66.1 Nozioni minime

Il linguaggio C è il fondamento dei sistemi Unix. Un minimo di conoscenza di questo linguaggio è importante per districarsi tra i programmi distribuiti in forma sorgente, pur senza volerli modificare.

Il linguaggio C richiede la presenza di un compilatore per generare un file eseguibile (o interpretabile) dal kernel. Se si dispone di un sistema GNU con i cosiddetti «strumenti di sviluppo», intendendo con questo ciò che serve a ricompilare il kernel, si dovrebbe disporre di tutto quello che è necessario per provare gli esempi di questi capitoli.

Il linguaggio C standard deve disporre di una libreria applicabile a ogni tipo di architettura e di sistema operativo; pertanto le funzionalità di tale libreria è molto limitata. In questi capitoli dedicati al linguaggio C, quando si vuole fare riferimento a funzioni che

sono definite al di fuori dello standard minimo, ciò viene annotato espressamente; in particolare, nel caso di estensioni che riguardano lo standard POSIX, può apparire anche una nota a margine simbolica.

66.1.1 Struttura fondamentale

Il contenuto di un sorgente in linguaggio C può essere suddiviso in tre parti: commenti, direttive del precompilatore e istruzioni C. I commenti vanno aperti e chiusi attraverso l'uso dei simboli '**/***' e '***/**'; se poi il compilatore è conforme a standard più recenti, è ammissibile anche l'uso di '**//**' per introdurre un commento che termina alla fine della riga.

```
/* Questo è un commento che continua
   su più righe e finisce qui. */

// Qui inizia un altro commento che termina alla fine della
// riga; pertanto, per ogni riga va ripetuta la sequenza
// "//" di apertura.
```

Le direttive del precompilatore rappresentano un linguaggio che guida alla compilazione del codice vero e proprio. L'uso più comune di queste direttive viene fatto per includere porzioni di codice sorgente esterne al file. È importante fare attenzione a non confondersi, dal momento che tali istruzioni iniziano con il simbolo '#': non si tratta di commenti.

Il programma C tipico richiede l'inclusione di codice esterno composto da file che terminano con l'estensione '.h'. La porzione di libreria che viene utilizzata più frequentemente è quella necessaria alla gestione dei flussi di standard input, standard output e standard error; si dichiara implicitamente il suo utilizzo includendo il file di intestazione 'stdio.h' nel modo seguente:

```
#include <stdio.h>
```

Le istruzioni C terminano con un punto e virgola ('**;**') e i raggruppamenti di queste (noti come «istruzioni composte») si fanno utilizzando le parentesi graffe ('**{ }**').[1]

***istruzione*;**

{*istruzione*; *istruzione*; *istruzione*;}

Generalmente, un'istruzione può essere interrotta e ripresa nella riga successiva, dal momento che la sua conclusione è dichiarata chiaramente dal punto e virgola finale. L'istruzione nulla viene rappresentata utilizzando un punto e virgola da solo.

I nomi scelti per identificare ciò che si utilizza all'interno del programma devono seguire regole determinate, definite dal compilatore C a disposizione. Ma per cercare di scrivere codice portabile in altre piattaforme, conviene evitare di sfruttare caratteristiche speciali del proprio ambiente. In particolare:

- un nome può iniziare con una lettera alfabetica e continuare con altre lettere, cifre numeriche e il trattino basso;
- i nomi potrebbero iniziare anche con il trattino basso, ma ciò è preferibile evitarlo, se non ci sono motivi validi per questo;[2]
- nei nomi si distinguono le lettere minuscole da quelle maiuscole (pertanto, '**Nome**' è diverso da '**nome**' e da tante altre combinazioni di minuscole e maiuscole).

La lunghezza dei nomi può essere un elemento critico; generalmente la dimensione massima dovrebbe essere di 32 caratteri, ma ci sono versioni di C che ne possono accettare solo una quantità inferiore. In particolare, il compilatore GNU ne accetta molti di più di 32. In ogni caso, il compilatore non rifiuta i nomi troppo lunghi, semplicemente non ne distingue più la differenza oltre un certo punto.

Il codice di un programma C è scomposto in funzioni, dove normalmente l'esecuzione del programma corrisponde alla chiamata della funzione ***main()***. Questa funzione può essere dichiarata senza parametri, '**int main (void)**', oppure con due parametri precisi: '**int main (int argc, char *argv[])**'.[3]

66.1.2 Ciao mondo!

Come sempre, il modo migliore per introdurre a un linguaggio di programmazione è di proporre un esempio banale, ma funzionante. Al solito si tratta del programma che emette un messaggio e poi termina la sua esecuzione.

```
/*
 *      Ciao mondo!
 */

#include <stdio.h>

/* La funzione main() viene eseguita automaticamente
   all'avvio. */
int main (void)
{
  /* Si limita a emettere un messaggio. */
  printf ("Ciao mondo!\n");
}
```

Nel programma sono state inserite alcune righe di commento. In particolare, all'inizio, l'asterisco che si trova nella seconda riga ha soltanto un significato estetico, per guidare la vista verso la conclusione del commento stesso.

Il programma si limita a emettere la stringa «Ciao Mondo!» seguita da un codice di interruzione di riga, rappresentato dal simbolo '**\n**'.

66.1.2.1 Compilazione

Per compilare un programma scritto in C si utilizza generalmente il comando '**cc**', anche se di solito si tratta di un collegamento simbolico al vero compilatore che si ha a disposizione. Supponendo di avere salvato il file dell'esempio con il nome 'ciao.c', il comando per la sua compilazione è il seguente:

```
$ cc ciao.c [Invio]
```

Quello che si ottiene è il file 'a.out' che dovrebbe già avere i permessi di esecuzione.

```
$ ./a.out [Invio]

  Ciao mondo!
```

Se si desidera compilare il programma definendo un nome diverso per il codice eseguibile finale, si può utilizzare l'opzione standard '**-o**'.

```
$ cc -o ciao ciao.c [Invio]
```

Con questo comando, si ottiene l'eseguibile '**ciao**'.

```
$ ./ciao [Invio]

  Ciao mondo!
```

In generale, se ciò è possibile, conviene chiedere al compilatore di mostrare gli avvertimenti (*warning*), senza limitarsi ai soli errori. Pertanto, nel caso il compilatore sia GNU C, è bene usare l'opzione '**-Wall**':

```
$ cc -Wall -o ciao ciao.c [Invio]
```

66.1.2.2 Emissione dati attraverso «printf()»

L'esempio di programma presentato sopra si avvale della funzione ***printf()***[4] per emettere il messaggio attraverso lo standard output. Questa funzione è più sofisticata di quanto possa apparire dall'esempio, in quanto permette di comporre il risultato da emettere. Negli esempi più semplici di codice C appare immancabilmente questa funzione, per cui è necessario descrivere subito, almeno in parte, il suo funzionamento.

```
int printf (stringa_di_formato [, espressione]…);
```

La funzione ***printf()*** emette attraverso lo standard output la stringa che costituisce il primo parametro, dopo averla rielaborata in base alla presenza di ***specificatori di conversione*** riferiti alle eventuali espressioni che compongono gli argomenti successivi; inoltre restituisce il numero di caratteri emessi.

L'utilizzo più semplice di ***printf()*** è quello che è già stato visto, cioè l'emissione di una stringa senza specificatori di conversione (il codice '**\n**' rappresenta un carattere preciso e non è uno specificatore, piuttosto si tratta di una cosiddetta sequenza di escape).

```
printf ("Ciao mondo!\n");
```

La stringa può contenere degli specificatori di conversione del tipo '**%i**', '**%c**', '**%f**',... e questi fanno ordinatamente riferimento agli argomenti successivi. L'esempio seguente fa in modo che la stringa incorpori il valore del secondo argomento nella posizione in cui appare '**%i**':

```
printf ("Totale fatturato: %i\n", 12345);
```

Lo specificatore di conversione '**%i**' stabilisce anche che il valore in questione deve essere trasformato secondo una rappresentazione decimale intera. Per cui, il risultato diviene esattamente quello che ci si aspetta.

```
Totale fatturato: 12345
```

66.1.3 Variabili e tipi

I tipi di dati elementari gestiti dal linguaggio C dipendono dall'architettura dell'elaboratore sottostante. In questo senso, volendo fare un discorso generale, è difficile definire la dimensione delle variabili numeriche; si possono dare solo delle definizioni relative. Solitamente, il riferimento è dato dal tipo numerico intero ('**int**') la cui dimensione in bit corrisponde a quella della ***parola***, ovvero dalla capacità dell'unità aritmetico-logica del microprocessore, oppure a qualunque altra entità che il microprocessore sia in grado di gestire con la massima efficienza. In pratica, con l'architettura x86 a 32 bit, la dimensione di un intero normale è di 32 bit, ma rimane la stessa anche con l'architettura x86 a 64 bit.

I documenti che descrivono lo standard del linguaggio C, definiscono la «dimensione» di una variabile come ***rango*** (*rank*).

66.1.3.1 Bit, byte e caratteri

A proposito della gestione delle variabili, esistono pochi concetti che sembrano rimanere stabili nel tempo. Il riferimento più importante in assoluto è il byte, che per il linguaggio C è almeno di 8 bit, ma potrebbe essere più grande.[5] Dal punto di vista del linguaggio C, il byte è l'elemento più piccolo che si possa indirizzare nella memoria centrale, questo anche quando la memoria fosse organizzata effettivamente a parole di dimensione maggiore del byte. Per esempio, in un elaboratore che suddivide la memoria in blocchi da 36 bit, si potrebbero avere byte da 9, 12, 18 bit o addirittura 36 bit.[6]

Una volta definito il byte, si considera che il linguaggio C rappresenti ogni variabile scalare come una sequenza continua di byte; pertanto, tutte le variabili scalari sono rappresentate come multipli di byte; di conseguenza anche le variabili strutturate lo sono, con la differenza che in tal caso potrebbero inserirsi dei «buchi» (in byte), dovuti alla necessità di allineare i dati in qualche modo.

Il tipo '**char**' (carattere), indifferentemente se si considera o meno il segno, rappresenta tradizionalmente una variabile numerica che occupa esattamente un byte, pertanto, spesso si confondono i termini «carattere» e «byte», nei documenti che descrivono il linguaggio C.

A causa della capacità limitata che può avere una variabile di tipo '**char**', il linguaggio C distingue tra un insieme di caratteri «minimo» e un insieme «esteso», da rappresentare però in altra forma.

66.1.3.2 Tipi primitivi

I tipi di dati primitivi rappresentano un valore **numerico** singolo, nel senso che anche il tipo '**char**' viene trattato come un numero. Il loro elenco essenziale si trova nella tabella 66.9.

Tabella 66.9. Elenco dei tipi comuni di dati primitivi elementari in C.

Tipo	Descrizione
`char`	Carattere (generalmente di 8 bit).
`int`	Intero normale.
`float`	Virgola mobile a precisione singola.
`double`	Virgola mobile a precisione doppia.

Come già accennato, non si può stabilire in modo generale quali siano le dimensioni esatte in bit dei vari tipi di dati, ovvero il rango, in quanto l'elemento certo è solo la relazione tra loro.

$$\texttt{char} \leq \texttt{int} \leq \texttt{float} \leq \texttt{double}$$

Questi tipi primitivi possono essere estesi attraverso l'uso di alcuni qualificatori: '**short**', '**long**', '**long long**', '**signed**'[7] e '**unsigned**'.[8] I primi tre si riferiscono al rango, mentre gli altri modificano il modo di valutare il contenuto di alcune variabili. La tabella 66.11 riassume i vari tipi primitivi con le combinazioni ammissibili dei qualificatori.

Tabella 66.11. Elenco dei tipi comuni di dati primitivi in C assieme ai qualificatori usuali.

Tipo	Abbreviazione	Descrizione
`char`		Tipo '**char**' per il quale non conta sapere se il segno viene considerato o meno.
`signed char`		Tipo '**char**' usato numericamente con segno.
`unsigned char`		Tipo '**char**' usato numericamente senza segno.
`short int` `signed short int`	`short` `signed short`	Intero più breve di '**int**', con segno.
`unsigned short int`	`unsigned short`	Tipo '**short**' senza segno.
`int` `signed int`		Intero normale, con segno.
`unsigned int`	`unsigned`	Tipo '**int**' senza segno.
`long int` `signed long int`	`long` `signed long`	Intero più lungo di '**int**', con segno.
`unsigned long int`	`unsigned long`	Tipo '**long**' senza segno.
`long long int` `signed long long int`	`long long` `signed long long`	Intero più lungo di '**long int**', con segno.
`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:

$$\texttt{char} \leq \texttt{short} \leq \texttt{int} \leq \texttt{long} \leq \texttt{long long}$$

$$\texttt{float} \leq \texttt{double} \leq \texttt{long double}$$

I tipi '**long**' e '**float**' potrebbero avere un rango uguale, altrimenti

non è detto quale dei due sia più grande.

Il programma seguente, potrebbe essere utile per determinare il rango dei vari tipi primitivi nella propria piattaforma.[9]

Listato 66.13. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/NxyS6KVy* , *http://ideone.com/uSVC3* .

```
#include <stdio.h>

int main (void)
{
    printf ("char          %i\n", (int) sizeof (char));
    printf ("short int     %i\n", (int) sizeof (short int));
    printf ("int           %i\n", (int) sizeof (int));
    printf ("long int      %i\n", (int) sizeof (long int));
    printf ("long long int %i\n", (int) sizeof (long long int));
    printf ("float         %i\n", (int) sizeof (float));
    printf ("double        %i\n", (int) sizeof (double));
    printf ("long double   %i\n", (int) sizeof (long double));
    return 0;
}
```

Il risultato potrebbe essere simile a quello seguente:

```
char          1
short int     2
int           4
long int      4
long long int 8
float         4
double        8
long double   12
```

I numeri rappresentano la quantità di caratteri, nel senso di valori '**char**', per cui il tipo '**char**' dovrebbe sempre avere una dimensione unitaria.[10]

I tipi primitivi di variabili mostrati sono tutti utili alla memorizzazione di valori numerici, a vario titolo. A seconda che il valore in questione sia trattato con segno o senza segno, varia lo spettro di valori che possono essere contenuti.

Nel caso di interi ('**char**', '**short**', '**int**', '**long**' e '**long long**'), la variabile può essere utilizzata per tutta la sua estensione a contenere un numero binario. Pertanto, quando la rappresentazione è senza segno, il massimo valore ottenibile è $(2^{n})-1$, dove ***n*** rappresenta il numero di bit a disposizione. Quando invece si vuole trattare il dato come un numero con segno, il valore numerico massimo ottenibile è circa la metà (se si usa la rappresentazione dei valori negativi in complemento a due, l'intervallo di valori va da $(2^{n-1})-1$ a $-(2^{n-1})$)

Nel caso di variabili a virgola mobile non c'è più la possibilità di rappresentare esclusivamente valori senza segno; inoltre, più che esserci un limite nella grandezza rappresentabile (che comunque esiste), c'è soprattutto un limite nel grado di approssimazione.

Le variabili '**char**' sono fatte, in linea di principio, per contenere il codice di rappresentazione di un carattere, secondo la codifica utilizzata nel sistema. Generalmente si tratta di un dato di 8 bit, ma non è detto che debba sempre essere così. A ogni modo, il fatto che questa variabile possa essere gestita in modo numerico, permette una facile conversione da lettera a codice numerico corrispondente.

Un tipo di valore che non è stato ancora visto è quello logico: *Vero* è rappresentato da un qualsiasi valore numerico intero diverso da zero, mentre *Falso* corrisponde a zero.

66.1.3.3 Costanti letterali comuni

Quasi tutti i tipi di dati primitivi hanno la possibilità di essere rappresentati in forma di costante letterale. In particolare, si distingue tra:

- costanti carattere, rappresentate da un carattere alfanumerico racchiuso tra apici singoli, come '**'A'**', '**'B'**',...;
- costanti intere, rappresentate da un numero senza decimali, e a seconda delle dimensioni può trattarsi di uno dei vari tipi di interi (escluso '**char**');
- costanti con virgola, rappresentate da un numero con decimali (un punto seguito da altre cifre, anche se si tratta solo di zeri) che, indipendentemente dalle dimensioni, di norma sono di tipo '**double**'.

Per esempio, 123 è generalmente una costante '**int**', mentre 123.0 è una costante '**double**'.

Le costanti che esprimono valori interi possono essere rappresentate con diverse basi di numerazione, attraverso l'indicazione di un prefisso: '**0*n***', dove ***n*** contiene esclusivamente cifre da zero a sette, viene inteso come un numero in base otto; '**0x*n***' o '**0X*n***', dove ***n*** può contenere le cifre numeriche consuete, oltre alle lettere da «A» a «F» (minuscole o maiuscole, indifferentemente) viene trattato come un numero in base sedici; negli altri casi, un numero composto con cifre da zero a nove è interpretato in base dieci.

Per quanto riguarda le costanti che rappresentano numeri con virgola, oltre alla notazione '***intero*.*decimali***' si può usare la «notazione scientifica». Per esempio, '**7e+15**' rappresenta l'equivalente di $7\cdot(10^{15})$, cioè un sette con 15 zeri. Nello stesso modo, '**7e-5**', rappresenta l'equivalente di $7\cdot(10^{-5})$, cioè 0,00007.

Il tipo di rappresentazione delle costanti numeriche, intere o con virgola, può essere specificato aggiungendo un suffisso, costituito da una o più lettere, come si vede nelle tabelle successive. Per esempio, '**123UL**' è un numero di tipo '**unsigned long int**', mentre '**123.0F**' è un tipo '**float**'. Si osservi che il suffisso può essere composto, indifferentemente, con lettere minuscole o maiuscole.

Tabella 66.15. Suffissi per le costanti che esprimono valori interi.

Suffisso	Descrizione
assente	In tal caso si tratta di un intero «normale» o più grande, se necessario.
U	Tipo senza segno ('**unsigned**').
L	Intero più grande della dimensione normale ('**long**').
LL	Intero molto più grande della dimensione normale ('**long long**').
UL	Intero senza segno, più grande della dimensione normale ('**unsigned long**').
ULL	Intero senza segno, molto più grande della dimensione normale ('**unsigned long long**').

Tabella 66.16. Suffissi per le costanti che esprimono valori con virgola.

Suffisso	Descrizione
assente	Tipo '**double**'.
F	Tipo '**float**'.
L	Tipo '**long double**'.

È possibile rappresentare anche le stringhe in forma di costante attraverso l'uso degli apici doppi, ma la stringa non è un tipo di dati primitivo, trattandosi piuttosto di un array di caratteri. Per il momento è importante fare attenzione a non confondere il tipo '**char**' con la stringa. Per esempio, '**'F'**' è un carattere (con un proprio valore numerico), mentre '**"F"**' è una stringa, ma la differenza tra i due è notevole. Le stringhe vengono descritte nella sezione 66.5.

I caratteri privi di rappresentazione grafica possono essere indicati, principalmente, attraverso tre tipi di notazione: ottale, esadecimale e simbolica. In tutti i casi si utilizza la barra obliqua inversa ('\') come carattere di escape, cioè come simbolo per annunciare che ciò che segue immediatamente deve essere interpretato in modo particolare.

La notazione ottale usa la forma '***ooo***', dove ogni lettera ***o*** rappresenta una cifra ottale. A questo proposito, è opportuno notare che

se la dimensione di un carattere fosse superiore ai fatidici 8 bit, occorrerebbero probabilmente più cifre (una cifra ottale rappresenta un gruppo di 3 bit).

La notazione esadecimale usa la forma '**\xhh**', dove ***h*** rappresenta una cifra esadecimale. Anche in questo caso vale la considerazione per cui ci vogliono più di due cifre esadecimali per rappresentare un carattere più lungo di 8 bit.

Dovrebbe essere logico, ma è il caso di osservare che la corrispondenza dei caratteri con i rispettivi codici numerici dipende dalla codifica utilizzata. Generalmente si utilizza la codifica ASCII, riportata anche nella sezione 47.7.5 (in questa fase introduttiva si omette di trattare la rappresentazione dell'insieme di caratteri universale).

La notazione simbolica permette di fare riferimento facilmente a codici di uso comune, quali <*CR*>, <*HT*>,... Inoltre, questa notazione permette anche di indicare caratteri che altrimenti verrebbero interpretati in maniera differente dal compilatore. La tabella 66.17 riporta i vari tipi di rappresentazione delle costanti carattere attraverso codici di escape.

Tabella 66.17. Elenco dei modi di rappresentazione delle costanti carattere attraverso codici di escape.

