MB_CUR_MAX 1128 MB_LEN_MAX 1071 1360 membro di una struttura 759 memccpy() 1172 memchr() 1192 1390 memcmp() 1184 1390 memcpy() 1171 1390 memmove() 1174 1390 *memory pad* 442 memset() 1217 1390 MERGE 1717 Minix 1008 minor() 1028 mkdir() 1031 1320 1442 mkfifo() 1032 1043 1320 1442 mknod() 1023 1320 1442 mktime() 1244 1399 mode_t 1312 1316 moltiplicazione binaria 2314 MOV 187 213 234 MOVE 1654 MOVSX 187 MOVZX 187 322 MUL 189 266 *multiboot specification* 471 multibyte 792 1148 MULTIPLY 1658 my.cnf 2067 2069 mysql 2067 2074 MySQL 2067 mysqladmin 2067 mysqld 2067 mysqldump 2100 mysql_install_db 2086 NASM 213 NCCS 1346 NDEBUG 1069 NEG 189 261 *new-line* 826 nlink_t 1312 NOFLSH 1349 NOP 187 NOT 193 not 1167 NOT 170 2345 not_eq 1167 NULL 735 1169 numero 1895 numero intero con segno 138 2316 numero intero senza segno 136 2315 numero in virgola mobile 140 2321 Objdump 213 OBJECT-COMPUTER 1504 *object-like macro* 623 OCCURS 1567 1571 ODBC 1890 2121 odbc.ini 2121 ODBCConfig 2124 ODBCDataSources 2121 odbcinst.ini 2121 offsetof 767 1169 offsetof() 1435 off_t 1312 *opcode* 127 OPEN 1660 1935 open() 980 1327 OpenCOBOL 1742 opendir() 1048 1344 operatore logico 170 2345 OPOST 1348 or 1167 OR 193 OR 170 2345 ora 1897 ordine dei byte 119 2358 organizzazione del *file* 1511 or_eq 1167 ottimizzazione 520 *overflow* 149 160 165 183 253 2326 O_ACCMODE 1323 O_APPEND 980 1323 O_CREAT 980 1323 O_DSYNC 1323

O_EXCL 980 1323 O_NOCTTY 980 1323 O_NONBLOCK 980 1323 O_RDONLY 980 1323 O_RDWR 980 1323 O_RSYNC 1323 O_SYNC 980 1323 O_TRUNC 980 1323 O_WRONLY 980 1323 PAGER 1982 parametro formale 670 *parity* 183 PARMRK 1347 parola 124 pathconf() 1331 pause() 1331 pclose() 1299 PERFORM 1664 perror() 1300 1406 PgAccess 2027 pg_database 1987 pg_dump 1991 pg_dumpall 1991 pg_hba.conf 1964 1966 pg_shadow 1987 pg_user 1987 PICTURE 1594 pid_t 1312 pila 99 2347 *pipe* 1034 pipe() 1038 1331 POP 196 326 POPA 196 341 350 POPAD 196 popen() 1299 POPF 196 PostgreSQL 1949 2027 postgresql.conf 1964 postmaster 1957 postmaster.conf 1964 precedenza operatori 586 2402 prestito 152 2330 PRId16 1158 1365 PRId32 1158 1365 PRId64 1158 1365 PRId8 1158 1365 PRIdFAST16 1158 1365 PRIdFAST32 1158 1365 PRIdFAST64 1158 1365 PRIdFAST8 1158 1365 PRIdLEAST16 1158 1365 PRIdLEAST32 1158 1365 PRIdLEAST64 1158 1365 PRIdLEAST8 1158 1365 PRIdMAX 1158 1365 PRIdPTR 1158 1365 PRIi16 1158 1365 PRIi32 1158 1365 PRIi64 1158 1365 PRIi8 1158 1365 PRIiFAST16 1158 1365 PRIiFAST32 1158 1365 PRIiFAST64 1158 1365 PRIiFAST8 1158 1365 PRIiLEAST16 1158 1365 PRIiLEAST32 1158 1365 PRIiLEAST64 1158 1365 PRIiLEAST8 1158 1365 PRIiMAX 1158 1365 PRIiPTR 1158 1365 printf() 568 753 860 1278 1421 2382 PRIo16 1158 1365 PRIo32 1158 1365 PRIo64 1158 1365 PRIo8 1158 1365 PRIoFAST16 1158 1365 PRIoFAST32 1158 1365 PRIoFAST64 1158 1365 PRIoFAST8 1158 1365 PRIoLEAST16 1158 1365 PRIoLEAST32 1158 1365 PRIoLEAST64 1158 1365 PRIoLEAST8 1158 1365 PRIoMAX

