Capitolo 81

Nozioni minime sul linguaggio C

«a2» 2013.11.11 --- Copyright © Daniele Giacomini -- appunti2@gmail.com http://informaticalibera.net

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.