Codice	ASCII	Altra codifica
\ooo	Notazione ottale in base alla codifica.	idem
\xhh	Notazione esadecimale in base alla codifica.	idem
\\	Una singola barra obliqua inversa ('\').	idem
\'	Un apice singolo destro.	idem
\"	Un apice doppio.	idem
\?	Un punto interrogativo (per impedire che venga inteso come parte di una sequenza triplice, o *trigraph*).	idem
\0	Il codice <*NUL*>.	Il carattere nullo (con tutti i bit a zero).
\a	Il codice <*BEL*> (*bell*).	Il codice che, rappresentato sullo schermo o sulla stampante, produce un segnale acustico (*alert*).
\b	Il codice <*BS*> (*backspace*).	Il codice che fa arretrare il cursore di una posizione nella riga (*backspace*).
\f	Il codice <*FF*> (*form feed*).	Il codice che fa avanzare il cursore all'inizio della prossima pagina logica (*form feed*).
\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 '**\xhh**', le altre sequenze esprimono un concetto, piuttosto di un codice numerico preciso. All'origine del linguaggio C, tutte le altre sequenze corrispondono a un solo carattere non stampabile, ma attualmente non è più garantito che sia così. In particolare, la sequenza '**\n**', nota come *new-line*, potrebbe essere espressa in modo molto diverso rispetto al codice <*LF*> tradizionale. Questo concetto viene comunque approfondito a proposito della gestione dei flussi di file.

In varie situazioni, il linguaggio C standard ammette l'uso di sequenze composte da due o tre caratteri, note come *digraph* e *trigraph* rispettivamente; ciò in sostituzione di simboli la cui rappresentazione, in quel contesto, può essere impossibile. In un sistema che ammetta almeno l'uso della codifica ASCII per scrivere il file sorgente, con l'ausilio di una tastiera comune, non c'è alcun bisogno di usare tali artifici, i quali, se usati, renderebbero estremamente complessa la lettura del sorgente. Pertanto, è bene sapere che esistono queste cose, ma è meglio non usarle mai. Tuttavia, siccome le sequenze a tre caratteri (*trigraph*) iniziano con una coppia di punti interrogativi, se in una stringa si vuole rappresentare una sequenza del genere, per evitare che il compilatore la traduca diversamente, è bene usare la sequenza '**\?\?**', come suggerisce la tabella 66.17.

Nell'esempio introduttivo appare già la notazione '**\n**' per rappresentare l'inserzione di un codice di interruzione di riga alla fine del messaggio di saluto:

```
…
  printf ("Ciao mondo!\n");
…
```

Senza di questo, il cursore resterebbe a destra del messaggio alla fine dell'esecuzione di quel programma, ponendo lì l'invito.

66.1.3.4 Valore numerico delle costanti carattere

Il linguaggio C distingue tra i caratteri di un insieme fondamentale e ridotto, da quelli dell'insieme di caratteri universale (ISO 10646). Il gruppo di caratteri ridotto deve essere rappresentabile in una 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 66.19. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/yrc2S7Xv* , *http://ideone.com/HCvsD* .

```
#include <stdio.h>

int main (void)
{
  char c;
  for (c = '0'; c <= 'Z'; c++)
    {
      printf ("%c", c);
    }
  printf ("\n");
  return 0;
}
```

Il programma di esempio che si vede nel listato appena mostrato, se prodotto per un ambiente in cui si utilizza la codifica ASCII, genera il risultato seguente:

```
0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

66.1.3.5 Campo di azione delle variabili

Il campo di azione delle variabili in C viene determinato dalla posizione in cui queste vengono dichiarate e dall'uso di qualificatori particolari. Nella fase iniziale dello studio del linguaggio basta considerare, approssimativamente, che quanto dichiarato all'interno di una funzione ha valore locale per la funzione stessa, mentre quanto dichiarato al di fuori, ha valore globale per tutto il file. Pertanto, in

questo capitolo si usano genericamente le definizioni di «variabile locale» e «variabile globale», senza affrontare altre questioni. Nella sezione 66.3 viene trattato questo argomento con maggiore dettaglio.

66.1.3.6 Dichiarazione delle variabili

La dichiarazione di una variabile avviene specificando il tipo e il nome della variabile, come nell'esempio seguente dove viene creata la variabile ***numero*** di tipo intero:

```
int numero;
```

La variabile può anche essere inizializzata contestualmente, assegnandole un valore, come nell'esempio seguente in cui viene dichiarata la stessa variabile ***numero*** con il valore iniziale di 1000:

```
int numero = 1000;
```

Una costante è qualcosa che non varia e generalmente si rappresenta attraverso una notazione che ne definisce il valore, ovvero attraverso una costante letterale. Tuttavia, a volte può essere più comodo definire una costante in modo simbolico, come se fosse una variabile, per facilitarne l'utilizzo e la sua identificazione all'interno del programma. Si ottiene questo con il modificatore '**`const`**'. Ovviamente, è obbligatorio inizializzala contestualmente alla sua dichiarazione. L'esempio seguente dichiara la costante simbolica ***pi*** con il valore del π:

```
const float pi = 3.14159265;
```

Le costanti simboliche di questo tipo, sono delle variabili per le quali il compilatore non concede che avvengano delle modifiche; pertanto, il programma eseguibile che si ottiene potrebbe essere organizzato in modo tale da caricare questi dati in segmenti di memoria a cui viene lasciato poi il solo permesso di lettura.

Tradizionalmente, l'uso di costanti simboliche di questo tipo è stato limitato, preferendo delle ***macro-variabili*** definite e gestite attraverso il precompilatore (come viene descritto nella sezione 66.2). Tuttavia, un compilatore ottimizzato è in grado di gestire al meglio le costanti definite nel modo illustrato dall'esempio, utilizzando anche dei valori costanti letterali nella trasformazione in linguaggio assemblatore, rendendo così indifferente, dal punto di vista del risultato, l'alternativa delle macro-variabili. Pertanto, la stessa guida *GNU coding standards* chiede di definire le costanti come variabili-costanti, attraverso il modificatore '**`const`**'.

66.1.3.7 Variabili costanti e variabili volatili

Come già descritto nella sezione precedente, una variabile può essere dichiarata con il modificatore '**`const`**' per sottolineare al compilatore che non deve essere modificata nel corso del programma, salva la possibilità di inizializzarla contestualmente alla sua dichiarazione.

```
const float pi = 3.14159265;
```

All'opposto della costante si può considerare un'area di memoria a cui accedono programmi differenti, in modo asincrono, ognuno con la facoltà di modificarla a proprio piacimento, oppure un'area che viene modificata direttamente dall'hardware. In questi casi, ovvero quando il compilatore non deve attuare delle semplificazioni che partano dalla presunzione del contenuto di una certa variabile, si usa il modificatore '**`volatile`**'. Si osservi l'esempio seguente:

```
…
volatile int i;
…
i = 1;
if (i > 0)
  {
    …
  }
else
  {
    …
  }
…
```


Anche se alla variabile ***i*** viene assegnato il valore uno, il compilatore non può escludere che nel momento della verifica della variabile questa abbia invece un valore differente. In altri termini, se la variabile ***i*** venisse dichiarata in modo normale, un compilatore ottimizzato potrebbe escludere le istruzioni sotto il controllo della parola chiave '**`else`**'.

Quando l'area di memoria che viene considerata «volatile», deve essere modificata da un processo estraneo, mentre il programma si limita semplicemente a leggerne il contenuto prendendo atto del valore che ha, la variabile può essere dichiarata simultaneamente con i modificatori '**`const`**' e '**`volatile`**', come nell'esempio seguente, dove, tra l'altro, si presume che la variabile in questione sia definita in un altro file-oggetto:

```
extern const volatile int variabile;
…
```

66.1.3.8 Il tipo indefinito: «void»

Lo standard del linguaggio C definisce un tipo particolare di variabile, individuato dalla parola chiave '**`void`**': si tratta di un tipo che rappresenta una variabile di rango nullo; la quale, come tale, non può contenere alcun valore.

66.1.4 Operatori ed espressioni

L'operatore è qualcosa che esegue un qualche tipo di funzione, su uno o più operandi, restituendo un valore.[11] Il valore restituito è di tipo diverso a seconda degli operandi utilizzati. Per esempio, la somma di due interi genera un risultato intero. Gli operandi descritti di seguito sono quelli più comuni e importanti.

Le espressioni sono formate spesso dalla valutazione di sottoespressioni (espressioni più piccole). Va osservato che ci sono circostanze in cui il contesto non impone che ci sia un solo ordine possibile nella valutazione delle sottoespressioni, ma il programmatore deve tenere conto di questa possibilità, per evitare che il risultato dipenda dalle scelte non prevedibili del compilatore.

Tabella 66.27. Ordine di precedenza tra gli operatori previsti nel linguaggio C. Gli operatori sono raggruppati a livelli di priorità equivalente, partendo dall'alto con la priorità maggiore, scendendo progressivamente alla priorità minore. Le variabili ***a***, ***b*** e ***c*** rappresentano la collocazione delle sottoespressioni da considerare ed esprimono l'ordine di associatività: prima ***a***, poi ***b***, poi ***c***.

Operatori	Annotazioni
`(a)` `[a]` `a->b` `a.b`	Le parentesi tonde usate per raggruppare una porzione di espressione hanno la precedenza su ogni altro operatore. Le parentesi quadre riguardano gli array; gli operatori '`->`' e '`.`', riguardano le strutture e le unioni.
`!a` `~a` `++a` `--a` `+a` `-a` `*a` `&a` `(tipo)` `sizeof a`	Gli operatori '`+`' e '`-`' di questo livello sono da intendersi come «unari», ovvero si riferiscono al segno di quanto appare alla loro destra. Gli operatori '`*`' e '`&`' di questo livello riguardano la gestione dei puntatori; le parentesi tonde si riferiscono al cast.
`a*b` `a/b` `a%b`	Moltiplicazione, divisione e resto della divisione intera.
`a+b` `a-b`	Somma e sottrazione.
`a<<b` `a>>b`	Scorrimento binario.
`a<b` `a<=b` `a>b` `a=>b`	Confronto.
`a==b` `a!=b`	Confronto.
`a&b`	AND bit per bit.
`a^b`	XOR bit per bit.

Operatori	Annotazioni
`a|b`	OR bit per bit.
`a&&b`	AND nelle espressioni logiche.
`a||b`	OR nelle espressioni logiche.
`c?b:a`	Operatore condizionale
`b=a` `b+=a` `b-=a` `b*=a` `b/=a` `b%=a` `b&=a` `b^=a` `b|=a` `b<<=a` `b>>=a`	Operatori di assegnamento.
`a, b`	Sequenza di espressioni (espressione multipla).

66.1.4.1 Operatori aritmetici

Gli operatori che intervengono su valori numerici sono elencati nella tabella 66.28. Per dare un significato alle descrizioni della tabella, occorre tenere presenta una caratteristica importante del linguaggio, per la quale, la maggior parte delle espressioni restituisce un valore. Per esempio, '`b = a = 1`' fa sì che la variabile ***a*** ottenga il valore 1 e che, successivamente, la variabile ***b*** ottenga il valore di ***a***. In questo senso, al problema dell'ordine di precedenza dei vari operatori si aggiunge anche l'ordine in cui le espressioni restituiscono un 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 66.28. Elenco degli operatori aritmetici e di quelli di assegnamento relativi a valori numerici.

Operatore e operandi	Descrizione
`++op`	Incrementa di un'unità l'operando prima che venga restituito il suo valore.
`op++`	Incrementa di un'unità l'operando dopo averne restituito il suo valore.
`--op`	Decrementa di un'unità l'operando prima che venga restituito il suo valore.
`op--`	Decrementa di un'unità l'operando dopo averne restituito il suo valore.
`+op`	Non ha alcun effetto.
`-op`	Inverte il segno dell'operando (prima di restituirne il valore).
`op1 + op2`	Somma i due operandi.
`op1 - op2`	Sottrae dal primo il secondo operando.
`op1 * op2`	Moltiplica i due operandi.
`op1 / op2`	Divide il primo operando per il secondo.
`op1 % op2`	Calcola il resto della divisione tra il primo e il secondo operando, i quali devono essere costituiti da valori interi.
`var = valore`	Assegna alla variabile il valore alla destra.
`op1 += op2`	`op1 = (op1 + op2)`
`op1 -= op2`	`op1 = (op1 - op2)`
`op1 *= op2`	`op1 = (op1 * op2)`
`op1 /= op2`	`op1 = (op1 / op2)`
`op1 %= op2`	`op1 = (op1 % op2)`

66.1.4.2 Operatori di confronto e operatori logici

Gli operatori di confronto determinano la relazione tra due operandi. Il risultato dell'espressione composta da due operandi posti a confronto è un numero intero ('`int`') e precisamente si ottiene uno se il confronto è valido e zero in caso contrario. Gli operatori di confronto sono elencati nella tabella 66.29.

Il linguaggio C non ha una rappresentazione specifica per i valori booleani *Vero* e *Falso*,[12] ma si limita a interpretare un valore pari a zero come *Falso* e un valore diverso da zero come *Vero*. Va osservato, quindi, che il numero usato come valore booleano, può essere espresso anche in virgola mobile, benché sia preferibile di gran lunga un intero normale.

Tabella 66.29. Elenco degli operatori di confronto. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
`op1 == op2`	*Vero* se gli operandi si equivalgono.
`op1 != op2`	*Vero* se gli operandi sono differenti.
`op1 < op2`	*Vero* se il primo operando è minore del secondo.
`op1 > op2`	*Vero* se il primo operando è maggiore del secondo.
`op1 <= op2`	*Vero* se il primo operando è minore o uguale al secondo.
`op1 >= op2`	*Vero* se il primo operando è maggiore o uguale al secondo.

Quando si vogliono combinare assieme diverse espressioni logiche, comprendendo in queste anche delle variabili che contengono un valore booleano, si utilizzano gli operatori logici (noti normalmente come: AND, OR, NOT, ecc.). Il risultato di un'espressione logica complessa è quello dell'ultima espressione elementare valutata effettivamente, in quanto le sottoespressioni che non possono cambiare l'esito della condizione complessiva non vengono valutate. Gli operatori logici sono elencati nella tabella 66.30.

Tabella 66.30. Elenco degli operatori logici. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
`! op`	Inverte il risultato logico dell'operando.
`op1 && op2`	Se il risultato del primo operando è *Falso* non valuta il secondo.
`op1 || op2`	Se il risultato del primo operando è *Vero* non valuta il secondo.

Un tipo particolare di operatore logico è l'operatore condizionale, il quale permette di eseguire espressioni diverse in relazione al risultato di una condizione. La sua sintassi si esprime nel modo seguente:

```
condizione ? espressione1 : espressione2
```

In pratica, se l'espressione che rappresenta la condizione si avvera, viene eseguita la prima espressione che segue il punto interrogativo, altrimenti viene eseguita quella che segue i due punti.

66.1.4.3 Operatori binari

Il linguaggio C consente di eseguire alcune operazioni binarie, sui **valori interi**, come spesso è possibile fare con un linguaggio assemblatore, anche se non è possibile interrogare degli indicatori (*flag*) che informino sull'esito delle azioni eseguite. Sono disponibili le operazioni elencate nella tabella 66.31.

Tabella 66.31. Elenco degli operatori binari. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
op1 `&` *op2*	AND bit per bit.
op1 `|` *op2*	OR bit per bit.
op1 `^` *op2*	XOR bit per bit (OR esclusivo).
op1 `<<` *op2*	Scorrimento a sinistra di ***op2*** bit (con ***op2*** che rappresenta un valore positivo o senza segno). A destra vengono aggiunti bit a zero
op1 `>>` *op2*	Scorrimento a destra di ***op2*** bit (con ***op2*** che rappresenta un valore positivo o senza segno). Il valore dei bit aggiunti a sinistra potrebbe tenere conto del segno.
`~`*op1*	Complemento a uno.
op1 `&=` *op2*	*op1* `=` (*op1* `&` *op2*)
op1 `|=` *op2*	*op1* `=` (*op1* `|` *op2*)
op1 `^=` *op2*	*op1* `=` (*op1* `^` *op2*)
op1 `<<=` *op2*	*op1* `=` (*op1* `<<` *op2*)
op1 `>>=` *op2*	*op1* `=` (*op1* `>>` *op2*)
op1 `~=` *op2*	*op1* `=` `~`*op2*

A seconda del compilatore e della piattaforma, lo scorrimento a destra potrebbe essere di tipo aritmetico, ovvero potrebbe tenere conto del segno del valore che viene fatto scorrere. Pertanto, non potendo fare affidamento su questa ipotesi, è bene che i valori di cui si fa lo scorrimento a destra siano sempre senza segno, o comunque positivi.

Per aiutare a comprendere l'uso degli operatori binari vengono mostrati alcuni esempi. In particolare si utilizzano due operandi di tipo '`char`' (a 8 bit) senza segno: ***a*** contenente il valore 42, pari a 00101010_2; ***b*** contenente il valore 51, pari a 00110011_2.

`c = a & b`

00101010_2	(42_{10})	AND
00110011_2	(51_{10})	=
00100010_2	(34_{10})	

`c = a | b`

00101010_2	(42_{10})	OR
00110011_2	(51_{10})	=
00111011_2	(59_{10})	

`c = a ^ b`

00101010_2	(42_{10})	XOR
00110011_2	(51_{10})	=
00011001_2	(25_{10})	

Lo scorrimento, invece, viene mostrato sempre solo per una singola unità: ***a*** contenente il valore 42; ***b*** contenente il valore 1.

`c = a << b`

00101010_2	(42_{10})	<<
00000001_2	(1_{10})	=
01010100_2	(84_{10})	

`c = a >> b`

00101010_2	(42_{10})	>>
00000001_2	(1_{10})	=
00010101_2	(21_{10})	

`c = ~a`

00101010_2	(42_{10})	
11010101_2	(213_{10})	

66.1.4.4 Conversione di tipo

Quando si assegna un valore a una variabile, nella maggior parte dei casi, il contesto stabilisce il tipo di questo valore in modo corretto. Di fatto, è il tipo della variabile ricevente che stabilisce la conversione necessaria. Tuttavia, il problema si pone anche durante la valutazione di un'espressione.

Per esempio, '`5/4`' viene considerata la divisione di due interi e, di conseguenza, l'espressione restituisce un valore intero, cioè 1. Diverso sarebbe se si scrivesse '`5.0/4.0`', perché in questo caso si tratterebbe della divisione tra due numeri a virgola mobile (per la precisione, di tipo '`double`') e il risultato è un numero a virgola mobile.

Quando si pone il problema di risolvere l'ambiguità si utilizza esplicitamente la conversione del tipo, attraverso un ***cast***:

```
(tipo) espressione
```

In pratica, si deve indicare tra parentesi tonde il nome del tipo di dati in cui deve essere convertita l'espressione che segue. Il problema sta nella precedenza che ha il cast nell'insieme degli altri operatori e in generale conviene utilizzare altre parentesi per chiarire la relazione che ci deve essere.

```
int x = 10;
double y;
…
y = (double) x/9;
```

In questo caso, la variabile intera ***x*** viene convertita nel tipo '`double`' (a virgola mobile) prima di eseguire la divisione. Dal momento che il cast ha precedenza sull'operazione di divisione, non si pongono problemi, inoltre, la divisione avviene trasformando implicitamente il 9 intero in un 9,0 di tipo '`double`'. In pratica, l'operazione avviene utilizzando valori '`double`' e restituendo un risultato '`double`'.

66.1.4.5 Espressioni multiple

Un'istruzione, cioè qualcosa che termina con un punto e virgola, può contenere diverse espressioni separate da una virgola. Tenendo presente che in C l'assegnamento di una variabile è anche un'espressione, la quale restituisce il valore assegnato, si veda l'esempio seguente:

```
int x;
int y;
…
y = 10, x = 20, y = x*2;
```

L'esempio mostra un'istruzione contenente tre espressioni: la prima assegna a ***y*** il valore 10, la seconda assegna a ***x*** il valore 20 e la terza sovrascrive ***y*** assegnandole il risultato del prodotto $x \cdot 2$. 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 appena visto, gli si può apportare una piccola modifica per comprendere il concetto:

```
int x;
int y;
int z;
…
z = (y = 10, x = 20, y = x*2);
```

La variabile ***z*** si trova a ricevere il valore dell'espressione '***y = x*2***', perché è quella che viene eseguita per ultima nel gruppo raccolto tra parentesi.

A proposito di «espressioni multiple» vale la pena di ricordare ciò che accade con gli assegnamenti multipli, con l'esempio seguente:

```
y = x = 10;
```

Qui si vede l'assegnamento alla variabile ***y*** dello stesso valore che viene assegnato alla variabile ***x***. In pratica, sia ***x***, sia ***y***, contengono alla fine il numero 10, perché le precedenze sono tali che è come se fosse scritto: '***y = (x = 10)***'.

66.1.5 Strutture di controllo di flusso

Il linguaggio C gestisce praticamente tutte le strutture di controllo di flusso degli altri linguaggi di programmazione, compreso *go-to* che comunque è sempre meglio non utilizzare e qui, volutamente, non viene presentato.

Le strutture di controllo permettono di sottoporre l'esecuzione di una parte di codice alla verifica di una condizione, oppure permettono di eseguire dei cicli, sempre sotto il controllo di una condizione. La parte di codice che viene sottoposta a questo controllo, può essere una singola istruzione, oppure un gruppo di istruzioni (precisamente si chiamerebbe istruzione composta). Nel secondo caso, è necessario delimitare questo gruppo attraverso l'uso delle parentesi graffe.

Dal momento che è comunque consentito di realizzare un gruppo di istruzioni che in realtà ne contiene una sola, probabilmente è meglio utilizzare sempre le parentesi graffe, in modo da evitare equivoci nella lettura del codice. Dato che le parentesi graffe sono usate nel codice C, se queste appaiono nei modelli sintattici indicati, significa che fanno parte delle istruzioni e non della sintassi.

Negli esempi, i rientri delle parentesi graffe seguono le indicazioni della guida *GNU coding standards*.

66.1.5.1 Struttura condizionale: «if»

La struttura condizionale è il sistema di controllo fondamentale dell'andamento del flusso delle istruzioni.