1158 1365 PRIoPTR 1158 1365 PRIu16 1158 1365 PRIu32 1158 1365 PRIu64 1158 1365 PRIu8 1158 1365 PRIuFAST16 1158 1365 PRIuFAST32 1158 1365 PRIuFAST64 1158 1365 PRIuFAST8 1158 1365 PRIuLEAST16 1158 1365 PRIuLEAST32 1158 1365 PRIuLEAST64 1158 1365 PRIuLEAST8 1158 1365 PRIuMAX 1158 1365 PRIuPTR 1158 1365 PRIx16 1158 1365 PRIX16 1158 1365 PRIX32 1158 1365 PRIx32 1158 1365 PRIx64 1158 1365 PRIX64 1158 1365 PRIx8 1158 1365 PRIX8 1158 1365 PRIxFAST16 1158 1365 PRIXFAST16 1158 1365 PRIXFAST32 1158 1365 PRIxFAST32 1158 1365 PRIxFAST64 1158 1365 PRIXFAST64 1158 1365 PRIxFAST8 1158 1365 PRIXFAST8 1158 1365 PRIxLEAST16 1158 1365 PRIXLEAST16 1158 1365 PRIXLEAST32 1158 1365 PRIxLEAST32 1158 1365 PRIxLEAST64 1158 1365 PRIXLEAST64 1158 1365 PRIxLEAST8 1158 1365 PRIXLEAST8 1158 1365 PRIXMAX 1158 1365 PRIxMAX 1158 1365 PRIxPTR 1158 1365 PRIXPTR 1158 1365 PROCEDURE DIVISION 1613 1628 1713 processo elaborativo in memoria 439 programma autonomo 470 programma *stand alone* 470 *promotion* 746 promozione 746 prototipo di funzione 608 2441 pseudocodifica 20 psql 1973 pthread_t 954 1312 PTRDIFF_MAX 1086 1361 PTRDIFF_MIN 1086 1361 ptrdiff_t 783 1086 1169 1361 puntatore 679 704 2478 2507 puntatore a funzione 730 puntatore a puntatori 723 725 2517 2522 puntatore nullo 735 PUSH 196 326 PUSHA 196 341 350 PUSHF 196 putc() 1289 putchar() 1289 1406 putchar_unlocked() 1297 putc_unlocked() 1297 puts() 843 1292 1406 P_tmpdir 1256 qsort() 1143 1379 Quicksort 44 900 raise() 1233 rand() 1136 1379 RAND_MAX

1128 rango 569 2383 2383 *rank* 569 2383 2383 RCL 193 287 RCR 193 287 RDBMS 1867 READ 1675 read() 987 1331 readdir() 1048 1344 readlink() 1331 realloc() 737 1140 1379 RECORD CONTAINS 1546 REDEFINES 1548 1560 regcomp() 918 919 1437 regerror() 918 932 1437 regex.h 918 1437 regexec() 918 923 927 1437 regexp 916 regex_t 918 919 1437 regfree() 918 923 1437 register 657 registro 124 183 regmatch_t 918 923 927 1437 1437 regoff_t 1437 relazione 1867 RELEASE 1724 remove() 1034 1261 1406 rename() 1261 1406 RENAMES 1589 reopen() 846 resb 239 resd 239 restrict 740 resw 239 RET 196 322 350 return 609 2443 RETURN 1722 REVOKE 1942 rewind() 1295 1406 rewinddir() 1048 1344 REWRITE 1681 re_sub 1437 ricerca binaria 36 riordino 53 56 riporto 149 152 164 164 165 169 254 2326 2330 2340 2341 rmdir() 1033 1331 rm_se 1437 rm_so 1437 ROL 193 284 ROLLBACK 1944 ROR 193 