```
if (condizione) istruzione
```

```
if (condizione) istruzione else istruzione
```

Se la condizione si verifica, viene eseguita l'istruzione o il gruppo di istruzioni che segue; quindi il controllo passa alle istruzioni successive alla struttura. Se viene utilizzata la sotto-struttura che si articola a partire dalla parola chiave '**`else`**', nel caso non si verifichi la condizione, viene eseguita l'istruzione che ne dipende. Sotto vengono mostrati alcuni esempi.

```
int i_importo;
…
if (i_importo > 10000000) printf ("L'offerta è vantaggiosa\n");
```

```
int i_importo;
int i_memorizza;
…
if (i_importo > 10000000)
  {
    i_memorizza = i_importo;
    printf ("L'offerta è vantaggiosa\n");
  }
else
  {
    printf ("Lascia perdere\n");
  }
```

L'esempio successivo, in particolare, mostra un modo grazioso per allineare le sottocondizioni, senza eccedere negli annidamenti:

```
int i_importo;
int i_memorizza;
…
if (i_importo > 10000000)
  {
    i_memorizza = i_importo;
    printf ("L'offerta è vantaggiosa\n");
  }
else if (i_importo > 5000000)
  {
    i_memorizza = i_importo;
    printf ("L'offerta è accettabile\n");
  }
else
  {
    printf ("Lascia perdere\n");
  }
```

66.1.5.2 Struttura di selezione: «switch»

La struttura di selezione che si attua con l'istruzione '**`switch`**', è un po' troppo complessa per essere rappresentata facilmente attraverso uno schema sintattico. In generale, questa struttura permette di **saltare** a una certa posizione della struttura, in base al risultato di un'espressione. L'esempio seguente mostra la visualizzazione del nome del mese, in base al valore di un intero.

```
int i_mese;
…
switch (i_mese)
  {
    case 1: printf ("gennaio\n"); break;
    case 2: printf ("febbraio\n"); break;
    case 3: printf ("marzo\n"); break;
    case 4: printf ("aprile\n"); break;
    case 5: printf ("maggio\n"); break;
    case 6: printf ("giugno\n"); break;
    case 7: printf ("luglio\n"); break;
    case 8: printf ("agosto\n"); break;
    case 9: printf ("settembre\n"); break;
    case 10: printf ("ottobre\n"); break;
    case 11: printf ("novembre\n"); break;
    case 12: printf ("dicembre\n"); break;
  }
```


Come si vede, dopo l'istruzione con cui si emette il nome del mese attraverso lo standard output, viene richiesta l'interruzione esplicita dell'analisi della struttura, attraverso l'istruzione '**`break`**', perché altrimenti verrebbero eseguite le istruzioni del caso successivo, se presente. Infatti, un gruppo di casi può essere raggruppato assieme, quando si vuole che ognuno di questi esegua lo stesso insieme di istruzioni.

```
int i_anno;
int i_mese;
int i_giorni;
…
switch (i_mese)
  {
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        i_giorni = 31;
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        i_giorni = 30;
        break;
    case 2:
        if (((i_anno % 4 == 0) && !(i_anno % 100 == 0)) ||
                (i_anno % 400 == 0))
            i_giorni = 29;
        else
            i_giorni = 28;
        break;
  }
```

È anche possibile dichiarare un caso predefinito che si verifichi quando nessuno degli altri si avvera.

```
int i_mese;
…
switch (i_mese)
  {
    case 1: printf ("gennaio\n"); break;
    case 2: printf ("febbraio\n"); break;
    …
    case 11: printf ("novembre\n"); break;
    case 12: printf ("dicembre\n"); break;
    default: printf ("mese non corretto\n"); break;
  }
```

La struttura di selezione che si ottiene con l'istruzione '**`switch`**' può apparire disarmonica rispetto all'organizzazione del linguaggio C, per la presenza delle voci '**`case`** ***valore*`:`**'. Queste voci sono sostanzialmente delle «etichette» che individuano una posizione nel codice, da raggiungere in base al valore preso in considerazione per la selezione.

66.1.5.3 Iterazione con condizione di uscita iniziale: «while»

L'iterazione si ottiene normalmente in C attraverso l'istruzione '**`while`**', la quale esegue un'istruzione, o un gruppo di queste, finché

la condizione continua a restituire il valore *Vero*. La condizione viene valutata prima di eseguire il gruppo di istruzioni e poi ogni volta che termina un ciclo, prima dell'esecuzione del successivo.

```
while (condizione) istruzione
```

L'esempio seguente fa apparire per 10 volte la lettera «x».

```
int i = 0;

while (i < 10)
  {
    i++;
    printf ("x");
  }
printf ("\n");
```

Nel blocco di istruzioni di un ciclo '**while**', ne possono apparire alcune particolari:

- '**break**', che serve a uscire definitivamente dalla struttura del ciclo;
- '**continue**', che serve a interrompere l'esecuzione del gruppo di istruzioni, riprendendo immediatamente con il ciclo successivo (a partire dalla valutazione della condizione).

L'esempio seguente è una variante del calcolo di visualizzazione mostrato sopra, modificato in modo da vedere il funzionamento dell'istruzione '**break**'. All'inizio della struttura, '**while (1)**' equivale a stabilire che il ciclo è senza fine, perché la condizione è sempre vera. In questo modo, solo la richiesta esplicita di interruzione dell'esecuzione della struttura (attraverso l'istruzione '**break**') permette l'uscita da questa.

```
int i = 0;

while (1)
  {
    if (i >= 10)
      {
        break;
      }
    i++;
    printf ("x");
  }
printf ("\n");
```

66.1.5.4 Iterazione con condizione di uscita finale: «do-while»

Una variante del ciclo '**while**', in cui l'analisi della condizione di uscita avviene dopo l'esecuzione del blocco di istruzioni che viene iterato, è definito dall'istruzione '**do**'.

```
do blocco_di_istruzioni while (condizione);
```

In questo caso, si esegue un gruppo di istruzioni una volta, poi se ne ripete l'esecuzione finché la condizione restituisce il valore *Vero*.

```
int i = 0;

do
  {
    i++;
    printf ("x");
  }
while (i < 10);
printf ("\n");
```

L'esempio mostrato è quello già usato nella sezione precedente, con l'adattamento necessario a utilizzare questa struttura di controllo.

La struttura di controllo '**do**…**while**' è in disuso, perché, generalmente, al suo posto si preferisce gestire i cicli di questo tipo attraverso una struttura '**while**', pura e semplice.

66.1.5.5 Ciclo enumerativo: «for»

In presenza di iterazioni in cui si deve incrementare o decrementare una variabile a ogni ciclo, si usa preferibilmente la struttura '**for**', che in C permetterebbe un utilizzo più ampio di quello comune:

```
for ([espressione1]; [espressione2]; [espressione3]) istruzione
```

La forma tipica di un'istruzione '**for**' è quella per cui la prima espressione corrisponde all'assegnamento iniziale di una variabile, la seconda a una condizione che deve verificarsi fino a che si vuole che sia eseguita l'istruzione (o il gruppo di istruzioni) e la terza all'incremento o decremento della variabile inizializzata con la prima espressione. In pratica, l'utilizzo normale del ciclo '**for**' potrebbe esprimersi nella sintassi seguente:

```
for (var = n; condizione; var++) istruzione
```

Il ciclo '**for**' potrebbe essere definito anche in maniera differente, più generale: la prima espressione viene eseguita una volta sola all'inizio del ciclo; la seconda viene valutata all'inizio di ogni ciclo e il gruppo di istruzioni viene eseguito solo se il risultato è *Vero*; l'ultima viene eseguita alla fine dell'esecuzione del gruppo di istruzioni, prima che si ricominci con l'analisi della condizione.

L'esempio già visto, in cui viene visualizzata per 10 volte una «x», potrebbe tradursi nel modo seguente, attraverso l'uso di un ciclo '**for**':

```
int i;

for (i = 0; i < 10; i++)
  {
    printf ("x");
  }
printf ("\n");
```

Anche nelle istruzioni controllate da un ciclo '**for**' si possono collocare istruzioni '**break**' e '**continue**', con lo stesso significato visto per il ciclo '**while**' e '**do**…**while**'.

Sfruttando la possibilità di inserire più espressioni in una singola istruzione, si possono realizzare dei cicli '**for**' molto più complessi, anche se questo è sconsigliabile per evitare di scrivere codice troppo difficile da interpretare. In questo modo, l'esempio precedente potrebbe essere ridotto a quello che segue, dove si usa un punto e virgola solitario per rappresentare un'istruzione nulla:

```
int i;

for (i = 0; i < 10; printf ("x"), i++)
  {
    ;
  }
printf ("\n");
```

Se si utilizzano istruzioni multiple, separate con la virgola, occorre tenere presente che **l'espressione che esprime la condizione deve rimanere singola** (se per la condizione si usasse un'espressione multipla, conterebbe solo la valutazione dell'ultima). Naturalmente, nel caso della condizione, si possono costruire condizioni complesse con l'ausilio degli operatori logici, ma rimane il fatto che l'operatore virgola ('**,**') non dovrebbe avere senso lì.

Nel modello sintattico iniziale si vede che le tre espressioni sono opzionali e rimane solo l'obbligo di mettere i punti e virgola relativi. L'esempio seguente mostra un ciclo senza fine che viene interrotto attraverso un'istruzione '**break**':

```
int i = 0;
for (;;)
  {
    if (i >= 10)
      {
        break;
      }
    printf ("x");
    i++;
  }
```

66.1.6 Funzioni

Il linguaggio C offre le funzioni come mezzo per realizzare la scomposizione del codice in subroutine. Prima di poter essere utilizzate attraverso una chiamata, le funzioni devono essere dichiarate, anche se non necessariamente descritte. In pratica, se si vuole indicare nel codice una chiamata a una funzione che viene descritta più avanti, occorre almeno dichiararne il prototipo.

Le funzioni del linguaggio C prevedono il passaggio di parametri solo **per valore**, con tutti i tipi di dati, esclusi gli array (che invece vanno passati per riferimento, attraverso il puntatore alla loro posizione iniziale in memoria).

Il linguaggio C, attraverso la libreria standard, offre un gran numero di funzioni comuni, i cui prototipi vengono incorporati nel codice attraverso l'inclusione di file di intestazione, con l'istruzione '**`#include`**' del precompilatore. Per esempio, come si è già visto, per poter utilizzare la funzione ***printf()*** si deve inserire la riga '**`#include <stdio.h>`**' nella parte iniziale del file sorgente.

66.1.6.1 Dichiarazione di un prototipo

Quando la descrizione di una funzione può essere fatta solo dopo l'apparizione di una sua chiamata, occorre dichiararne il prototipo all'inizio, secondo la sintassi seguente:

tipo nome ([***tipo*** [***nome***] [, …]]) ;

Il tipo, posto all'inizio, rappresenta il tipo di valore che la funzione restituisce. Se la funzione non deve restituire alcunché, si utilizza il tipo '**`void`**'. Se la funzione utilizza dei parametri, il tipo di questi deve essere elencato tra le parentesi tonde. L'istruzione con cui si dichiara il prototipo termina regolarmente con un punto e virgola.

Lo standard C stabilisce che una funzione che non richiede parametri deve utilizzare l'identificatore '**`void`**' in modo esplicito, all'interno delle parentesi.

Segue la descrizione di alcuni esempi.

- `int fattoriale (int);`
  In questo caso, viene dichiarato il prototipo della funzione '**`fattoriale`**', che richiede un parametro di tipo '**`int`**' e restituisce anche un valore di tipo '**`int`**'.
- `int fattoriale (int n);`
  Come nell'esempio precedente, dove in più, per comodità si aggiunge il nome del parametro che comunque viene ignorato dal compilatore.
- `void elenca (void);`
  Si tratta della dichiarazione di una funzione che fa qualcosa senza bisogno di ricevere alcun parametro e senza restituire alcun valore ('**`void`**' è il tipo che rappresenta una variabile di rango nullo e, come tale, incapace di accogliere qualunque valore).

66.1.6.2 Descrizione di una funzione

La descrizione della funzione, rispetto alla dichiarazione del prototipo, richiede l'indicazione dei nomi da usare per identificare i parametri (mentre nel prototipo questi sono facoltativi) e naturalmente l'aggiunta delle istruzioni da eseguire. Le parentesi graffe che appaiono nello schema sintattico fanno parte delle istruzioni necessarie.

tipo nome ([***tipo parametro*** [, …]])
{
  istruzione ;
  …
}

Per esempio, la funzione seguente esegue il prodotto tra i due parametri forniti e ne restituisce il risultato:

```
int prodotto (int x, int y)
{
  return (x * y);
}
```

I parametri indicati tra parentesi, rappresentano una dichiarazione di variabili locali[13] che contengono inizialmente i valori usati nella chiamata. Il valore restituito dalla funzione viene definito attraverso l'istruzione '**`return`**', come si può osservare dall'esempio. Naturalmente, nelle funzioni di tipo '**`void`**' l'istruzione '**`return`**' va usata senza specificare il valore da restituire, oppure si può fare a meno del tutto di tale istruzione.

Nei manuali tradizionale del linguaggio C si descrivono le funzioni nel modo visto nell'esempio precedente; al contrario, nella guida *GNU coding standards* si richiede di mettere il nome della funzione in corrispondenza della colonna uno, così:

```
int
prodotto (int x, int y)
{
  return (x * y);
}
```

Le variabili dichiarate all'interno di una funzione, oltre a quelle dichiarate implicitamente come mezzo di trasporto degli argomenti della chiamata, sono visibili solo al suo interno, mentre quelle dichiarate al di fuori di tutte le funzioni, sono variabili globali, accessibili potenzialmente da ogni parte del programma.[14] Se una variabile locale ha un nome coincidente con quello di una variabile globale, allora, all'interno della funzione, quella variabile globale non è accessibile.

Le regole da seguire, almeno in linea di principio, per scrivere programmi chiari e facilmente modificabili, prevedono che si debba fare in modo di rendere le funzioni indipendenti dalle variabili globali, fornendo loro tutte le informazioni necessarie attraverso i parametri. In questo modo diventa del tutto indifferente il fatto che una variabile locale vada a mascherare una variabile globale; inoltre, ciò permette di non dover tenere a mente il ruolo di queste variabili globali e (se non si usano le variabili «statiche») fa sì che si ottenga una funzione completamente «rientrante».

66.1.6.3 File di intestazione e libreria

Una libreria di funzioni si compone almeno di due parti fondamentali: i prototipi delle funzioni e la descrizione delle funzioni stesse. Secondo la tradizione, l'inclusione di codice attraverso l'istruzione '**`#include`**' del precompilatore, si usa esclusivamente per includere «file di intestazione», contraddistinti convenzionalmente da un nome che termina con il suffisso '**`.h`**'. Questi file di intestazione devono essere costruiti con certi criteri, in modo che la loro inclusione multipla non possa creare problemi. Per quanto riguarda le funzioni, questi file possono contenerne esclusivamente i prototipi.

Per esempio, si potrebbe dire che per poter usare la funzione ***printf()*** si debba includere la «libreria» standard '`stdio.h`'. L'affermazione in sé può essere accettabile, ma non è precisa. Infatti, il file di intestazione '`stdio.h`' contiene prototipi e altre definizioni della porzione della libreria standard che consente di usare la funzione ***printf()***, ma la descrizione effettiva di tale funzione si trova in un altro file.

66.1.7 Vincoli nei nomi

Quando si definiscono variabili e funzioni nel proprio programma, occorre avere la prudenza di non utilizzare nomi che coincidano con quelli delle librerie che si vogliono usare e che non possano andare in conflitto con l'evoluzione del linguaggio. A questo proposito va osservata una regola molto semplice: non si vanno usati nomi «esterni» che inizino con il trattino basso ('_'); in tutti gli altri casi, 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 serve a impedire, per esempio, la creazione di nomi come '**_Bool**' o '**__STDC_IEC_559__**'. Rimane quindi la possibilità di usare nomi che inizino con un trattino basso, purché continuino con un carattere minuscolo e siano visibili solo nell'ambito del file sorgente che si compone.

66.1.8 I/O elementare

L'input e l'output elementare che si usa nella prima fase di apprendimento del linguaggio C si ottiene attraverso l'uso di due funzioni fondamentali: ***printf()*** e ***scanf()***. La prima si occupa di emettere una stringa dopo averla trasformata in base a dei codici di composizione determinati; la seconda si occupa di ricevere input (generalmente da tastiera) e di trasformarlo secondo codici di conversione simili alla prima. Infatti, il problema che si incontra inizialmente, quando si vogliono emettere informazioni attraverso lo standard output per visualizzarle sullo schermo, sta nella necessità di convertire in qualche modo tutti i dati che non siano già di tipo '**char**'. Dalla parte opposta, quando si inserisce un dato che non sia da intendere come un semplice carattere alfanumerico, serve una conversione adatta nel tipo di dati corretto.

Per utilizzare queste due funzioni, occorre includere il file di intestazione '`stdio.h`', come è già stato visto più volte negli esempi.

Le due funzioni, ***printf()*** e ***scanf()***, hanno in comune il fatto di disporre di una quantità variabile di parametri, dove solo il primo è stato precisato. Per questa ragione, la stringa che costituisce il primo argomento deve contenere tutte le informazioni necessarie a individuare quelli successivi; pertanto, si fa uso di ***specificatori di conversione*** che definiscono il tipo e l'ampiezza dei dati da trattare. A titolo di esempio, lo specificatore '**%i**' si riferisce a un valore intero di tipo '**int**', mentre '**%li**' si riferisce a un intero di tipo '**long int**'.

Vengono mostrati solo alcuni esempi, perché una descrizione più approfondita nell'uso delle funzioni ***printf()*** e ***scanf()*** appare in altre sezioni (67.3 e 69.17). Si comincia con l'uso di ***printf()***:

```
…
double capitale = 1000.00;
double tasso    = 0.5;
int    interesse = (capitale * tasso) / 100;
…
printf ("%s: il capitale %f, ", "Ciao", capitale);
printf ("investito al tasso %f%% ", tasso);
printf ("ha prodotto un interesse pari a %i.\n", interesse);
…
```

Gli specificatori di conversione usati in questo esempio si possono considerare quelli più comuni: '**%s**' incorpora una stringa; '**%f**' traduce in testo un valore che originariamente è di tipo '**double**'; '**%i**' traduce in testo un valore '**int**'; inoltre, '**%%**' viene trasformato semplicemente in un carattere percentuale nel testo finale. Alla fine, l'esempio produce l'emissione del testo: «Ciao: il capitale 1000.00, investito al tasso 0.500000% ha prodotto un interesse pari a 5.»

La funzione ***scanf()*** è un po' più difficile da comprendere: la stringa che definisce il procedimento di interpretazione e conversione 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 ("%i", &importo);
…
```

Il pezzo di codice mostrato emette la frase seguente e resta in attesa dell'inserimento di un valore numerico intero, seguito da [*Invio*]:

```
Inserisci l'importo: _
```

Questo valore viene inserito nella variabile ***importo***. Si deve osservare il fatto che gli argomenti successivi alla stringa di conversione sono dei puntatori, per cui, avendo voluto inserire il dato nella variabile ***importo***, questa è stata indicata preceduta dall'operatore '**&**' in modo da fornire alla funzione l'indirizzo corrispondente (si veda la sezione 66.5 sulla gestione dei puntatori).

Con una stessa funzione ***scanf()*** è possibile inserire dati per diverse variabili, come si può osservare dall'esempio seguente, ma in questo caso, per ogni dato viene richiesta la separazione con spazi orizzontali o anche con la pressione di [*Invio*].

```
printf ("Inserisci il capitale e il tasso:");
scanf ("%i%f", &capitale, &tasso);
```

66.1.9 Restituzione di un valore

In un sistema Unix e in tutti i sistemi che si rifanno a quel modello, i programmi, di qualunque tipo siano, al termine della loro esecuzione, restituiscono un valore che può essere utilizzato da uno script di shell per determinare se il programma ha fatto ciò che si voleva o se è intervenuto qualche tipo di evento che lo ha impedito.

Convenzionalmente si tratta di un valore numerico, con un intervallo di valori abbastanza ristretto, in cui zero rappresenta una conclusione normale, ovvero priva di eventi indesiderati, mentre qualsiasi altro valore rappresenta un'anomalia. A questo proposito si consideri quello «strano» atteggiamento degli script di shell, per cui zero equivale a *Vero*.

Lo standard del linguaggio C prescrive che la funzione ***main()*** debba restituire un tipo intero, contenente un valore compatibile con l'intervallo accettato dal sistema operativo: tale valore intero è ciò che dovrebbe lasciare di sé il programma, al termine del proprio funzionamento.

Se il programma deve terminare, per qualunque ragione, in una funzione diversa da ***main()***, non potendo usare l'istruzione '**return**' per questo scopo, si può richiamare la funzione ***exit()***:

```
exit (valore_restituito);
```

La funzione ***exit()*** provoca la conclusione del programma, dopo aver provveduto a scaricare i flussi di dati e a chiudere i file. Per questo motivo, non restituisce un valore all'interno del programma, al contrario, fa in modo che il programma restituisca il valore indicato come argomento.

Per poterla utilizzare occorre includere il file di intestazione '`stdlib.h`' che tra l'altro dichiara già due macro-variabili adatte a definire la conclusione corretta o errata del programma: ***EXIT_SUCCESS*** e ***EXIT_FAILURE***.[15] L'esempio seguente mostra in che modo queste macro-variabili potrebbero essere usate:

```
#include <stdlib.h>
…
…
if (…)
  {
    exit (EXIT_SUCCESS);
  }
else
  {
```

```
    exit (EXIT_FAILURE);
  }
```

Naturalmente, se si può concludere il programma nella funzione ***main()***, si può fare lo stesso con l'istruzione '**`return`**':

```
#include <stdlib.h>
…
…
int main (…)
{
    …
    if (…)
      {
        return (EXIT_SUCCESS);
      }
    else
      {
        return (EXIT_FAILURE);
      }
    …
}
```

66.1.10 Attributi per GNU C

Il compilatore GNU C prevede l'uso di «attributi» nel proprio codice, come estensione del linguaggio. Dal momento che il compilatore GNU C è molto importante e diffuso, conviene sapere che forma possono avere tali attributi, almeno per non restare sbalorditi nella lettura del codice di altri autori:

```
__attribute__ ((tipo_di_attributo))
```

Frequentemente, questi attributi vanno collocati alla fine della dichiarazione di ciò a cui si riferiscono, come nell'esempio seguente, dove viene assegnato l'attributo '**`deprecated`**' al prototipo di una funzione:

```
…
mia_funzione (void) __attribute__ ((deprecated));
…
```

Se può servire, il nome dell'attributo può apparire anche preceduto e terminato da due trattini bassi; pertanto, l'esempio già visto può essere scritto anche così:

```
…
mia_funzione (void) __attribute__ ((__deprecated__));
…
```

Il fatto che siano previsti tali attributi dal compilatore GNU C, rende molto difficile l'individuazione di un errore frequente e banale: la mancanza del punto e virgola alla fine di un prototipo di funzione. Per esempio, si può supporre di avere realizzato un proprio file di intestazione con il contenuto seguente:

```
…
mia_funzione_1 (int a, int b);
mia_funzione_2 (int a, int b)
mia_funzione_3 (int a, int b);
mia_funzione_4 (int a, int b);
…
```

Come si vede, il prototipo di ***mia_funzione_2()*** non è concluso con il punto e virgola. Durante la compilazione di un file che include questa porzione di codice, l'errore che viene evidenziato dal compilatore GNU C è incomprensibile, rispetto alla realtà effettiva:

```
In file included from …/lib/stdio.h:5,
                 from asctime.c:3:
…/lib/stdarg.h: In function 'asctime':
…/lib/stdarg.h:4: error: storage class specified for
  parameter 'va_list'
In file included from …/lib/stdio.h:10,
                 from asctime.c:3:
…/lib/sys/types.h:8: error: storage class specified for
  parameter 'blkcnt_t'
…/lib/sys/types.h:9: error: storage class specified for
  parameter 'blksize_t'
```



```
…/lib/sys/types.h:10: error: storage class specified for
  parameter 'dev_t'
…
…
…/lib/stdio.h:94: error: expected declaration specifiers
  or '...' before 'va_list'
…/lib/stdio.h:96: error: expected declaration specifiers
  or '...' before 'va_list'
asctime.c:7: error: expected '=', ',', ';', 'asm' or
  '__attribute__' before '{' token
asctime.c:99: error: old-style parameter declarations in
  prototyped function definition
asctime.c:99: error: expected '{' at end of input
make: *** [asctime] Error 1
…
```

L'esempio mostrato si riferisce a un errore provocato volutamente nel file di intestazione '`time.h`', a cui mai viene fatto riferimento nell'analisi del compilatore. Pertanto, di fronte a errori così incomprensibili, è determinante il controllo della conclusione corretta dei prototipi delle funzioni, all'interno dei file di intestazione prodotti per proprio conto.

66.2 Istruzioni del precompilatore

Il linguaggio C non può fare a meno del precompilatore e le sue direttive sono regolate dallo standard. Il precompilatore è un programma, o quella parte del compilatore, che si occupa di pre-elaborare un sorgente per generarne uno nuovo, il quale poi viene compilato con tutte le trasformazioni apportate.

Tradizionalmente, in un sistema operativo che si rifà al modello dei sistemi Unix, il precompilatore è costituito dal programma '**`cpp`**' che può essere utilizzato direttamente o in modo trasparente dal compilatore '**`cc`**'. Volendo simulare i passaggi iniziali della compilazione di un programma ipotetico denominato '`prg.c`', evidenziando il ruolo del precompilatore, questi si potrebbero esprimere così:

```
$ cpp -E -o prg.i prg.c [Invio]

$ cc -o prg.o prg.i [Invio]

$ …
```

In questo caso, il file '`prg.i`' generato dal precompilatore è quello che viene chiamato dalla documentazione standard una ***unità di traduzione***. Una unità di traduzione singola può essere il risultato della fusione di diversi file, incorporati attraverso le direttive '**`#include`**', come viene descritto nel capitolo. Ciò che occorre osservare è che, quando si parla di campo di azione legato al «file», ci si riferisce al file generato dal precompilatore, ovvero all'unità di traduzione.

Va osservato che esistono programmi che utilizzano il precompilatore del linguaggio C per fini estranei al linguaggio stesso. Per esempio i file di configurazione delle risorse di X (il sistema grafico) vengono fatti elaborare da '**`cpp`**' prima di essere interpretati.

66.2.1 Linguaggio a sé stante

Le direttive del precompilatore rappresentano un linguaggio a sé stante, con proprie regole. In generale:

- le direttive iniziano con il simbolo '`#`', preferibilmente nella prima colonna;
- le direttive non utilizzano alcun simbolo di conclusione (non si usa il punto e virgola);
- ogni direttiva occupa una riga, la quale può essere spezzata e ripresa in righe successive, utilizzando il simbolo '`\`' subito prima del codice di interruzione di riga;
- su una riga può essere inserita una sola direttiva, perché non c'è altro modo di distinguere la fine di una dall'inizio della successiva.

Se appare un simbolo '`#`' privo di altre indicazioni, questo viene semplicemente ignorato dal precompilatore. Di solito le direttive del

precompilatore si scrivono senza annidamenti, ma questo fatto rischia di rendere particolarmente complicata la lettura del sorgente. A ogni modo, se si usano gli annidamenti, di solito questi riguardano solo le altre direttive e non il codice del linguaggio C puro e semplice.

I commenti del linguaggio C possono apparire solo alla fine delle direttive, ma non in tutte; pertanto vanno usati con prudenza. Vengono usati sicuramente alla fine delle direttive '**#else**' e '**#endif**' per ricordare a quale condizione si riferiscono.

66.2.2 Direttiva «#include»

La direttiva '**#include**' permette di includere un file. Generalmente si tratta di un cosiddetto ***file di intestazione***, contenente una serie di definizioni necessarie al file sorgente in cui vengono incorporate. Il file da incorporare può essere indicato delimitandolo con le parentesi angolari, oppure con gli apici doppi; il modo in cui si delimita il nome del file serve a stabilire come questo deve essere cercato:[16]

```
#include <file>
```

```
#include "file"
```

I due esempi seguenti mostrano la richiesta di includere il file '`stdio.h`' secondo le due forme possibili:

```
#include <stdio.h>
```

```
#include "stdio.h"
```

Delimitando il nome tra parentesi angolari si fa riferimento a un file che dovrebbe trovarsi in una posizione stabilita dalla configurazione del compilatore; per esempio, nel caso di GNU C in un sistema GNU/Linux, dovrebbe trattarsi della directory '`/usr/include/`'. Se invece si delimita il nome tra apici doppi, generalmente si fa riferimento a una posizione precisa nel file system, attraverso l'indicazione di un percorso (secondo la modalità prevista dal sistema operativo); pertanto, scrivendo il nome del file come nell'esempio, si dovrebbe intendere che la sua collocazione debba essere la directory corrente.

Di norma, quando si indica un file da includere delimitandolo con gli apici doppi e senza indicare alcun percorso, se questo file non si trova nella directory corrente, allora viene cercato nella directory predefinita, come se fosse stato indicato tra le parentesi angolari.

Un file incorporato attraverso la direttiva '**#include**', può a sua volta fare lo stesso con altri; naturalmente, questa possibilità va considerata per evitare di includere più volte lo stesso file e di solito si usa un accorgimento che viene descritto più avanti nel capitolo.

66.2.3 Direttiva «#define»

La direttiva '**#define**' serve a definire quelle che sono note come ***macro***, ovvero delle variabili del precompilatore che, successivamente, il precompilatore stesso espande secondo regole determinate. Lo standard del linguaggio C distingue queste macro in due categorie: *object-like macro* e *function-like macro*. Nel corso di questi capitoli si usa la definizione di ***macro-variabile*** nel primo caso e di ***macroistruzione*** nel secondo.

Come sottoinsieme delle macro-variabili vengono considerate le ***costanti manifeste***, per rappresentare dei valori semplici che si ripetono nel sorgente. Per esempio, ***NULL*** è la costante manifesta standard per rappresentare il puntatore nullo.

```
#define macro [sequenza_di_caratteri]
```

La direttiva '**#define**' usata secondo la sintassi mostrata consente di definire delle macro-variabili, ovvero ciò che lo standard definisce *object-like macro*. Ciò che si ottiene è la sostituzione nel sorgente del

nome indicato con la sequenza di caratteri che lo segue. Si osservi l'esempio seguente:

```
#define SALUTO Ciao! Come stai?
```

In questo caso viene dichiarata la macro-variabile ***SALUTO*** in modo tale che tutte le occorrenze di questo nome, successive alla sua dichiarazione, vengano sostituite con '**Ciao! Come stai?**'. È molto importante comprendere questo particolare: tutto ciò che appare dopo il nome della macro, a parte lo spazio che lo separa, viene utilizzato nella sostituzione. L'esempio seguente, invece rappresenta un programma completo.

Listato 66.68. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/8xkVUB59* , *http://ideone.com/HSVI2* .

```
#include <stdio.h>
#define SALUTO "Ciao! come stai?\n"
int main (void)
{
  printf (SALUTO);
  return 0;
}
```

In questo caso, la macro-variabile ***SALUTO*** può essere utilizzata in un contesto in cui ci si attende una stringa letterale, perché include gli apici doppi che sono necessari per questo scopo. Nell'esempio si vede l'uso della macro-variabile come argomento della funzione ***printf()*** e l'effetto del programma è quello di mostrare il messaggio seguente:

```
Ciao! come stai?
```

È bene precisare che la sostituzione delle macro-variabili non avviene se i loro nomi appaiono tra apici doppi, ovvero all'interno di stringhe letterali. Si osservi l'esempio seguente.

Listato 66.70. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/qfPSZZm0* , *http://ideone.com/CAAk3* .

```
#include <stdio.h>
#define SALUTO Ciao! come stai?
int main (void)
{
  printf ("SALUTO\n");
  return 0;
}
```

In questo caso, la funzione ***printf()*** emette effettivamente la parola '**SALUTO**' e non avviene alcuna espansione di macro:

```
SALUTO
```

Una volta compreso il meccanismo basilare della direttiva '**#define**' si può osservare che questa può essere utilizzata in modo più complesso, facendo anche riferimento ad altre macro già definite:

```
#define UNO 1
#define DUE UNO+UNO
#define TRE DUE+UNO
```

In presenza di una situazione come questa, utilizzando la macro ***TRE***, si ottiene prima la sostituzione con '**DUE+UNO**', quindi con '**UNO+UNO+1**', infine con '**1+1+1**' (dopo, tocca al compilatore).

Tradizionalmente i nomi delle macro-variabili vengono definiti utilizzando solo lettere maiuscole, in modo da poterli distinguere facilmente nel sorgente.

Come è possibile vedere meglio in seguito, è sensato anche dichiarare una macro senza alcuna corrispondenza. Ciò può servire per le direttive '**#ifdef**' e '**#ifndef**'.

Nella definizione di una macro-variabile può apparire l'operatore '**##**', con lo scopo di attaccare ciò che si trova alle sue estremità. Si osservi l'esempio seguente.

Listato 66.73. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/XQ6Ns1AT* , *http://ideone.com/R5G3o* .

```
#include <stdio.h>
#define UNITO 1234 ## 5678
int main (void)
{
  printf ("%i\n", UNITO);
  return 0;
}
```

Eseguendo questo programma si ottiene semplicemente l'emissione del numero 12345678. Questo operatore può servire anche per unire assieme il nome di una macro-variabile, benché questo sia poco consigliabile.

Listato 66.74. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/lZ0QKzln* , *http://ideone.com/qOqC1* .

```
#include <stdio.h>
#define MIAMACRO 12345678
#define UNITO MI ## A ## MA ## CRO
int main (void)
{
  printf ("%i\n", UNITO);
  return 0;
}
```

66.2.4 Direttiva «#define» con parametri

La direttiva '**#define**' può essere usata per creare una macroistruzione, ovvero una cosa che viene usata con l'apparenza di una funzione:

```
#define macro(parametro[, parametro]...) sequenza_di_caratteri
```

Per comprendere il meccanismo è meglio avvalersi di esempi. In quello seguente, l'istruzione '**i = QUADRATO(i)**' si traduce in '**i = (i)*(i)**':

```
#define QUADRATO(A)       (A)*(A)
…
…
i = QUADRATO (i);
…
```

Si osservi il fatto che, nella definizione, la stringa di sostituzione è stata composta utilizzando le parentesi: ciò permette di evitare problemi successivamente, nelle precedenze di valutazione delle espressioni, se l'argomento della funzione simulata attraverso la macroistruzione è composto:

```
…
i = QUADRATO (123 * 34 + 3);
…
```

In questo caso, la sostituzione genera '**i = (123 * 34 + 3)*(123 * 34 + 3)**' e si può vedere che le parentesi sono appropriate. L'esempio seguente, costituito da un programma completo, mostra l'uso di due parametri.

Listato 66.77. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/MAxBs8MQ* , *http://ideone.com/ML1LU* .

```
#include <stdio.h>
#define MAX(X, Y) ((X) > (Y) ? (X) : (Y))
int main (void)
{
  printf ("valore massimo tra %i e %i: %i\n",
          3, 4, MAX (3, 4));
  return 0;
}
```

La macroistruzione '**MAX (3, 4)**' si traduce in '**((3) > (4) ? (3) : (4))**'.

È molto importante fare attenzione alla spaziatura nella dichiarazione di una macroistruzione: si può scrivere '**#define MAX(x,y)** …', '**#define MAX(x,y)** …', '**#define MAX(x,y)** …', '**#define MAX(x, y)** …', ecc. Quello che invece non si può proprio è l'inserimento di uno spazio tra il nome della macroistruzione e la parentesi tonda aperta. Pertanto, se si scrive '**#define MAX (x, y)** …' si commette un errore!
Al contrario, quando la macroistruzione viene richiamata, questo spazio può essere inserito senza problemi, come apparso già negli esempi.

Nella definizione di una macroistruzione può essere usato l'operatore '**##**' già descritto nella sezione precedente. Nell'esempio seguente si ottiene di visualizzare il numero 12345678.

Listato 66.78. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/GHXfnaHL* , *http://ideone.com/WRD5n* .

```
#include <stdio.h>
#define UNISCI(A, B) A ## B
int main (void)
{
  printf ("%i\n", UNISCI(1234, 5678));
  return 0;
}
```

Inoltre, è disponibile l'operatore '**#**' che ha lo scopo di racchiudere tra apici doppi la metavariabile che lo segue immediatamente. Si osservi l'esempio seguente:

Listato 66.79. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/1nEG1ryz* , *http://ideone.com/czt2V* .

```
#include <stdio.h>
#define STRINGATO(a) # a
#define SALUTO STRINGATO (Ciao! come stai?\n)
int main (void)
{
  printf (SALUTO);
  return 0;
}
```

Prima viene definita la macroistruzione ***STRINGATO***, con la quale si vuole che il suo argomento sia raccolto tra apici doppi. Subito dopo viene definita la macro-variabile ***SALUTO*** che viene rimpiazzata da '**STRINGATO (Ciao! come stai?\n)**' e quindi da '**"Ciao! come stai?\n"**'. Alla fine, il programma mostra regolarmente il messaggio già visto in un altro esempio precedente:

```
Ciao! come stai?
```

Si osservi cosa accadrebbe modificando l'esempio nel modo seguente, dove si vuole che la macroistruzione ***STRINGATO*** utilizzi due parametri:

```
…
#define STRINGATO(a, b) # a # b
#define SALUTO STRINGATO (Ciao!, come stai?\n)
…
```

Evidentemente si vuole che i due argomenti forniti alla macroistruzione ***STRINGATO*** siano raccolti ognuno tra apici doppi, pertanto la macro-variabile si trova a essere dichiarata, sostanzialmente come '**"Ciao!" "come stai?\n"**'. Alla fine il risultato mostrato dal programma è differente, perché la sequenza delle due stringhe viene intesa come una sequenza sola, ma in tal caso manca lo spazio tra le due parti:

```
Ciao!come stai?
```

Si può complicare ulteriormente l'esempio per dimostrare fino a dove si estende la competenza dell'operatore '**#**', come si vede nel listato successivo.

Listato 66.83. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/idGYtl78* , *http://ideone.com/17IQ17c* .

```
#include <stdio.h>
#define STRINGATO(a, b) # a , b
#define SALUTO STRINGATO (%i un amore\n, 6)
int main (void)
{
  printf (SALUTO);
  return 0;
}
```

Qui gli spazi sono importanti, infatti, la macroistruzione ***STRINGATO*** si traduce in '**"*a*" , *b***' e la virgola non avrebbe potuto essere unita alla lettera «a», altrimenti sarebbe stata inserita dentro la coppia di apici doppi. La macro-variabile ***SALUTO*** si traduce poi in '**"%i un amore\n" , 6**', pertanto, alla fine, il programma mostra il messaggio seguente:

```
6 un amore
```

Per concludere viene mostrato un esempio ulteriore, con il quale si crea una sorta di funzione che il precompilatore deve trasformare in un blocco di istruzioni. Viene simulato il comportamento della funzione standard ***strncpy()***, senza però restituire un valore.

Listato 66.85. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/eCbQbPAR* , *http://ideone.com/A2Q2L* .

```
#include <stdio.h>

#define STRNCPY(DST, ORG, N) { \
    char *s1 = (DST); \
    const char *s2 = (ORG); \
    size_t n = (N); \
    int i; \
    for (i = 0; i < n && s2[i] != 0; i++) \
      { \
        s1[i] = s2[i]; \
      } \
    s1[i] = 0; }

int main (void)
{
  char stringa[100];
  STRNCPY (stringa, "Buon giorno a tutti!", 50)   // [1]
                            // [1] Si osservi che manca il
                            //     punto e virgola finale!
  printf ("%s\n", stringa);
  return 0;
}
```

Si può vedere che, per richiamare questa macroistruzione, non si richiede che le sia aggiunto il punto e virgola. Infatti, la macro in sé si espande in un raggruppamento tra parentesi graffe, che non ne ha bisogno; d'altra parte, volendoglielo aggiungere, non si può creare alcun problema.

La dichiarazione di una macroistruzione può prevedere una quantità variabile di parametri, come avviene già per le funzioni (sezione 66.6.3). Per ottenere questo si aggiungono dei puntini di sospensione alla fine dell'elenco dei parametri fissi, quindi, si utilizza la parola chiave '**__VA_ARGS__**' per individuare gli argomenti opzionali. L'esempio seguente riproduce il funzionamento di ***printf()***, richiamando la stessa funzione.

Listato 66.86. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/j98SG985Wo* , *http://ideone.com/cOKlA* .

```
#include <stdio.h>

#define PRINTF(A, ...) printf (A, __VA_ARGS__)

int main (void)
{
  PRINTF ("I primi numeri interi: %i, %i, %i\n", 1, 2, 3);
  return 0;
}
```


Questa volta il punto e virgola finale serve, perché non è stato incluso nella definizione della macroistruzione.