284 rotazione 168 2344 *rvalue* 674 R_OK 1330 SAL 193 281 SAR 193 281 SBB 189 263 scanf() 869 1287 1428 SCHAR_MAX 1071 1360 SCHAR_MIN 1071 1360 SCNd16 1158 1365 SCNd32 1158 1365 SCNd64 1158 1365 SCNd8 1158 1365 SCNdFAST16 1158 1365 SCNdFAST32 1158 1365 SCNdFAST64 1158 1365 SCNdFAST8 1158 1365 SCNdLEAST16 1158 1365 SCNdLEAST32 1158 1365 SCNdLEAST64 1158 1365 SCNdLEAST8 1158 1365 SCNdMAX 1158 1365 SCNdPTR 1158 1365 SCNi16 1158 1365 SCNi32 1158 1365 SCNi64 1158 1365 SCNi8 1158 1365 SCNiFAST16 1158 1365 SCNiFAST32 1158 1365 SCNiFAST64 1158 1365 SCNiFAST8 1158 1365 SCNiLEAST16 1158 1365 SCNiLEAST32 1158 1365 SCNiLEAST64 1158 1365 SCNiLEAST8 1158 1365 SCNiMAX

1158 1365 SCNiPTR 1158 1365 SCNo16 1158 1365 SCNo32 1158 1365 SCNo64 1158 1365 SCNo8 1158 1365 SCNoFAST16 1158 1365 SCNoFAST32 1158 1365 SCNoFAST64 1158 1365 SCNoFAST8 1158 1365 SCNoLEAST16 1158 1365 SCNoLEAST32 1158 1365 SCNoLEAST64 1158 1365 SCNoLEAST8 1158 1365 SCNoMAX 1158 1365 SCNoPTR 1158 1365 SCNu16 1158 1365 SCNu32 1158 1365 SCNu64 1158 1365 SCNu8 1158 1365 SCNuFAST16 1158 1365 SCNuFAST32 1158 1365 SCNuFAST64 1158 1365 SCNuFAST8 1158 1365 SCNuLEAST16 1158 1365 SCNuLEAST32 1158 1365 SCNuLEAST64 1158 1365 SCNuLEAST8 1158 1365 SCNuMAX 1158 1365 SCNuPTR 1158 1365 SCNx16 1158 1365 SCNx32 1158 1365 SCNx64 1158 1365 SCNx8 1158 1365 SCNxFAST16 1158 1365 SCNxFAST32 1158 1365 SCNxFAST64 1158 1365 SCNxFAST8 1158 1365 SCNxLEAST16 1158 1365 SCNxLEAST32 1158 1365 SCNxLEAST64 1158 1365 SCNxLEAST8 1158 1365 SCNxMAX 1158 1365 SCNxPTR 1158 1365 scorrimento 164 164 2340 2341 SD 1543 SEARCH 1684 SEEK_CUR 1256 1406 SEEK_END 1256 1406 SEEK_SET 1256 1406 *segmentation fault* 439 segno 147 2323 SELECT 1512 1516 1522 1919 sequenza multibyte 1148 SET 1695 SETA 205 SETAE 205 SETB 205 SETBE 205 setbuf() 1268 1406 SETC 205 SETE 205 setegid() 1331 seteuid() 1331 SETG 205 SETGE 205 setgid() 1331 SETL 205 SETLE 205 setlocale() 1436 SETNA 205 SETNAE 205 SETNB 205 SETNBE 205 SETNC 205 SETNE 205 SETNG 205 SETNGE 205 SETNL 205 SETNLE 205 SETNO 205 SETNS 205 SETNZ 205 SETO 205 setpgid() 1331 SETS 205 setsid() 1331 setuid() 1331 setvbuf() 1268 1406 SETZ 205 *shared*

object 421 *shift* 164 164 275 2340 2341 SHL 193 277 316 short 571 2385 SHR 193 277 316 SHRT_MAX 1071 1360 SHRT_MIN 1071 1360 SI 183 SIGABRT 1226 1227 SIGALRM 1227 SIGBUS 1227 SIGCHLD 1227 SIGCONT 1227 SIGFPE 1226 1227 SIGHUP 1227 SIGILL 1226 1227 SIGINT 1226 1227 SIGKILL 1227 *sign* 160 signal() 1233 signal.h 1223 signed 571 2385 significante 142 SIGN IS 1566 SIGPIPE 1227 SIGPOLL 1227 SIGPROF 1227 SIGQUIT 1227 SIGSEGV 1226 1227 SIGSTOP 1227 SIGSYS 1227 SIGTERM 1226 1227 SIGTRAP 1227 SIGTTIN 1227 SIGTTOU 1227 SIGURG 1227 SIGUSR1 1227 SIGUSR2 1227 SIGVTALRM 1227 SIGXCPU 1227 SIGXFSZ 1227 SIG_ATOMIC_MAX 1086 1361 SIG_ATOMIC_MIN 1086 1361 sig_atomic_t 1086 1225 1361 SIG_DFL 1231 SIG_ERR 1231 SIG_IGN 1231 sistema binario 2282 sistema decimale 2281 sistema esadecimale 2285 sistema ottale 2284 sizeof 687 SIZE_MAX 1086 1361 size_t 781 1086 1169 1312 1361 sleep() 1331 snprintf() 1278 1421 somma binaria 2312 SORT 1524 1713 sottrazione binaria 2313 SOURCE-COMPUTER 1504 SP 183 SPECIAL-NAMES 1505 specificatore di conversione 568 2382 specifiche *multiboot* 471 speed_t 1345 sprintf() 1278 1421 SQL 1890 2000 2041 SQLite 2103 srand() 1136 1379 sscanf() 1287 1428 SSIZE_MAX 1075 ssize_t 1312 *stack* 99 2347 *stack frame* 341 *stand alone* 470 START 1699 stat() 1320 1442 stat.h 979 1315 1442 static 651 657 stdarg.h 746 1123 1359 stdbool.h 1168 stddef.h 1169 1435 stderr 845 1260 STDERR_FILENO 1330 stdint.h 1077 1361 STDIN_FILENO 1330 stdio 845 1260 stdio.h 822 1254 1406 1421 1428 stdlib.h 737 1127 1379 stdout 845 1260 STDOUT_FILENO

1330 STOP RUN 1703 *storage duration* 673 strcasecmp() 1322 strcat() 714 1181 1390 strchr() 714 1193 1390 strcmp() 714 1186 1390 strcoll() 714 1188 1390 strcpy() 714 1175 1390 strcspn() 714 1198 1390 strdup() 1178 *stream* 822 strerror() 1219 1390 strerror_r() 1220 strftime() 1249 1399 STRING 1704 string.h 714 1170 1390 stringa 120 710 1893 2360 2511 stringa estesa 792 strings.h 1321 strlen() 714 1221 1390 strncasecmp() 1322 strncat() 714 1182 1390 strncmp() 714 1188 1390 strncpy() 714 1176 1390 strpbrk() 714 1200 1390 strrchr() 714 1195 1390 strspn() 714 1197 1390 strstr() 1202 1390 strtod() 1130 1379 strtof() 1130 1379 strtoimax() 1166 strtok() 1204 1390 strtok_r() 1211 strtol() 1130 1379 strtold() 1130 1379 strtoll() 1130 1379 strtouimax() 1166 strtoul() 1130 1379 strtoull() 1130 1379 struct 759 structure stat 1319 struct dirent 1048 1048 1343 struct termios 1346 struct tm 787 1242 struttura 759 strxfrm() 1190 1390 st_atime 1319 1442 st_blksize 1319 1442 st_blocks 1319 1442 st_ctime 1319 1442 st_dev 1319 1442 st_gid 1319 1442 st_ino 1319 1442 st_mode 1319 1442 st_mtime 1319 1442 st_nlink 1319 1442 st_rdev 1319 1442 st_size 1319 1442 st_uid 1319 1442 SUB 189 235 259 SUBTRACT 1707 super blocco 997 switch 599 2428 symlink() 1331 SYNCHRONIZED 1568 sysconf() 1331 system() 1142 S_IFBLK 1316 1442 S_IFCHR 1316 1442 S_IFDIR 1316 1442 S_IFIFO 1316 1442 S_IFLNK 1316 1442 S_IFMT 1316 1442 S_IFREG 1316 1442 S_IFSOCK 1316 1442 S_IRGRP 980 1316