A proposito di '**__VA_ARGS__**' va ancora osservato che individua sì gli argomenti opzionali, ma di questi ne deve essere specificato almeno uno. Pertanto, la macroistruzione ***PRINTF()***, per come è stata dichiarata nell'esempio precedente, va usata sempre con almeno due argomenti. In questo caso, per poter usare la macroistruzione con un argomento solo, la sua definizione va modificata nel modo seguente:

```
…
#define PRINTF(...) printf (__VA_ARGS__)
…
```

66.2.5 Direttive «#if», «#else», «#elif» e «#endif»

Le direttive '**#if**', '**#else**', '**#elif**' e '**#endif**', permettono di delimitare una porzione di codice che debba essere utilizzato o ignorato in relazione a una certa espressione che può essere calcolata solo attraverso definizioni precedenti.

```
#if espressione
    espressione
[#elif espressione
      espressione]
…
[#else
      espressione]
#endif
```

Le espressioni che rappresentano le condizioni da valutare seguono regole equivalenti a quelle del linguaggio, tenendo conto che se si vogliono usare delle variabili, queste possono solo essere quelle del precompilatore. L'esempio seguente mostra la dichiarazione di una macro-variabile a cui si associa un numero, quindi si vede un confronto basato sul valore in cui si espande la macro-variabile stessa:

```
#define DIM_MAX 1000
…
…
int main (void)
{
…
#if DIM_MAX>100
  printf ("Dimensione enorme.\n");
  …
#else
  printf ("Dimensione normale.\n");
  …
#endif
…
}
```

L'esempio mostra il confronto tra la macro-variabile ***DIM_MAX*** e il valore 100. Essendo stata dichiarata per tradursi in 1 000, il confronto è equivalente a 1 000 > 100 che risulta vero, pertanto il compilatore include solo le istruzioni relative.

Gli operatori di confronto che si possono utilizzare per le espressioni logiche sono i soliti, in particolare, è bene ricordare che per valutare l'uguaglianza si usa l'operatore '**==**', come nell'esempio successivo:

```
#define NAZIONE ita
…
…
int main (void)
{
#if NAZIONE==ita
  char valuta[] = "EUR";
  …
#elif NAZIONE==usa
  char valuta[] = "USD";
  …
#endif
…
}
```

Queste direttive condizionali possono essere annidate; inoltre possono contenere anche altri tipi di direttiva del precompilatore.

66.2.6 Direttive «#if defined», «#if !defined», «#ifdef» e «#ifndef»

Nelle espressioni che esprimono una condizione per la direttiva '**#if**' è possibile usare l'operatore '**defined**', seguito dal nome di una macro-variabile. La condizione '**defined** ***macro***' si avvera se la macro indicata risulta definita, anche se dovesse essere priva di valore. Per converso, la condizione '**!defined** ***macro***' si avvera quando la macro non risulta definita.

La direttiva '**#if defined**' può essere abbreviata come '**#ifdef**', mentre '**#if !defined**' si può esprimere come '**#ifndef**'.

```
#define DEBUG
…
int main (void)
{
…
#if defined DEBUG
  printf ("Punto di controllo n. 1\n");
  …
#endif // DEBUG
…
}
```

```
#define DEBUG
…
int main (void)
{
…
#ifdef DEBUG
  printf ("Punto di controllo n. 1\n");
  …
#endif // DEBUG
…
}
```

I due esempi equivalenti mostrano il caso in cui sia dichiarata una macro ***DEBUG*** (che non si traduce in alcunché) e in base alla sua esistenza viene incluso il codice che mostra un messaggio particolare.

```
#define OK
…
int main (void)
{
#if !defined OK
  printf ("Punto di controllo n. 1\n");
  …
#endif // OK
…
}
```

```
#define OK
…
int main (void)
{
#ifndef OK
  printf ("Punto di controllo n. 1\n");
  …
#endif // OK
…
}
```

Questi due esempi ulteriori sono analoghi a quanto già mostrato, con la differenza che le istruzioni controllate vengono incluse nella compilazione solo se la macro indicata non è stata dichiarata.

Quando si scrivono delle condizioni basate sull'esistenza o meno di una macro, può essere utile aggiungere alla conclusione un commento con cui si ricorda a quale macro si sta facendo riferimento, in modo da districarsi più facilmente in presenza di più livelli di annidamento. Ma occorre fare molta attenzione, perché se si commettono errori con questi commenti il compilatore non può dare alcuna segnalazione in merito e si rende incomprensibile il sorgente alla rilettura successiva.

Esiste una situazione ricorrente in cui viene utilizzata la direttiva '**#if !defined**' o '**#ifndef**' che è bene conoscere. Spesso i file di intestazione che vengono inclusi con direttive '**#include**' includono a loro volta tutto quello che serve loro, ma così facendo c'è la possibilità che lo stesso file venga incluso più volte. Per evitare di prendere in considerazione una seconda volta lo stesso file, si usa un artificio molto semplice, come si vede nel listato successivo che riproduce il contenuto del file '`stdbool.h`' di una libreria standard ipotetica:

```
#ifndef _STDBOOL_H
#define _STDBOOL_H      1

#define bool    _Bool
#define true    1
#define false   0
#define __bool_true_false_are_defined   1

#endif // _STDBOOL_H
```

Come si vede, se il codice viene eseguito per la prima volta, la condizione '**ifndef** ***_STDBOOL_H***' non si avvera e di conseguenza la macro-variabile ***_STDBOOL_H*** viene creata effettivamente e quindi viene considerato tutto il resto del codice fino alla direttiva '**#endif**'. Ma quando si tenta di eseguire lo stesso codice per la seconda volta, o per altre volte successive, dato che la macro-variabile ***_STDBOOL_H*** risulta già definita, questo codice viene ignorato semplicemente, senza altre conseguenze.

Le direttive che consentono di compilare selettivamente solo una porzione del codice, consentono di realizzare del codice molto sofisticato, ma rischiano di renderlo estremamente complesso da interpretare attraverso la lettura umana. Pertanto, è bene limitarne l'uso alle situazioni che sono utili effettivamente.

66.2.7 Direttiva «#undef»

La direttiva '**#undef**' permette di eliminare una macro a un certo punto del sorgente:

```
#undef macro
```

Si mostra un esempio molto semplice, nel quale prima si dichiara la macro-variabile ***NAZIONE***, poi, quando non serve più, questa viene eliminata.

```
#define NAZIONE ita
…
/* In questa posizione, NAZIONE risulta definita */
…
#undef NAZIONE
…
/* In questa posizione, NAZIONE non è definita */
…
```

66.2.8 Direttiva «#line»

Di norma, il compilatore abbastanza evoluto consente di inserire nel file eseguibile delle informazioni che consentano di abbinare il codice eseguibile alle righe del file sorgente originale. Per esempio, con GNU C si può usare l'opzione '**-gstabs**' e altre simili. Naturalmente, in condizioni normali il compilatore conta da solo le righe e annota il nome del file sorgente originale.

Con la direttiva '**#line**' è possibile istruire il compilatore in modo che tenga in considerazione un numero di riga differente, ma soprattutto consente di specificare a quale file sorgente ci si vuole riferire.

```
#line n_riga ["nome_file_sorgente"]
```

C'è da osservare che, per il programmatore, è poco probabile che sia necessario indicare una riga diversa nello stesso sorgente. In effetti, diventa più utile se si abbina il nome di un altro file. Per comprendere come possa essere utilizzata questa possibilità, occorre ipotizzare la costruzione di un altro compilatore per un linguaggio nuovo, con il quale si genera codice in linguaggio C. A titolo di esempio si suppone di volere tradurre il file '`hanoi.pseudo`' che si vede nel listato 66.96 in un sorgente C, denominato '`hanoi.c`', mantenendo il riferimento alle righe originali.

Listato 66.96. Il file '`hanoi.pseudo`'.

```
1   HANOI (N, P1, P2)
2       IF N > 0
3           THEN
4               HANOI (N-1, P1, 6-P1-P2)
5               scrivi: "Muovi l'anello" N "dal piolo" P1 "al piolo" P2
6               HANOI (N-1, 6-P1-P2, P2)
7       END IF
8   END HANOI
9
10  MAIN ()
11      HANOI (3, 1, 2)
12  END MAIN
```

Per ottenere il risultato atteso, il file '`hanoi.c`' deve contenere diverse direttive '**#line**', come si vede nel listato 66.97, anche se alcune di quelle potrebbero essere omesse, contando sull'incremento automatico da parte del compilatore.

Listato 66.97. Il file '`hanoi.c`'.

```
#include <stdio.h>

#line 1 "hanoi.pseudo"
void hanoi (int N, int P1, int P2)
{
  #line 2 "hanoi.pseudo"
  if (N > 0)
    {
      #line 4 "hanoi.pseudo"
      hanoi (N-1, P1, 6-P1-P2);
      #line 5 "hanoi.pseudo"
      printf ("Muovi l'anello %i dal piolo %i al piolo %i\n", N, P1, P2);
      #line 6 "hanoi.pseudo"
      hanoi (N-1, 6-P1-P2, P2);
      #line 7 "hanoi.pseudo"
    }
  #line 8 "hanoi.pseudo"
}

#line 10 "hanoi.pseudo"
int main (void)
{
  #line 11 "hanoi.pseudo"
  hanoi (3, 1, 2);
  #line 12 "hanoi.pseudo"
  return 0;
  #line 12 "hanoi.pseudo"
}
```

La compilazione del file '`hanoi.c`' potrebbe avvenire nel modo seguente:

```
$ cc -Wall -gstabs hanoi.c
```

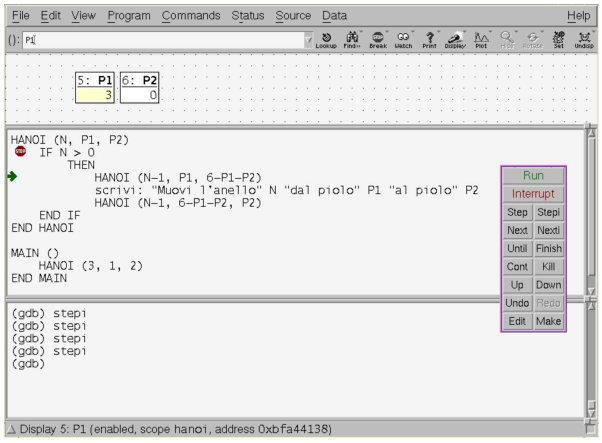
Si dovrebbe ottenere il file eseguibile '**a.out**' e si verifica sommariamente se funziona:

```
$ ./a.out

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

Il risultato è quello previsto. Se lo si esegue con l'ausilio di programmi come GDB, si può osservare che il riferimento al sorgente originale è quello del file '`hanoi.pseudo`':


```
$ gdb a.out

(gdb) break main [Invio]

  Breakpoint 1 at 0x80483d8: file hanoi.pseudo, line 11.
(gdb) run [Invio]

  Starting program: /home/tizio/a.out
  …
  Breakpoint 1, main () at hanoi.pseudo:11
  11          HANOI (3, 1, 2)
(gdb) stepi [Invio]

  0x080483e0      11          HANOI (3, 1, 2)
(gdb) stepi [Invio]

  0x080483e8      11          HANOI (3, 1, 2)
(gdb) stepi [Invio]

  0x080483ef      11          HANOI (3, 1, 2)
(gdb) stepi [Invio]

  hanoi (n=3, p1=1, p2=2) at hanoi.pseudo:2
  2           IF N > 0
(gdb) stepi [Invio]

  0x08048355      2           IF N > 0
(gdb) stepi [Invio]

  0x08048357      2           IF N > 0
(gdb) stepi [Invio]

  2           IF N > 0
  (gdb) stepi
  0x0804835e      2           IF N > 0
(gdb) stepi [Invio]

  4                   HANOI (N-1, P1, 6-P1-P2)
(gdb) stepi [Invio]

  0x08048363      4                   HANOI (N-1, P1, 6-P1-P2)
(gdb) quit [Invio]
```

Figura 66.110. Esecuzione controllata del programma attraverso DDD.



66.2.9 Direttiva «#error»

La direttiva '**#error**' serve a generare un messaggio diagnostico in fase di compilazione, normalmente con lo scopo di interrompere lì il procedimento. In pratica è un modo per interrompere la compilazione già in fase di elaborazione da parte del precompilatore, al verificarsi di certe condizioni.

```
#error messaggio
```

Il messaggio viene trattato in modo letterale, senza l'espansione delle macro.

```
#if ! __STDC_IEC_559__
#error compilatore non conforme alle specifiche IEC 60599!
#endif
```

L'esempio mostra una situazione verosimile per l'utilizzo della direttiva '**#error**', dove si controlla che il valore in cui si espande la macro-variabile ***__STDC_IEC_559__*** sia diverso da zero, ma se non è così viene visualizzato il messaggio di errore e la compilazione dovrebbe venire interrotta.

66.2.10 Macro predefinite

Lo standard del C prevede che il compilatore disponga di alcune macro-variabili predefinite, elencate sinteticamente nella tabella successiva.

Tabella 66.112. Macro-variabili predefinite secondo lo standard.

Macro-variabile	Descrizione
`__DATE__` `__TIME__`	La data e l'ora della compilazione sono accessibili attraverso le macro-variabili ***__DATE__*** e ***__TIME__***. Il formato della prima macro-variabile è '**"*Mmm gg aaaa*"**' e quello della seconda è '**"*hh*:*mm*:*ss*"**'. Come si vede, le due macro-variabili si espandono in una stringa delimitata correttamente da apici doppi.
`__FILE__` `__LINE__`	Attraverso le macro-variabili ***__FILE__*** e ***__LINE__*** il programma può accedere all'informazione sul nome del file sorgente e della riga originale. Il nome del file e il numero della riga possono essere alterati attraverso la direttiva '**#line**'.

`__STDC__`
`__STDC_HOSTED__`
`__STDC_VERSION__`
	La macro-variabile ***__STDC__*** che si espande nel valore '**1**' sta a indicare che si tratta di un compilatore conforme allo standard; la macro ***__STDC_HOSTED__***, se si espande nel valore '**1**', indica una conformità stretta, definita come *hosted implementation*; la macro ***__STDC_VERSION__*** si espande nella versione dello standard. Il valore in cui si espande la terza macro-variabile contiene l'anno e il mese, come per esempio '**199901L**', con la specificazione che si tratta di una costante numerica di tipo '**long int**'.

`__STDC_IEC_559__`	Se esiste la macro-variabile ***__STDC_IEC_599__*** che si espande nel valore '**1**', si intende indicare la conformità alle specifiche dello standard IEC 60559, inerenti l'aritmetica a virgola mobile.
`__STDC_IEC_559_COMPLEX__`	Se esiste la macro-variabile ***__STDC_IEC_599_COMPLEX__*** che si espande nel valore '**1**', si intende indicare la conformità alle specifiche dello standard IEC 60559, inerenti l'aritmetica «complessa».

Macro-variabile	Descrizione
`__STDC_ISO_10646__`	Se esiste la macro-variabile ***__STDC_ISO_10646__***, questa dovrebbe espandersi nella versione dello standard ISO/IEC 10646 che riguarda la codifica universale dei caratteri. La versione che si ottiene è un numero contenente l'anno e il mese, seguito dalla lettera «L», a indicare che si tratta di una costante numerica di tipo '**long int**'.

A parte il caso di ***__FILE__*** e ***__LINE__***, le macro-variabili si espandono in un valore fisso.

66.2.11 Pragma

Attraverso i «pragma» è possibile dare al compilatore delle istruzioni che sono al di fuori dello standard. Il pragma, in sé, è un messaggio testuale che viene passato al compilatore, il quale può interpretarlo in fase di precompilazione o in quella successiva. Lo standard prevede due forme per esprimere un pragma al compilatore:

```
#pragma messaggio
```

```
_Pragma ("messaggio")
```

Il testo che compone il pragma nella sua prima forma viene trattato letteralmente, mentre quello del secondo modello richiede la protezione di alcuni caratteri: '**\"**' e '****' corrispondono rispettivamente a '**"**' e '****'. I due esempi seguenti sono equivalenti:

```
#pragma GCC dependency "parse.y"
```

```
_Pragma ("GCC dependency \"parse.y\"")
```

Lo standard prevede anche che sia possibile creare delle macroistruzioni che incorporino un pragma, come nell'esempio seguente:

```
#define DO_PRAGMA(x) _Pragma (#x)
DO_PRAGMA (GCC dependency "parse.y")
```

66.3 Dal campo di azione alla compilazione

Il problema del campo di azione di variabili e funzioni va visto assieme a quello della compilazione di un programma composto da più file sorgenti, attraverso la produzione di file-oggetto distinti. Leggendo questo capitolo occorre tenere presente che la descrizione della questione è semplificata, omettendo alcuni dettagli. D'altra parte, per poter comprendere il problema la semplificazione è necessaria, tenendo conto che nel linguaggio C, per controllare il campo di azione delle variabili e delle funzioni, si utilizzano parole chiave non proprio «azzeccate» e in certi casi con significati diversi in base al contesto.

Per una descrizione più precisa e dettagliata, dopo la lettura di questo capitolo è necessario rivolgersi ai documenti che definiscono lo standard del linguaggio.

66.3.1 Il punto di vista del «collegatore»

Il programma che raccoglie assieme diversi file-oggetto per creare un file eseguibile (ovvero il *linker*), deve «collegare» i riferimenti incrociati a simboli di variabili e funzioni. In pratica, se nel file '`uno.o`' si fa riferimento alla funzione ***f()*** dichiarata nel file '`due.o`', nel programma risultante tale riferimento deve essere risolto con degli indirizzi appropriati. Naturalmente, lo stesso vale per le variabili globali, dichiarate da una parte e utilizzate anche dall'altra.

Per realizzare questi riferimenti incrociati, occorre che le variabili e le funzioni utilizzate al di fuori del file-oggetto in cui sono dichiarate, siano pubblicizzate in modo da consentire il richiamo da

altri file-oggetto. Per quanto riguarda invece le variabili e le funzioni dichiarate e utilizzate esclusivamente nello stesso file-oggetto, non serve questa forma di pubblicità.

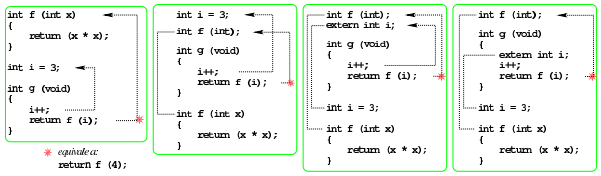
Nei documenti che descrivono il linguaggio C standard si usa una terminologia specifica per distinguere le due situazioni: quando una variabile o una funzione viene dichiarata e usata solo internamente al file-oggetto rilocabile che si ottiene, è sufficiente che abbia una «collegabilità interna», ovvero un ***linkage interno***; quando invece la si usa anche al di fuori del file-oggetto in cui viene dichiarata, richiede una «collegabilità esterna», ovvero un ***linkage esterno***.

Nel linguaggio C, il fatto che una variabile o una funzione sia accessibile al di fuori del file-oggetto rilocabile che si ottiene, viene determinato in modo implicito, in base al contesto, nel senso che non esiste una classe di memorizzazione esplicita per definire questa cosa.

66.3.2 Campo di azione legato al file sorgente

Il file sorgente che si ottiene dopo l'elaborazione da parte del precompilatore, è suddiviso in componenti costituite essenzialmente dalla dichiarazione di variabili e di funzioni (prototipi inclusi). L'ordine in cui appaiono queste componenti determina la ***visibilità*** reciproca: in linea di massima si può accedere solo a quello che è già stato dichiarato. Inoltre, in modo predefinito, dopo la trasformazione in file-oggetto, queste componenti sono accessibili anche da altri file, per i quali, l'ordine di dichiarazione nel file originale non è più importante.[17]

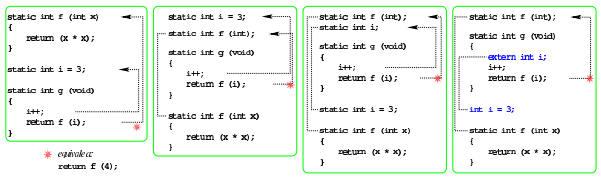
Figura 66.116. Quattro file sorgenti equivalenti, a confronto. La variabile ***i***, la funzione ***f()*** e la funzione ***g()*** sarebbero accessibili anche da altri file. La funzione ***g()*** utilizza la variabile ***i***, dichiarata esternamente a lei.



Nell'esempio della figura precedente, la funzione ***g()*** accede direttamente alla variabile ***i*** che risulta dichiarata al di fuori della funzione stessa. Il campo di azione di questa variabile inizia dalla sua dichiarazione e termina alla fine del file; quando la variabile viene definita in una posizione successiva al suo utilizzo, questa deve essere dichiarata preventivamente come «esterna», attraverso lo specificatore di classe di memorizzazione '**`extern`**'.

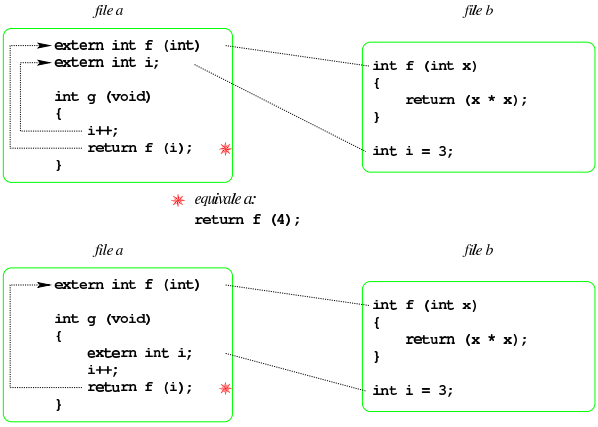
Per isolare le funzioni e la variabile degli esempi mostrati, in modo che non siano disponibili per il collegamento con altri file, si dichiarano per il solo uso locale attraverso lo specificatore di classe di memorizzazione '**`static`**', come si vede nella figura successiva. Va osservato che, nell'ultimo caso, la variabile ***i*** non può essere isolata dall'esterno, perché si trova in una posizione successiva al suo utilizzo, pertanto vi si accede come se fosse dichiarata in un altro file.

Figura 66.117. Quattro file sorgenti equivalenti a confronto, in cui, dove è stato possibile, le variabili e le funzioni sono state isolate dal collegamento con l'esterno.



Per accedere a una funzione o a una variabile definita in un altro file[18] si deve dichiarare localmente la funzione o la variabile con lo specificatore di classe di memorizzazione '**`extern`**'. La figura successiva mostra l'esempio già apparso, ma diviso in due file.

Figura 66.118. Due file collegati tra di loro: il primo file («a») viene proposto in due versioni equivalenti.



Questi esempi mostrano che è possibile dichiarare la variabile «esterna» direttamente all'interno della funzione che ne fa uso; tuttavia, per la scrittura di un programma ordinato, è più grazioso se questa dichiarazione appare al di fuori delle funzioni.

Negli esempi mostrati non appare la funzione ***main()*** che, invece, in un programma comune deve esistere. È da osservare che la funzione ***main()*** non può essere dichiarata con lo specificatore di classe di memorizzazione '**`static`**', anche se tutto è incluso in un file unico, perché dopo la produzione del file-oggetto rilocabile, per produrre un file eseguibile si associano normalmente delle librerie che contengono il codice iniziale del programma, il quale va a chiamare poi la funzione ***main()***. In altre parole, la compilazione prevede quasi sempre l'associazione con un file-oggetto fantasma contenente il codice responsabile della chiamata della funzione ***main()***, la quale, così, deve essere accessibile all'esterno del proprio file.

Tabella 66.119. Specificatori di classe di memorizzazione utilizzabili nella dichiarazione delle funzioni e delle variabili al di fuori delle funzioni.

Parola chiave	Descrizione
	L'assenza dello specificatore di classe implica la dichiarazione di una variabile o di una funzione accessibile anche da altri file.
`static`	Lo specificatore di classe '**`static`**' definisce una variabile o una funzione che può essere utilizzata solo all'interno del file in cui appare.
`extern`	Indica il riferimento a una variabile o a una funzione dichiarata in un altro file, oppure, nel caso delle variabili, anche nel file stesso ma in una posizione successiva.

66.3.3 Semplificazione dovuta all'uso comune dei file di intestazione

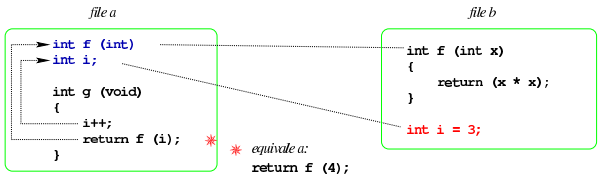
Nella tradizione del linguaggio C si fa uso di file di intestazione, ovvero porzioni di codice, in cui, tra le altre cose, si vanno a mettere i prototipi delle funzioni e le dichiarazioni delle variabili globali, a cui tutto il programma deve poter accedere.

Per semplificare questo lavoro di fusione, spesso un file incluso ne include automaticamente altri, da cui il proprio codice può dipendere. Così facendo, può anche succedere che lo stesso prototipo o la stessa variabile appaiano dichiarati più volte nello stesso file finale (quello generato dal precompilatore).

Oltre a questo fatto, se il proprio programma è suddiviso in più file, i quali devono includere questo o quel file di intestazione, diventa impossibile precisare da quale parte i prototipi e le variabili vengono dichiarate e da quale altra parte vengono richiamate. Pertanto, di norma si lascia fare al compilatore. L'esempio di compilazione

di due file, presentato alla fine della sezione precedente, va rivisto secondo quanto si vede nella figura successiva.

Figura 66.120. Due file collegati tra di loro senza dichiarare espressamente la classe di memorizzazione '**extern**'.



Naturalmente, è bene che le funzioni e le variabili pubbliche siano dichiarate sempre nello stesso modo; inoltre, se le variabili pubbliche devono essere inizializzate, ciò può avvenire una volta sola, in un solo file.

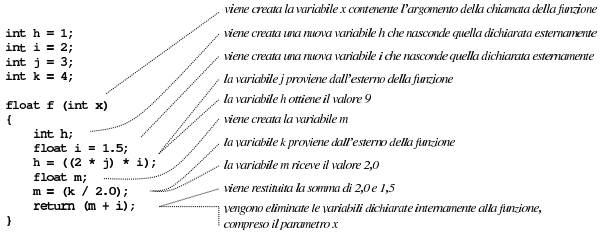
La classe di memorizzazione '**extern**' è predefinita per i prototipi di funzione (purché non siano incorporati all'interno di altre funzioni) e per la dichiarazione delle variabili, purché assieme alla dichiarazione non ci sia anche un'inizializzazione. In pratica, nell'esempio non si può dichiarare espressamente con la parola chiave '**extern**' la variabile ***i*** nel file 'b', dove viene anche inizializzata. Se si tenta di farlo, il compilatore dovrebbe segnalare un errore.

66.3.4 Campo di azione interno alle funzioni

All'interno delle funzioni sono accessibili le variabili globali dichiarate esternamente a loro (come descritto nella sezione precedente), inoltre sono dichiarate implicitamente le variabili che costituiscono i parametri, dai quali si ricevono gli argomenti della chiamata, e si possono aggiungere altre variabili «locali». I parametri e le altre variabili che si dichiarano nella funzione sono visibili solo nell'ambito della funzione stessa; inoltre, se i nomi delle variabili e dei parametri sono gli stessi di variabili dichiarate esternamente, ciò rende temporaneamente inaccessibili quelle variabili esterne.

In condizioni normali, sia le variabili che costituiscono i parametri, sia le altre variabili dichiarate localmente all'interno di una funzione, vengono eliminate all'uscita dalla funzione stessa. Di norma ciò avviene utilizzando la pila dei dati che di solito ogni processo elaborativo dispone (si veda eventualmente la sezione 64.10).

Figura 66.121. Variabili «automatiche» dichiarate implicitamente come tali.



Le variabili create all'interno di una funzione, nel modo descritto dalla figura precedente, sono ***variabili automatiche*** ed è possibile esplicitare questa loro caratteristica con lo specificatore di classe di memorizzazione '**auto**'. Pertanto, la stessa cosa sarebbe stata ottenuta scrivendo l'esempio come nella figura successiva.

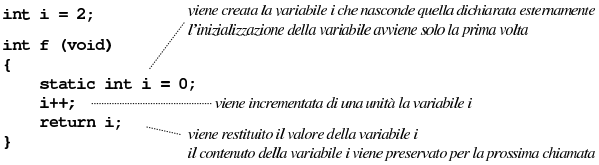
Figura 66.122. Variabili «automatiche» dichiarate espressamente attraverso lo specificatore di classe di memorizzazione '**auto**'.

```
int h = 1;
int i = 2;
int j = 3;
int k = 4;

float f (int x)
{
    auto int h;
    auto float i = 1.5;
    h = ((2 * j) * i);
    auto float m;
    m = (k / 2.0);
    return (m + i);
}
```

All'interno di una funzione è possibile utilizzare variabili che facciano riferimento a porzioni di memoria che non vengono rilasciate all'uscita della funzione stessa, pur isolandole rispetto alle variabili dichiarate esternamente. Si ottiene questo con lo specificatore di classe di memorizzazione '**static**' che non va confuso con lo stesso specificatore usato per le variabili dichiarate esternamente alle funzioni. In altre parole, quando in una funzione si dichiara una variabile con lo specificatore di classe di memorizzazione '**static**', si ottiene di conservare il contenuto di quella variabile che torna a essere accessibile nelle chiamate successive della funzione.

Di norma, la dichiarazione di una variabile di questo tipo coincide con la sua inizializzazione; in tal caso, l'inizializzazione avviene solo quando si chiama la funzione la prima volta.

Figura 66.123. Variabili «statiche» (da intendersi come variabili private) dichiarate all'interno delle funzioni.



All'interno delle funzioni possono essere usati anche gli specificatori di classe di memorizzazione '**register**' e '**extern**', come descritto nella tabella successiva.

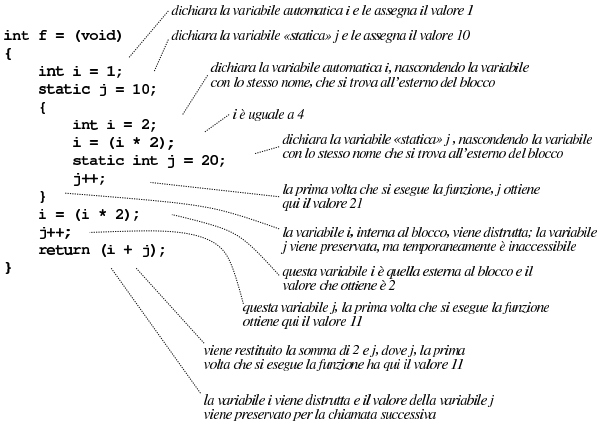
Tabella 66.124. Specificatori di classe di memorizzazione utilizzabili nella dichiarazione delle variabili all'interno delle funzioni.

Parola chiave	Descrizione
`auto`	È lo specificatore di classe di memorizzazione predefinito e indica che la variabile viene creata in corrispondenza della dichiarazione e viene eliminata all'uscita della funzione.
`register`	Con lo specificatore di classe di memorizzazione '**register**' si chiede di creare una variabile automatica che, se possibile, utilizzi un registro del microprocessore o qualunque altra risorsa limitata che possa ridurne i tempi di accesso.
`static`	Definisce una variabile «privata» allocando della memoria che non viene rilasciata alla conclusione dell'attività della funzione, conservando il valore memorizzato per la chiamata successiva della stessa funzione. Si tratta comunque di una variabile a cui può accedere solo la funzione in cui è dichiarata.
`extern`	Indica il riferimento a una variabile dichiarata «esternamente» (come già mostrato nella sezione precedente). In generale, sarebbe meglio dichiarare in questo modo solo le variabili che sono definite al di fuori delle funzioni, lasciando che le funzioni vi accedano semplicemente in qualità di variabili globali.

66.3.5 Campo di azione interno ai raggruppamenti di istruzioni

Le variabili dichiarate all'interno di raggruppamenti di istruzioni, ovvero all'interno di parentesi graffe, si comportano esattamente come quelle dichiarate all'interno delle funzioni: il loro campo di azione termina all'uscita dal blocco. L'esempio della figura successiva mostra un raggruppamento di istruzioni contenente la dichiarazione di una variabile automatica e di una «statica», con la descrizione dettagliata di ciò che accade, dentro e fuori dal raggruppamento.

Figura 66.125. Vita delle variabili all'interno dei raggruppamenti di istruzioni.



La dimostrazione serve a comprendere che, all'interno di una funzione, la posizione in cui si dichiara una variabile non è indifferente: in generale, per migliorare la leggibilità del codice, sarebbe bene dichiarare le variabili all'inizio delle funzioni, evitando accuratamente di farlo all'interno di raggruppamenti annidati.

66.3.6 Funzioni annidate

Così come esistono i raggruppamenti di istruzioni, all'interno dei quali la dichiarazione delle variabili ha un proprio campo di azione limitato, è possibile anche dichiarare delle sottofunzioni, accessibili solo all'interno delle funzioni stesse, dopo che sono state dichiarate. Queste sottofunzioni non possono avere uno specificatore di classe di memorizzazione e appartengono esclusivamente alla funzione che le contiene.

In generale, l'uso di sottofunzioni è sconsigliabile e, d'altra parte, originariamente non era permesso.

66.3.7 Visibilità, accessibilità, staticità

Va chiarita la distinzione che c'è tra la visibilità di una variabile e l'accessibilità al suo contenuto. Quando una funzione dichiara delle variabili automatiche o statiche con un certo nome, se questa funzione chiama a sua volta un'altra funzione che al suo interno fa uso di variabili con lo stesso nome, queste ultime non si riferiscono alla prima funzione. Si osservi l'esempio del listato seguente.

Listato 66.126. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/bZoOp7vp* , *http://ideone.com/ZpYU4* .

```
#include <stdio.h>

int x = 100;

int f (void)
{
    return x;
}

int main (int argc, char *argv[])
{
    int x = 7;
    printf ("x == %i\n", x);
    printf ("f() == %i\n", f());
    return 0;
}
```


Avviando questo programma si ottiene il testo seguente:

```
x == 7
f() == 100
```

In pratica, la funzione ***f()*** che utilizza la variabile ***x***, si riferisce alla variabile con quel nome, dichiarata esternamente alle funzioni, che risulta inizializzata con il valore 100, ignorando perfettamente che la funzione ***main()*** la sta chiamando mentre gestisce una propria variabile automatica con lo stesso nome. Pertanto, la variabile automatica ***x*** della funzione ***main()*** non è visibile alle funzioni che questa chiama a sua volta.

D'altra parte, anche se la variabile automatica ***x*** non risulta visibile, il suo contenuto può essere accessibile, dal momento della sua dichiarazione fino alla fine della funzione (ma questo richiede l'uso di puntatori, come descritto nella sezione 66.5). Alla fine dell'esecuzione della funzione, tutte le sue variabili automatiche perdono la propria identità, in quanto scaricate dalla pila dei dati, e il loro spazio di memoria può essere utilizzato per altri dati (per altre variabili automatiche di altre funzioni).

Si osservi che lo stesso risultato si otterrebbe anche se la variabile ***x*** della funzione ***main()*** fosse dichiarata come statica:

```
…
int main (int argc, char *argv[])
{
    static int x = 7;
    printf ("x == %i\n", x);

    printf ("f() == %i\n", f());
    return 0;
}
```

Le variabili statiche, siano esse dichiarate al di fuori o all'interno delle funzioni, hanno in comune il fatto che utilizzano la memoria dal principio alla fine del funzionamento del programma, anche se dal punto di vista del programma stesso non sono sempre visibili. Pertanto, il loro spazio di memoria sarebbe sempre accessibile, anche se sono oscurate temporaneamente o se ci si trova fuori dal loro campo di azione, attraverso l'uso di puntatori. Naturalmente, il buon senso richiede di mettere la dichiarazione di variabili statiche al di fuori delle funzioni, se queste devono essere manipolate da più di una di queste.

Le variabili che utilizzano memoria dal principio alla fine dell'esecuzione del programma, ma non sono statiche, sono quelle variabili dichiarate all'esterno delle funzioni, per le quali il compilatore predispone un simbolo che consenta la loro identificazione nel file-oggetto. Il fatto di non essere statiche (ovvero il fatto di guadagnare un simbolo di riconoscimento nel file-oggetto) consente loro di essere condivise tra più file (intesi come unità di traduzione), ma per il resto valgono sostanzialmente le stesse regole di visibilità. Il buon senso stesso fa capire che tali variabili possano essere dichiarate solo esternamente alle funzioni, perché dentro le funzioni si usa prevalentemente la pila dei dati e perché comunque, ciò che è dichiarato dentro la funzione deve avere una visibilità limitata.

66.3.8 Compilazione di un progetto composto da più file

Viene riproposto l'esempio utilizzato più volte in questo capitolo, nella sua versione per due file, completandolo con una funzione ***main()***, in modo da poterlo compilare e dimostrare i passaggi necessari in situazioni del genere.

Listato 66.129. File ‘a.c’.

```
#include <stdio.h>

int f (int);
int i;

int g (void)
{
    i++;
    return f (i);
}

int main (void)
{
    printf ("valore originale di i = %i, ", i);
    printf ("valore restituito da g() = %i\n", g());
    printf ("valore originale di i = %i, ", i);
    printf ("valore restituito da g() = %i\n", g());
    printf ("valore originale di i = %i, ", i);
    printf ("valore restituito da g() = %i\n", g());
    printf ("valore originale di i = %i, ", i);
    printf ("valore restituito da g() = %i\n", g());
    return 0;
}
```

Listato 66.130. File ‘b.c’.

```
int f (int x)
{
    return (x * x);
}

int i = 1;
```

Disponendo di più file sorgenti separati, la compilazione avviene in due fasi: la generazione dei file oggetto e il «collegamento» (*link*) di questi in modo da ottenere un file eseguibile. Fortunatamente, tutto questo può essere gestito tramite lo stesso compilatore ‘**cc**’.

Per generare i file oggetto si utilizza ‘**cc**’ con l’opzione ‘**-c**’; se si può disporre del compilatore GNU C, è meglio aggiungere anche l’opzione ‘**-Wall**’. Si suppone che il primo file sia stato nominato ‘a.c’ e il secondo ‘b.c’. Si inizia dalla compilazione dei singoli file in modo da generare i file oggetto ‘a.o’ e ‘b.o’.

```
$ cc -Wall -c a.c [Invio]

$ cc -Wall -c b.c [Invio]
```

Quindi si passa all’unione dei due risolvendo i riferimenti incrociati, generando il file eseguibile ‘**prova**’.

```
$ cc -o prova a.o b.o [Invio]
```

Ecco cosa si dovrebbe vedere eseguendo il file che si ottiene dalla compilazione:

```
$ ./prova [Invio]

  valore originale di i = 1, valore restituito da g() = 4
  valore originale di i = 2, valore restituito da g() = 9
  valore originale di i = 3, valore restituito da g() = 16
  valore originale di i = 4, valore restituito da g() = 25
```

Per un uso migliore del compilatore si veda la parte 65.

66.3.9 Osservazioni sulla vita delle costanti letterali

Una costante letterale può essere gestita dal compilatore come meglio crede, ma quando si tratta di un’informazione che non può risiedere completamente in una parola del microprocessore e non si può collocare in un’istruzione del linguaggio macchina, è evidente che debba essere conservata nella memoria usata dal programma. Si osservi l’esempio seguente:

```
void f (void)
{
    char x[] = "ciao amore";
    printf ("%s\n", x);
}
```

L’array ***x[]***, o meglio, il puntatore che lo rappresenta, viene creato ogni volta alla chiamata della funzione ***f()*** e anche distrutto alla sua conclusione. Ma questo array viene inizializzato ogni volta con una

stringa prestabilita, la quale deve essere disponibile per tutto il tempo di funzionamento del programma. In altri termini, quella stringa è un array senza nome allocato in memoria dal principio dell’esecuzione del programma, pertanto al di fuori della pila dei dati.

66.3.10 Libreria standard e file di intestazione

La libreria standard del linguaggio C prevede la disponibilità di una serie di funzioni, macro del precompilatore e tipi di dati per usi specifici.

Dal punto di vista del programmatore, si ha la percezione della presenza di questa libreria attraverso l’inclusione dei «file di intestazione», ovvero di quei file che per tradizione hanno un nome che finisce per ‘.h’ e si incorporano attraverso le direttive ‘**#include**’ del precompilatore. Tuttavia, di norma le funzioni della libreria standard sono contenute in un file-oggetto già compilato (che può essere realizzato in forma differente, a seconda che serva per l’accesso dinamico alle funzioni, oppure che debba essere incorporato nel file eseguibile finale, come spiegato nella sezione 65.7), noto come libreria C, o solo Libc, che viene incluso automaticamente nella compilazione di un progetto, a meno di escluderlo espressamente.

Con il compilatore GNU C, per escludere l’utilizzo di qualunque libreria predefinita vanno usate le opzioni ‘**-nostartfiles**’ e ‘**-nodefaultlibs**’; eventualmente l’opzione ‘**-nostdlibs**’ dovrebbe valere per entrambe queste opzioni e può essere usata assieme a loro, benché sia ridondante.

Anche se la libreria C viene realizzata nel modo descritto, il concetto di libreria standard non si esaurisce nei file-oggetto che contengono le sue funzioni, perché rimane la necessità di dichiarare le macro del precompilatore, i tipi di dati che fanno parte dello standard complessivo, ma soprattutto i prototipi delle funzioni che compongono la libreria. Pertanto, i file di intestazione rimangono indispensabili e fanno parte integrante della libreria.

A titolo dimostrativo, si può osservare il programma seguente che, pur facendo uso della libreria standard, in quanto si sfrutta la funzione ***printf()***, non incorpora alcun file di intestazione. In tal caso, però, è indispensabile dichiarare il prototipo della funzione utilizzata:

```
extern int printf (const char *format,...);

int main (void)
{
    printf ("Ciao a tutti!\n");
    return 0;
}
```

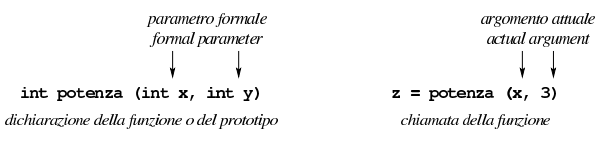
66.4 Annotazioni sulla terminologia

I documenti che descrivono lo standard del linguaggio C utilizzano una terminologia specifica. Qui si descrivono alcuni di quei termini con delle annotazioni riguardo al contesto a cui si riferiscono.

66.4.1 Parametri e argomenti

Generalmente, i termini «argomento» e «parametro», riferiti alle funzioni o alle procedure dei linguaggi di programmazione, vengono usati in modo intercambiabile, benché si intuisca una differenza tra i due. Lo standard C chiarisce l’ambito corretto di utilizzo per entrambi: i valori annotati in una chiamata di funzione sono gli argomenti attuali; le variabili che descrivono formalmente ciò che una funzione deve ricevere dall’esterno sono i parametri formali.

Figura 66.134. Distinzione tra parametri e argomenti.



66.4.2 Byte e caratteri

Secondo il linguaggio C, il byte è l'unità di memorizzazione più piccola che possa essere utilizzata per contenere un carattere, tra quelli dell'insieme minimo. Pertanto, per definizione, il tipo '**char**' (indifferentemente se con o senza segno) occupa esattamente un byte.

In pratica, per il linguaggio C il byte non è necessariamente un insieme di otto bit, anche se di norma questa corrispondenza è valida.

Va considerato anche che il tipo '**char**', senza altre indicazioni, può essere inteso come valore con segno o senza segno, a seconda della piattaforma. Tuttavia, come punto fermo, l'insieme di caratteri minimo deve essere rappresentabile con valori positivi. In pratica, di norma questo insieme minimo di caratteri corrisponde alla codifica ASCII, la quale si rappresenta completamente con 7 bit, pertanto l'ottavo bit di un byte standard potrebbe essere usato come segno, senza interferire con l'interpretazione corretta dei caratteri. In altri termini, per utilizzare il tipo '**char**' in modo compatibile da una piattaforma all'altra, questo va considerato solo per i valori utili alla rappresentazione dell'insieme di caratteri minimo, con i quali si ha la certezza di avere a che fare sempre solo con valori positivi.

66.4.3 Unità di traduzione

Il file generato dal precompilatore, formato normalmente dall'incorporazione di diversi file, viene definito una ***unità di traduzione***. Il concetto di «traduzione» deriva dal fatto che il precompilatore, oltre a incorporare altri file, traduce le macro-variabili e le macroistruzioni espandendole secondo la loro dichiarazione; pertanto, i file sorgenti originali subiscono una prima trasformazione che produce il codice C vero e proprio.

Quando si fa riferimento al campo di azione delle variabili definite al di fuori delle funzioni, si afferma che questo riguarda l'ambito del file. In tal caso, per file si intende l'unità di traduzione.

66.4.4 «Linkage»

Quando si fa riferimento a variabili o funzioni che sono dichiarate esternamente a tutte le funzioni, il campo di azione è legato al file (nel senso di unità di traduzione), essendo accessibili solo a partire dalla dichiarazione stessa. Quando si combinano assieme più file attraverso il meccanismo del «collegamento» (*link*), il programma che esegue questo compito tratta i nomi uguali di variabili e di funzioni nel senso di un riferimento alla stessa cosa (la stessa variabile o la stessa funzione). Quando una variabile o una funzione è dichiarata in modo tale da consentire questo collegamento, si ha un ***linkage esterno***; quando la dichiarazione è tale da impedirlo (con lo specificatore di classe di memorizzazione '**static**'), si ha un ***linkage interno***.

Si può rendere esplicito che una variabile o una funzione sono da cercarsi al di fuori del proprio file, oppure in una posizione più avanzata dello stesso file, richiedendo un *linkage esterno* con lo specificatore di classe di memorizzazione '**extern**'. In tal caso, si può collegare esternamente anche una variabile indicata all'interno di una funzione o di un altro tipo di blocco, sempre con lo specificatore '**extern**'.

Le variabili che, diversamente, sono dichiarate all'interno di un blocco di qualunque genere, non sono collegabili, soprattutto nel caso delle variabili automatiche, la cui vita dipende dal blocco in cui sono contenute.

66.4.5 Durata di memorizzazione

Nella documentazione standard si usa spesso il termine *storage duration*, ovvero ***durata di memorizzazione***, per fare riferimento al tempo di vita di una certa informazione contenuta in memoria.

Di norma si possono distinguere due casi fondamentali: ciò che viene memorizzato in un'area di memoria sempre disponibile (anche se non è detto che a ogni parte del programma sia consentito di accedervi) e ciò che si mette nella pila dei dati. Nel primo caso di parla di *static storage duration*, in quanto i dati stanno lì e non si muovono; nel secondo si parla di *automatic storage duration*, in quanto la memoria della pila viene liberata e riutilizzata in modo dinamico.

È per questa ragione che, nella dichiarazione delle variabili all'interno delle funzioni, esiste lo specificatore di classe '**static**', a indicare una variabile che, pur essendo accessibile solo all'interno della funzione, va collocata al di fuori della pila dei dati, in modo da conservare il proprio contenuto durante le chiamate successive della stessa funzione.

66.4.6 «Lvalue» e «rvalue»

Nello standard del linguaggio C, il termine *lvalue* indica, approssimativamente, ciò che appare a sinistra di un operatore di assegnamento, nelle condizioni per cui ciò è ammissibile. Per esempio, nell'espressione seguente, la variabile ***x*** rappresenta un *lvalue*:

```
x = 3;
```

L'espressione seguente, invece, **non è valida**, perché la costante '**3**' non può essere un *lvalue*:

```
3 = x;  // Non è valida, perché «3» non è un «lvalue».
```

Il termine poteva significare, originariamente, *left-value*, da contrapporsi a un possibile *right-value*, costituito da ciò che in un'espressione si trova alla destra dell'operatore di assegnamento. Tuttavia, lo standard attuale definisce la sigla in questione un *location value*, ovvero un'espressione che si riferisce a un'area di memorizzazione.

Un'espressione che sia un *lvalue* deve anche consentire la lettura dell'area di memorizzazione a cui si riferisce; pertanto, ciò che è un *lvalue* deve poter essere usato alla destra di un operatore di assegnamento (in qualità di *rvalue*). D'altra parte, non è garantito che un *lvalue* individui sempre un'area di memorizzazione modificabile, dal momento che esistono variabili qualificate come costanti, alle quali si assegna un valore in fase di dichiarazione, ma successivamente non è più consentita la modifica. Per distinguere anche questa situazione, volendo escludere il caso delle costanti, si specifica che l'espressione *lvalue* deve anche essere modificabile.

Tabella 66.137. Operatori che richiedono un operando di tipo *lvalue*. In tutti i casi, escluso '***&lvalue***', deve trattarsi di un ***lvalue*** modificabile in quel contesto.

Parola chiave	Descrizione
&lvalue	Indirizzo di ***lvalue***.
++lvalue *lvalue++* *--lvalue* *lvalue--*	Incremento e decremento.
lvalue=rvalue *lvalue+=rvalue* *lvalue-=rvalue* *lvalue*=rvalue* *lvalue%=rvalue* *lvalue<<=rvalue* *lvalue>>=rvalue* *lvalue&=rvalue* *lvalue^=rvalue* *lvalue|=rvalue* *lvalue~=rvalue*	Assegnamenti.

Attualmente, lo standard C, al posto di *rvalue*, preferisce esprimere il concetto come «valore di un'espressione».

66.4.7 «Digraph» e «Trigraph»

In varie situazioni lo standard C consente l'utilizzo di sequenze speciali di caratteri, in sostituzione di simboli che in certi contesti potrebbero mancare, essendo invece indispensabili. In generale, quando per la scrittura dei file sorgenti si può contare su un insieme di caratteri pari a quello della codifica ASCII, queste sequenze speciali non vanno usate assolutamente, perché complicano terribilmente la lettura dei file. A ogni modo, conviene essere a conoscenza della loro esistenza e del significato che assumono.

Digraph	*Trigraph*	Carattere corrispondente
`<:`	`??(`	`[`
`:>`	`??)`	`]`
`<%`	`??<`	`{`
`%>`	`??>`	`}`
`%:`	`??=`	`#`
`%:%:`	`??=??=`	`##`
	`??!`	`|`
	`??'`	`^`
	`??/`	`\`
	`??-`	`~`

66.5 Puntatori, array, stringhe e allocazione dinamica della memoria

All'inizio del capitolo sono stati mostrati solo i tipi di dati più semplici. Per poter utilizzare gli array si gestiscono dei puntatori alle zone di memoria contenenti tali strutture.

Quando si ha a che fare con i puntatori è importante considerare che il modello di memoria che si ha di fronte è un'astrazione, nel senso che una struttura di dati appare idealmente continua, mentre nella realtà il compilatore potrebbe anche provvedere a scomporla in blocchi separati.

Nella spiegazione che si fa qui, come nelle altre sezioni del capitolo, l'esposizione è semplificata rispetto alle definizioni dello standard; pertanto, per un approccio più preciso ci si deve rivolgere ai documenti ufficiali sul linguaggio C.

66.5.1 Espressioni a cui si assegnano dei valori

Quando si utilizza un operatore di assegnamento, come '**=**' o altri operatori composti, ciò che si mette alla sinistra rappresenta la «variabile ricevente» del risultato dell'espressione che si trova alla destra dell'operatore (nel caso di operatori di assegnamento composti, l'espressione alla destra va considerata come quella che si ottiene scomponendo l'operatore). Ma il linguaggio C consente di rappresentare quella «variabile ricevente» attraverso un'espressione, come nel caso dei puntatori che vengono descritti in questo capitolo. Pertanto, per evitare confusione, la documentazione dello standard chiama l'espressione a sinistra dell'operatore di assegnamento un *lvalue* (*Left value* o *Location value*).

Nel capitolo si evita questa terminologia, tuttavia è importante comprendere che un'espressione può rappresentare una «variabile», pur senza averle dato un nome (nella sezione 66.4.6 il concetto di *lvalue* e di *rvalue* viene descritto con migliore dettaglio).

66.5.2 Puntatori

Una variabile, di qualunque tipo sia, rappresenta normalmente un valore posto da qualche parte nella memoria del sistema.[19] Quando si usano i tipi di dati normali, è il compilatore a prendersi cura di tradurre i riferimenti agli spazi di memoria rappresentati simbolicamente attraverso dei nomi.

Attraverso l'operatore di indirizzamento e-commerciale ('**&**'), è possibile ottenere il puntatore (riferito alla rappresentazione ideale di memoria del linguaggio C) a una variabile «normale». Tale valore può essere inserito in una variabile particolare, adatta a contenerlo: una ***variabile puntatore***.

Per esempio, se ***p*** è una variabile puntatore adatta a contenere l'indirizzo di un intero, l'esempio mostra in che modo assegnare a tale variabile il puntatore alla variabile ***i***:

```
int i = 10;
…
p = &i;  // L'indirizzo di «i» viene assegnato al
         // puntatore «p».
```

La dichiarazione di una variabile puntatore avviene in modo simile a quello delle variabili normali, con l'aggiunta di un asterisco prima del nome. L'esempio seguente dichiara la variabile ***p*** come puntatore a un tipo '**int**'. Si osservi che va indicato il tipo di dati a cui si punta, perché questa informazione è parte integrante del puntatore.

```
int *p;
```

Non deve essere interesse del programmatore il modo esatto in cui si rappresentano i puntatori dei vari tipi di dati, diversamente non ci sarebbe l'utilità di usare un linguaggio come il C invece di un semplice assemblatore di linguaggio macchina.

Una volta dichiarata la variabile puntatore, questa viene utilizzata normalmente, senza asterisco, finché si intende fare riferimento al puntatore stesso.

L'asterisco usato nella dichiarazione serve a definire il tipo di dati, quindi, '**int *p**' rappresenta la dichiarazione della variabile ***p*** di tipo '**int ***'. Tuttavia si può fare un ragionamento leggermente differente, con l'aiuto delle parentesi: '**int (*p)**' è la dichiarazione di una zona di memoria senza nome, di tipo '**int**', a cui punta la variabile ***p*** attraverso la dereferenziazione ***p**. Le due cose sono equivalenti, in quanto portano comunque alla creazione della variabile ***p*** di tipo puntatore a intero, ma la seconda forma consente di comprendere, successivamente, la sintassi per la creazione di un puntatore a funzione.

È importante chiarire subito in che modo si dichiarano più variabili puntatore con una sola istruzione; si osservi l'esempio seguente in cui si creano le variabili ***p*** e ***p2***, in particolare per il fatto che l'asterisco va ripetuto:

```
int *p, *p2;
```

Attraverso l'operatore di «dereferenziazione», l'asterisco ('*****'), è possibile accedere alla zona di memoria a cui la variabile punta. Per «dereferenziare» si intende quindi l'azione con cui si toglie il riferimento e si raggiungono i dati a cui un puntatore si riferisce.[20]

Attenzione a non fare confusione con gli asterischi: una cosa è quello usato per dichiarare o per dereferenziare un puntatore e un'altra è l'operatore con cui invece si ottiene la moltiplicazione.

L'esempio già accennato potrebbe essere chiarito nel modo seguente, dove si mostra anche la dichiarazione della variabile puntatore:

```
int i = 10;
int *p;
…
p = &i;
```

A questo punto, dopo aver assegnato a ***p*** il puntatore alla variabile ***i***, è possibile accedere alla stessa area di memoria in due modi diversi: attraverso la variabile ***i***, oppure attraverso la dereferenziazione di ***p***, ovvero la traduzione ****p***.

```
int i = 10;
int *p;
…
p = &i;
…
*p = 20;
```

Nell'esempio, l'istruzione '***p=20**' è tecnicamente equivalente a '**i=20**'. Per chiarire un po' meglio il ruolo delle variabili puntatore, si può complicare l'esempio nel modo seguente:

```
int i = 10;
int *p;
int *p2;
…
p = &i;
…
p2 = p;
…
*p2 = 20;
```

In particolare è stata aggiunta una seconda variabile puntatore, ***p2***, solo per fare vedere che è possibile passare un puntatore anche ad altre variabili senza dover usare l'asterisco. Comunque, in questo caso, '***p2=20**' è tecnicamente equivalente sia a '***p=20**', sia a '**i=20**'.

Si osservi che l'asterisco è un operatore che, evidentemente, ha la precedenza rispetto a quelli di assegnamento. Eventualmente, anche in questo caso si possono usare le parentesi per togliere ambiguità al codice:

```
int i = 10;
int *p;
…
p = &i;
…
(*p2) = 20;
```

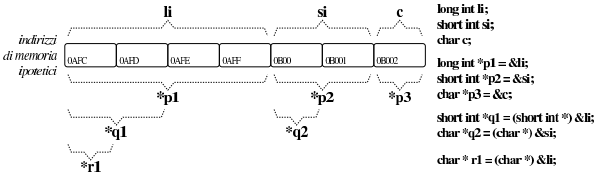
Come accennato inizialmente, il tipo di dati a cui un puntatore si rivolge, fa parte integrante del puntatore stesso. Ciò è importante perché quando si dereferenzia un puntatore occorre sapere quanto è grande l'area di memoria a cui si deve accedere a partire dal puntatore. Per questa ragione, quando si assegna a una variabile puntatore un altro puntatore, questo deve essere compatibile, nel senso che deve riferirsi allo stesso tipo di dati, altrimenti si rischia di ottenere un risultato inatteso. A questo proposito, l'esempio seguente contiene probabilmente un errore:

```
char *pc;
int  *pi;
…
pi = pc;  // I due puntatori si riferiscono a dati di tipo
          // differente!
…
```

Quando invece si vuole trasformare realmente un puntatore in modo che si riferisca a un tipo di dati differente, si può usare un cast, come si farebbe per convertire i valori numerici:

```
char *pc;
int  *pi;
…
pi = (int *) pc;  // Il programmatore dimostra di essere
                  // consapevole di ciò che sta facendo
                  // attraverso un cast!
…
…
```

Nello schema seguente appare un esempio che dovrebbe consentire di comprendere la differenza che c'è tra i puntatori, in base al tipo di dati a cui fanno riferimento. In particolare, ***p1***, ***q1*** e ***r1*** fanno tutti riferimento all'indirizzo ipotetico $0AFC_{16}$, ma l'area di memoria che considerano è diversa, pertanto ****p1***, ****q1*** e ****r1*** sono tra loro «variabili» differenti, anche se si sovrappongono parzialmente.

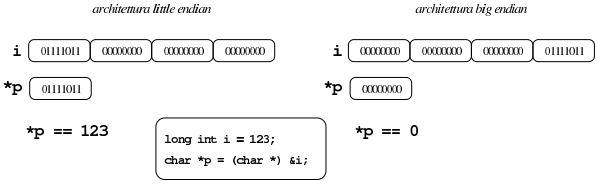


L'esempio seguente rappresenta un programma completo che ha lo scopo di determinare se l'architettura dell'elaboratore è di tipo *big endian* o di tipo *little endian*. Per capirlo si dichiara una variabile di tipo '**long int**' che si intende debba essere di rango superiore rispetto al tipo '**char**', assegnandole un valore abbastanza basso da poter essere rappresentato anche in un tipo '**char**' senza segno. Con un puntatore di tipo '**char ***' si vuole accedere all'inizio della variabile contenente il numero intero '**long int**': se già nella porzione letta attraverso il puntatore al primo «carattere» si trova il valore assegnato alla variabile di tipo intero, vuol dire che i byte sono invertiti e si ha un'architettura *little endian*, mentre diversamente si presume che sia un'architettura *big endian*.

Listato 66.149. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/Abe4VgIo* , *http://ideone.com/WVpmK* .

```
#include <stdio.h>
int main (void)
{
    long int i = 123;
    char *p = (char *) &i;
    if (*p == 123)
      {
        printf ("little endian\n");
      }
    else
      {
        printf ("big endian\n");
      }
    return 0;
}
```

Figura 66.150. Schematizzazione dell'operato del programma di esempio, per determinare l'ordine dei byte usato nella propria architettura.



Il linguaggio C utilizza il passaggio degli argomenti alle funzioni per valore; per ottenere il passaggio per riferimento occorre utilizzare dei puntatori. Si immagini di volere realizzare una funzione banale che modifica la variabile utilizzata nella chiamata, sommandovi una quantità fissa. Invece di passare il valore della variabile da modificare, si può passare il suo puntatore; in questo modo la funzione (che comunque deve essere stata realizzata appositamente per questo scopo) agisce nell'area di memoria a cui punta questo puntatore.

```
…
void funzione_stupida (int *x)
{
    (*x)++;
}
…
int main (void)
{
    int y = 10;
    …
    funzione_stupida (&y);
    …
    return 0;
}
```

L'esempio mostra la dichiarazione e descrizione di una funzione che

non restituisce alcun valore e ha un parametro costituito da un puntatore a un intero. Il lavoro della funzione è solo quello di incrementare il valore contenuto nell'area di memoria a cui si riferisce tale puntatore.

Poco dopo, nella funzione ***main()*** inizia il programma vero e proprio; viene dichiarata la variabile ***y*** corrispondente a un intero normale inizializzato a 10, poi, a un certo punto viene chiamata la funzione vista prima, passando il puntatore a ***y***.

Il risultato è che dopo la chiamata, la variabile ***y*** contiene il valore precedente incrementato di un'unità.

Quando si usano i puntatori, invece delle variabili comuni, occorre considerare che se la vita della variabile a cui un puntatore fa riferimento si è esaurita, il puntatore relativo diventa privo di valore. Questo significa che il fatto di avere conservato il puntatore a una certa area di memoria **non** implica automaticamente la garanzia che tale zona contenga dati validi o che sia ancora raggiungibile.

66.5.3 Array

Nel linguaggio C, l'array è una sequenza ordinata di elementi dello stesso tipo nella rappresentazione ideale di memoria di cui si dispone. In questo senso, quando si dichiara un array, quello che il programmatore ottiene in pratica è il riferimento alla posizione iniziale di questo, mentre gli elementi successivi si raggiungono tenendo conto della lunghezza di ogni elemento.

Questo ragionamento vale in senso generale ed è un po' approssimativo. In contesti particolari, il riferimento a un array restituisce qualcosa di diverso dal puntatore al primo elemento.

Visto in questi termini, si può intendere che l'array in C è sempre a una sola dimensione, tutti gli elementi devono essere dello stesso tipo in modo da avere la stessa lunghezza e la quantità degli elementi, una volta definita, è fissa.

È compito del programmatore ricordare la quantità di elementi che compone l'array, perché determinarlo diversamente è complicato e a volte non è possibile. Inoltre, quando un programma tenta di accedere a una posizione oltre il limite degli elementi esistenti, c'è il rischio che non si verifichi alcun errore, arrivando però a dei risultati imprevedibili.

Lo standard prescrive che sia consentito raggiungere l'indirizzo successivo all'ultimo elemento, anche se tale contenuto diventa privo di significato. Ciò serve a garantire che non si provochino errori nell'accesso alla memoria, se l'indice va oltre il limite di un array, ma per una sola posizione, per leggere un contenuto privo di utilità. In pratica, ciò significa che dopo un array ci deve essere qualunque altra variabile, o al limite uno spazio inutilizzato. Ma questo è compito del compilatore.

La dichiarazione di un array avviene in modo intuitivo, definendo il tipo degli elementi e la loro quantità. L'esempio seguente mostra la dichiarazione dell'array ***a*** di sette elementi di tipo '**`int`**':

```
int a[7];
```

Per accedere agli elementi dell'array si utilizza un indice, il cui valore iniziale è sempre zero e, di conseguenza, quello con cui si raggiunge l'elemento ***n***-esimo deve avere il valore ***n***−1. L'esempio seguente mostra l'assegnamento del valore 123 al **secondo** elemento:

```
a[1] = 123;
```

In presenza di array monodimensionali che hanno una quantità ridotta di elementi, può essere sensato attribuire un insieme di valori iniziale all'atto della dichiarazione.

Alcuni compilatori consentono l'inizializzazione degli array solo quando questi sono dichiarati all'esterno delle funzioni, con un campo di azione globale, oppure all'interno delle funzioni, ma dichiarati come «statici», nel senso che continuano a esistere all'uscita della funzione.