1442 S_IROTH 980 1316 1442 S_IRUSR 980 1316 1442 S_IRWXG 980 1316 1442 S_IRWXO 980 1316 1442 S_IRWXU 980 1316 1442 S_ISBLK() 1316 1442 S_ISCHR() 1316 1442 S_ISDIR() 1316 1442 S_ISFIFO() 1316 1442 S_ISGID 980 1316 1442 S_ISLNK() 1316 1442 S_ISREG() 1316 1442 S_ISSOCK() 1316 1442 S_ISUID 980 1316 1442 S_ISVTX 980 1316 1442 S_IWGRP 980 1316 1442 S_IWOTH 980 1316 1442 S_IWUSR 980 1316 1442 S_IXGRP 980 1316 1442 S_IXOTH 980 1316 1442 S_IXUSR 980 1316 1442 tcflag_t 1345 tcgetattr() 1349 tcgetpgrp() 1331 TCSADRAIN 1349 TCSAFLUSH 1349 TCSANOW 1349 tcsetattr() 1349 tcsetpgrp() 1331 tempnam() 1261 termios.h 1345 TEST 200 *thread* 953 time() 1243 1399 time.h 1239 1399 time_t 787 1241 1243 1312 TinyCOBOL 1740 tmpfile() 1261 1406 tmpnam() 1261 1406 TMP_MAX 1256 1406 toascii() 1120 tolower() 1117 1375 TOSTOP 1349 toupper() 1118 1375 traboccamento 149 165 253 2326 *translation unit* 672 *trigraph* 580 676 2397 true 1168 ttyname() 1331 ttyname_r() 1331 tupla 1867 typedef 772 types.h 1312 U 575 2391 UCHAR_MAX 1071 1360 uid_t 1312 UINT16_C() 1079 1361 UINT16_MAX 1078 1361 uint16_t 1078 1361 UINT32_C() 1079 1361 UINT32_MAX 1078 1361 uint32_t 1078 1361 UINT64_C() 1079 1361 UINT64_MAX 1078 1361 uint64_t 1078 1361 UINT8_C() 1079 1361 UINT8_MAX 1078 1361 uint8_t 1078 1361 UINTMAX_C() 1085 1361 UINTMAX_MAX 1085 1361 uintmax_t 1085 1361 UINTPTR_MAX 1084 1361 uintptr_t 1084 1361 UINT_FAST16_MAX 1082 1361 uint_fast16_t 1082 1361 UINT_FAST32_MAX 1082 1361 uint_fast32_t

1082 1361 UINT_FAST64_MAX 1082 1361 uint_fast64_t 1082 1361 UINT_FAST8_MAX 1082 1361 uint_fast8_t 1082 1361 UINT_LEAST16_MAX 1079 1361 uint_least16_t 1079 1361 UINT_LEAST32_MAX 1079 1361 uint_least32_t 1079 1361 UINT_LEAST64_MAX 1079 1361 uint_least64_t 1079 1361 UINT_LEAST8_MAX 1079 1361 uint_least8_t 1079 1361 UINT_MAX 1071 1360 UL 575 2391 ULL 575 2391 ULLONG_MAX 1071 1360 ULONG_MAX 1071 1360 umask() 1320 1442 ungetc() 1289 1406 Unicode 1611 union 768 unione 768 unistd.h 979 1329 unità di traduzione 619 672 unixODBC 2121 unlink() 1032 1331 1340 Unproto 815 unsigned 571 2385 UPDATE 1918 USAGE 1563 USHRT_MAX 1071 1360 VALUE 1570 VALUE OF 1547 va_arg 746 va_arg() 1123 1359 va_copy() 1123 1359 va_end 746 va_end() 1123 1359 va_list 746 784 1123 1359 va_start 746 va_start() 1123 1359 VEOF 1347 VEOL 1347 VERASE 1347 vettore 110 2351 vfprintf() 1279 1421 vfscanf() 1288 1428 VINTR 1347 virgola mobile 140 2321 VKILL 1347 VMIN 1347 void 586 608 779 2402 2441 volatile 584 vprintf() 860 1279 1421 VQUIT 1347 vscanf() 869 1288 1428 vsnprintf() 1279 1421 vsprintf() 1279 1421 vsscanf() 1288 1428 VSTART 1347 VSTOP 1347 VSUSP 1347 VTIME 1347 wait() 945 WCHAR_MAX 1086 1361 WCHAR_MIN 1086 1361 wchar_t 786 792 1086 1169 1361 wcstoimax() 1166 wcstombs() 1153 1379 wcstouimax() 1166 wctomb() 1151 1379 WEOF 828 while 602 2433 WINT_MAX 1086 1361 WINT_MIN 1086 1361 wint_t 787 1086 1361 *word* 183 WORD_BIT 1075 WORKING-STORAGE SECTION 1555 WRITE 1709 write() 987 1331 WWW-SQL 2041 W_OK 1330

x86 244 x86-32 179 XCHG 187 xor 1167 XOR 193 XOR 170 2345 xor_eq 1167 X_OK 1330 *zero* 160 183 *zero terminated string* 120 2360 zombie 948 # 564 2375 #define 623 #define() 627 #define()...# 627 #define()...## 627 #define()...__VA_ARGS__ 627 #define...## 623 #elif 634 #else 634 #endif 634 #error 645 #if 634 #ifdef 636 #ifndef 636 #if !defined 636 #if defined 636 #include 621 #line 640 #pragma 649 #undef 640 & 586 592 679 2402 2414 2478 &* 709 &= 586 592 2402 2414 && 586 590 2402 2413 ^ 586 592 2402 2414 ^= 586 592 2402 2414 ~ 586 592 2402 2414 ~= 586 592 2402 2414 \... 575 2394 \0 575 2394 \? 575 2394 \a 575 2394 \b 575 2394 \f 575 2394 \n 575 2394 \r 575 2394 \t 575 2394 \v 575 2394 \x... 575 2394 \" 575 2394 \\ 575 2394 \' 575 2394 | 586 592 2402 2414 |= 586 592 2402 2414 || 586 590 2402 2413 {...} 564 2375 $PGDATA 1951 1954 $PGHOST 1973 $PGPORT 1973 $PGTZ 2026 _Bool 778 _Exit() 1141 1379 _exit() 1331 _IOFBF 1256 1406 _IOLBF 1256 1406 _IONBF 1256 1406 _POSIX2_... 1075 _POSIX_... 1075 _Pragma 649 _PROTOTYPE 810 _XOPEN_... 1075 __bool_true_false_are_defined 1168 __DATE__ 646 __FILE__ 646 __func__ 754 __LINE__ 646 __STDC_HOSTED__ 646 __STDC_IEC_559__ 646 __STDC_IEC_COMPLEX__ 646 __STDC_ISO_10646__ 646 __STDC_VERSION__ 646 __STDC__ 646 __TIME__ 646 __udivdi3() 1069 __umoddi3() 1069 __VA_ARGS__ 627 '...' 575 2391 , 596 2420 - 586 588 2402 2406 -= 586 588 2402 2406 -- 586 588 2402 2406 -> 761 < 586 590 2402 2410 <= 586 590 2402 2410 << 586 592 2402 2414 <<= 586 592 2402 2414 > 586 590 2402 2410

>= 586 590 2402 2410 >> 586 592 2402 2414 >>= 586 592 2402 2414 % 586 588 2402 2406 %+... 853 1270 %...c 853 866 1270 %...d 853 866 1270 %...e 853 866 1270 %...f 853 866 1270 %...g 866 1270 %...hd 853 866 1270 %...hhd 866 1270 %...hhi 866 1270 %...hhn 1270 %...hho 866 1270 %...hhu 866 1270 %...hhx 866 1270 %...hi 853 866 1270 %...hn 1270 %...ho 853 866 1270 %...hu 853 866 1270 %...hx 853 866 1270 %...i 853 866 1270 %...lc 853 866 1270 %...ld 853 866 1270 %...Le 853 866 1270 %...Lf 853 866 1270 %...Lg 866 1270 %...li 866 1270 %...lld 853 866 1270 %...lli 866 1270 %...lln 1270 %...llo 853 866 1270 %...llu 853 866 1270 %...llx 853 866 1270 %...ln 1270 %...lo 853 866 1270 %...ls 853 866 1270 %...lu 853 866 1270 %...lx 853 866 1270 %...n 1270 %...o 853 866 1270 %...s 853 866 1270 %...u 853 866 1270 %...x 853 866 1270 %0... 853 1270 %= 586 588 2402 2406 %-... 853 1270