```
int a[] = {123, 453, 2, 67};
```

L'esempio mostrato dovrebbe chiarire in che modo si possono dichiarare gli elementi dell'array, tra parentesi graffe, togliendo così la necessità di specificare la quantità di elementi. Tuttavia, le due cose possono coesistere:

```
int a[10] = {123, 453, 2, 67};
```

In tal caso, l'array si compone di 10 elementi, di cui i primi quattro con valori prestabiliti, mentre gli altri ottengono il valore zero. Si osservi però che il contrario non può essere fatto:

```
int a[5] = {123, 453, 2, 67, 32, 56, 78};      // Non si può!
```

Gli standard recenti del linguaggio C consentono anche la dichiarazione di array per i quali il compilatore non può sapere subito la quantità di elementi da predisporre, **purché ciò avvenga nel campo di azione delle funzioni** (o di blocchi inferiori). In pratica, in questi casi è possibile indicare la quantità di elementi attraverso un'espressione che si traduca in un numero intero, come nell'esempio seguente, dove la quantità di elementi è data dal prodotto tra la variabile ***s*** e la costante 3:

```
int s = 33;
…
int a[s * 3];
```

Gli array dichiarati al di fuori delle funzioni (quelli il cui campo di azione è legato al file) e quelli che, pur essendo dichiarati nelle funzioni, continuano a esistere per tutto il tempo di esecuzione del programma (in quanto «statici»), possono avere soltanto una quantità di elementi già stabilita in fase di compilazione. Per fare riferimento a array definiti in altri file, oppure in posizioni più avanzate dello stesso file, è possibile usare una dichiarazione «esterna», nella quale è bene specificare la quantità di elementi, ma questa deve essere coerente con quella della dichiarazione a cui si fa riferimento:

```
extern int i[3];
…
int i[3];
```

In alternativa si può fare una dichiarazione esterna di un array senza specificarne la quantità di elementi, ma questo implica che, fino a quando non appare la dichiarazione completa, l'array sia di tipo incompleto e non si possa determinare la sua dimensione con l'aiuto dell'operatore '**`sizeof`**':

```
extern int i[];         // Tipo incompleto.
…
int i[3];
```

La scansione di un array avviene generalmente attraverso un'iterazione enumerativa, in pratica con un ciclo '**`for`**' che si presta particolarmente per questo scopo. Si osservi l'esempio seguente:

```
int a[7];
int i;
…
for (i = 0; i < 7; i++)
  {
    …
    a[i] = …;
    …
  }
```

L'indice ***i*** viene inizializzato a zero, in modo da cominciare dal primo elemento dell'array; il ciclo può continuare fino a che ***i*** continua a essere inferiore a sette, infatti l'ultimo elemento dell'array ha indice sei; alla fine di ogni ciclo, prima che riprenda il successivo, viene incrementato l'indice di un'unità.

Per scandire un array in senso opposto, si può agire in modo analogo, come nell'esempio seguente:

```
int a[7];
int i;
…
for (i = 6; i >= 0; i--)
  {
    …
    a[i] = …;
    …
  }
```

Questa volta l'indice viene inizializzato in modo da puntare alla posizione finale; il ciclo viene ripetuto fino a che l'indice è maggiore o uguale a zero; alla fine di ogni ciclo, l'indice viene decrementato di un'unità.

Se non si può conoscere la dimensione dell'array, questa deve essere calcolata con l'ausilio dell'operatore '**sizeof**', come nell'esempio seguente, ammesso che il contesto sia tale da consentire all'operatore di restituire un valore valido:

```
// Da qualche parte si dichiara il valore di «x» come numero
// intero.
…
int a[7 * x];
int i;
…
int s = (sizeof a) / (sizeof (a[0]));
for (i = 0; i < s; i++)
  {
    …
    a[i] = …;
    …
  }
```

Il calcolo della quantità di elementi è ottenuto determinando la dimensione dell'array in byte e dividendo tale valore per la dimensione in byte di un intero, ovvero per la dimensione di ogni elemento dell'array stesso.

Quando un array è argomento dell'operatore '**sizeof**', si ottiene la dimensione complessiva dell'array stesso (nell'unità gestita da '**sizeof**'. Tuttavia occorre considerare che, se l'array non è ancora stato definito nella sua dimensione, non si può avere il risultato atteso.

66.5.4 Array multidimensionali

Gli array in C sono monodimensionali, però nulla vieta di creare un array i cui elementi siano array tutti uguali. Per esempio, nel modo seguente, si dichiara un array di cinque elementi che a loro volta sono insiemi di sette elementi di tipo '**int**'. Nello stesso modo si possono definire array con più di due dimensioni.

```
int a[5][7];
```

L'esempio seguente mostra il modo normale di scandire un array a due dimensioni:

```
int a[5][7];
int i;
int j;
…
for (i = 0; i < 5; i++)
  {
    …
    for  (j = 0; j < 7; j++)
      {
        …
        a[i][j] = …;
        …
      }
    …
  }
```

Anche se in pratica un array a più dimensioni è solo un array «normale» in cui si individuano dei sottogruppi di elementi, la scansione

deve avvenire sempre indicando formalmente lo stesso numero di elementi prestabiliti per le dimensioni rispettive, anche se dovrebbe essere possibile attuare qualche trucco. Per esempio, tornando al listato mostrato, se si vuole scandire in modo continuo l'array, ma usando un solo indice, bisogna farlo gestendo l'ultimo:

```
int a[5][7][9];
int j;
…
for (j = 0; j < (5 * 7 * 9); j++)
  {
    …
    a[0][0][j] = …;
    …
  }
```

Rimane comunque da osservare il fatto che questo non sia un bel modo di programmare.

Anche gli array a più dimensioni possono essere inizializzati, secondo una modalità analoga a quella usata per una sola dimensione, con la differenza che l'informazione sulla quantità di elementi per dimensione non può essere omessa. L'esempio seguente è un programma completo, in cui si dichiara e inizializza un array a due dimensioni, per poi mostrarne il contenuto:

Listato 66.166. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/Ht9r6QwN* , *http://ideone.com/xzD7f* .

```
#include <stdio.h>

int main (int argc, char *argv[])
{
  int a[3][4] = {{1,  2,  3,  4},
                 {5,  6,  7,  8},
                 {9, 10, 11, 12}};
  int i, j;

  for (i = 0; i < 3; i++)
    {
      for (j = 0; j < 4; j++)
        {
          printf ("a[%i][%i]=%i\t", i, j, a[i][j]);
        }
      printf ("\n");
    }

  return 0;
}
```

Il programma dovrebbe mostrare il testo seguente:

```
a[0][0]=1       a[0][1]=2       a[0][2]=3       a[0][3]=4
a[1][0]=5       a[1][1]=6       a[1][2]=7       a[1][3]=8
a[2][0]=9       a[2][1]=10      a[2][2]=11      a[2][3]=12
```

Anche nell'inizializzazione di un array a più dimensioni si possono omettere degli elementi, come nell'estratto seguente:

```
…
  int a[3][4] = {{1, 2},
                 {5, 6, 7, 8}};
…
```

In tal caso, il programma si mostrerebbe così:

```
a[0][0]=1       a[0][1]=2       a[0][2]=0       a[0][3]=0
a[1][0]=5       a[1][1]=6       a[1][2]=7       a[1][3]=8
a[2][0]=0       a[2][1]=0       a[2][2]=0       a[2][3]=0
```

Di certo, pur sapendo di voler utilizzare un array a più dimensioni, si potrebbe pretendere di inizializzarlo come se fosse a una sola, come nell'esempio seguente, ma il compilatore dovrebbe avvisare del fatto:

```
…
  int a[3][4] = {1, 2, 3, 4, 5, 6,            // Così non è
                 7, 8, 9, 10, 11, 12};        // grazioso.
…
```

66.5.5 Natura dell'array

Inizialmente si è accennato al fatto che quando si crea un array, quello che viene restituito in pratica è un puntatore alla sua posizione iniziale, ovvero all'indirizzo del primo elemento di questo. Si può intuire che non sia possibile assegnare a un array un altro array, anche se ciò potrebbe avere significato. Al massimo si può assegnare elemento per elemento.

Per evitare errori del programmatore, la variabile che contiene l'indirizzo iniziale dell'array, quella che in pratica rappresenta l'array stesso, è in **sola lettura**. Quindi, nel caso dell'array già visto, la variabile ***a*** non può essere modificata, mentre i singoli elementi ***a[i]*** sì:

```
int a[7];
```

Data la filosofia del linguaggio C, se fosse possibile assegnare un valore alla variabile ***a***, si modificherebbe il puntatore, facendo in modo che questo punti a un array differente. Ma per raggiungere questo risultato vanno usati i puntatori in modo esplicito. Si osservi l'esempio seguente.

Listato 66.172. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/SLI8GS82* , *http://ideone.com/RImyk* .

```
#include <stdio.h>

int main (void)
{
  int a[3];
  int *p;

  p = a;        // «p» diventa un alias dell'array «a».

  p[0] = 10;    // Si può fare solo con gli array
  p[1] = 100;   // a una sola dimensione.
  p[2] = 1000;  //

  printf ("%i %i %i \n",  a[0], a[1], a[2]);

  return 0;
}
```

Viene creato un array, ***a***, di tre elementi di tipo '**int**', e subito dopo una variabile puntatore, ***p***, al tipo '**int**'. Si assegna quindi alla variabile ***p*** il puntatore rappresentato da ***a***; da quel momento si può fare riferimento all'array indifferentemente con il nome ***a*** o ***p***.

Si può osservare anche che l'operatore '**&**', seguito dal nome di un array, produce ugualmente l'indirizzo dell'array che è equivalente a quello fornito senza l'operatore stesso, con la differenza che riguarda l'array nel suo complesso:

```
…
p = &a;    // I due puntatori non sono dello stesso tipo!
…
```

Pertanto, in questo caso si pone il problema di compatibilità del tipo di puntatore che si può risolvere con un cast esplicito:

```
…
p = (int *) &a;   // «p» diventa un alias dell'array «a».
…
```

In modo analogo, si può estrapolare l'indice che rappresenta l'array dal primo elemento, cosa che si ottiene senza incorrere in problemi di compatibilità tra i puntatori. Si veda la trasformazione dell'esempio nel modo seguente.

Listato 66.175. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/91Df91s1dq* , *http://ideone.com/6rPpE* .

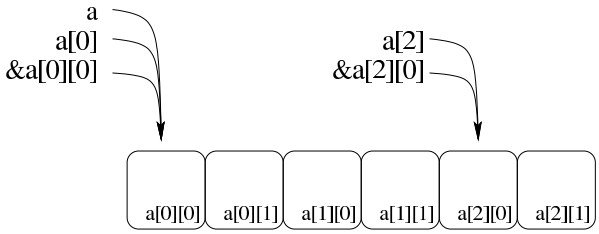
```
#include <stdio.h>

int main (void)
{
  int a[3];
  int *p;

  p = &a[0];    // «p» diventa un alias dell'array «a».

  p[0] = 10;    // Si può fare solo con gli array
  p[1] = 100;   // a una sola dimensione.
  p[2] = 1000;  //

  printf ("%i %i %i \n",  a[0], a[1], a[2]);

  return 0;
}
```

Anche se si può usare un puntatore come se fosse un array, va osservato che la variabile ***p***, in quanto dichiarata come puntatore, viene considerata in modo differente dal compilatore; per esempio non è possibile determinare la dimensione dell'array a cui punta attraverso l'operatore '**sizeof**', perché si otterrebbe semplicemente la quantità di byte che costituisce la variabile puntatore.

Quando si opera con array a più dimensioni, il riferimento a una porzione di array restituisce l'indirizzo della porzione considerata. Per esempio, si supponga di avere dichiarato un array a due dimensioni, nel modo seguente:

```
int a[3][2];
```

Se a un certo punto, in riferimento allo stesso array, si scrivesse '**a[2]**', si otterrebbe l'indirizzo del terzo gruppo di due interi:



Tenendo d'occhio lo schema appena mostrato, considerato che si sta facendo riferimento all'array ***a*** di 3×2 elementi di tipo '**int**', va osservato che:

- in condizioni normali '**a**' si traduce nel puntatore a un array di due elementi di tipo '**int**';
- '**a[0]**' e '**&a[0][0]**' si traducono nel puntatore a un elemento di tipo '**int**' (precisamente il primo);
- '**&a**' si traduce nel puntatore a un array composto da 3×2 elementi di tipo '**int**'.

Pertanto, se questa volta si volesse assegnare a una variabile puntatore di tipo '**int ***' l'indirizzo iniziale dell'array, nell'esempio seguente si creerebbe un problema di compatibilità:

```
…
int a[3][2];
int *p;
p = a;    // I due puntatori non sono dello stesso tipo!
…
```

Pertanto, occorrerebbe riferirsi all'inizio dell'array in modo differente oppure attraverso un cast.

66.5.6 Puntatori costanti

Si può far sì che un puntatore funzioni in modo più simile a quello di un array a una sola dimensione, dichiarando il puntatore come costante, nel senso che il puntatore in sé non può essere cambiato:

```
…
    int a[3];
    int *const p = a;  // Puntatore in sola lettura.
    p[1] = 9;
    p = a;             // Questo non si può!
…
```

L'esempio seguente, invece, fa sì che la memoria a cui si vuole accedere tramite il puntatore sia protetta in sola lettura:

```
…
    int a[3];
    const int *p = a;   // Qui è la memoria a essere
                        // in sola lettura.
    p[1] = 9;           // Questo non si può!
    p = a;
…
```

Anche se si può bloccare il puntatore, così da farlo funzionare in modo equivalente a un array vero e proprio, rimane però il fatto che '**sizeof**', usato per «misurare» un puntatore, restituisce comunque la grandezza della variabile che costituisce il puntatore stesso. Inoltre ci sono altre questioni che riguardano i puntatori, affrontate in una sezione separata, a proposito dell'aritmetica dei puntatori.

66.5.7 Array e funzioni

Si è visto che le funzioni possono accettare solo parametri composti da tipi di dati elementari, compresi i puntatori. In questa situazione, l'unico modo per trasmettere a una funzione un array attraverso i parametri, è quello di inviarne il puntatore iniziale. Di conseguenza, le modifiche che vengono poi apportate da parte della funzione si riflettono nell'array di origine. Si osservi l'esempio seguente.

Listato 66.181. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/2Zibjb5j* , *http://ideone.com/geaQV* .

```
#include <stdio.h>

void elabora (int *p)
{
    p[0] = 10;
    p[1] = 100;
    p[2] = 1000;
}

int main (void)
{
    int a[3];

    elabora (a);
    printf ("%i %i %i \n",  a[0], a[1], a[2]);

    return 0;
}
```

La funzione ***elabora()*** utilizza un solo parametro, rappresentato da un puntatore a un tipo '**int**'. La funzione **presume** che il puntatore si riferisca all'inizio di un array di interi e così assegna alcuni valori ai primi tre elementi (anche il numero degli elementi non può essere determinato dalla funzione).

All'interno della funzione ***main()*** viene dichiarato l'array ***a*** di tre elementi interi e subito dopo viene passato come argomento alla funzione ***elabora()***. Così facendo, in realtà si passa il puntatore al primo elemento dell'array.

Infine, la funzione altera gli elementi come è già stato descritto e gli effetti si possono osservare così:

```
10 100 1000
```

L'esempio potrebbe essere modificato per presentare la gestione dell'array in modo più elegante. Per la precisione si tratta di ritoccare la funzione '**elabora**':


```
void elabora (int a[])
{
    a[0] = 10;
    a[1] = 100;
    a[2] = 1000;
}
```

Si tratta sostanzialmente della stessa cosa, solo che si pone l'accento sul fatto che l'argomento è un array di interi, benché di tipo incompleto.

In entrambi i casi, se all'interno della funzione si tenta di misurare la dimensione dell'array con l'operatore '**sizeof**', si ottiene solo la grandezza della variabile usata per contenere il puntatore relativo. Sarebbe anche possibile specificare la dimensione dell'array, senza però che questo fatto abbia delle conseguenze significative e senza che '**sizeof**' la consideri:

```
void elabora (int a[])  // Anche così sizeof restituisce
{                       // solo la grandezza del puntatore.
    a[0] = 10;
    a[1] = 100;
    a[2] = 1000;
}
```

66.5.8 Aritmetica dei puntatori

Con le variabili puntatore è possibile eseguire delle operazioni elementari: possono essere incrementate e decrementate. Il risultato che si ottiene è il riferimento a una zona di memoria adiacente, in funzione della dimensione del tipo di dati per il quale è stato creato il puntatore. Si osservi l'esempio seguente:

```
int i = 10;
int j;
int *p = &i;
p++;
j = *p;          // Attenzione!
```

In questo caso viene creato un puntatore al tipo '**int**' che inizialmente contiene l'indirizzo della variabile ***i***. Subito dopo questo puntatore viene incrementato di una unità e ciò comporta che si riferisca a un'area di memoria adiacente, immediatamente successiva a quella occupata dalla variabile ***i*** (molto probabilmente si tratta dell'area occupata dalla variabile ***j***). Quindi si tenta di copiare il valore di tale area di memoria, interpretato come '**int**', all'interno della variabile ***j***.

Se un programma del genere funziona nell'ambito di un sistema operativo che controlla l'utilizzo della memoria, se l'area che si tenta di raggiungere incrementando il puntatore non è stata allocata, si ottiene un «errore di segmentazione» e l'arresto del programma stesso. L'errore si verifica quando si tenta l'accesso, mentre la modifica del puntatore è sempre lecita.

Lo stesso meccanismo riguarda tutti i tipi di dati che non sono array, perché per gli array, l'incremento o il decremento di un puntatore riguarda i componenti dell'array stesso. In pratica, quando si gestiscono tramite puntatori, gli array sono da intendere come una serie di elementi dello stesso tipo e dimensione, dove, nella maggior parte dei casi, il nome dell'array si traduce nell'indirizzo del primo elemento:

```
int i[3] = { 1, 3, 5 };
int *p;
…
p = i;
```

Nell'esempio si vede che il puntatore ***p*** punta all'inizio dell'array di interi ***i[]***.

```
*p = 10; // Equivale a:  i[0] = 10.
p++;
*p = 30; // Equivale a:  i[1] = 30.
p++;
*p = 50; // Equivale a:  i[2] = 50.
```

Ecco che, incrementando il puntatore, si accede all'elemento adiacente successivo, in funzione della dimensione del tipo di dati. De-

crementando il puntatore si ottiene l'effetto opposto, di accedere all'elemento precedente. La stessa cosa avrebbe potuto essere ottenuta così, senza alterare il valore contenuto nella variabile ***p***:

```
*(p + 0) = 10; // Equivale a:  i[0] = 10.
*(p + 1) = 30; // Equivale a:  i[1] = 30.
*(p + 2) = 50; // Equivale a:  i[2] = 50.
```

Inoltre, come già visto in altre sezioni, si potrebbe usare il puntatore con la stessa notazione propria dell'array, ma ciò solo perché si opera a una sola dimensione:

```
p[0] = 10; // Equivale a:  i[0] = 10.
p[1] = 30; // Equivale a:  i[1] = 30.
p[2] = 50; // Equivale a:  i[2] = 50.
```

Questo lascia intuire che '**i[*n*]**' corrisponda in pratica a '***(i + n)***', cosa che è vera per lo standard del linguaggio, ma potrebbe non essere accettabile dal compilatore che si usa effettivamente:

```
*(i + 0) = 10; // Equivale a:  i[0] = 10.
*(i + 1) = 30; // Equivale a:  i[1] = 30.
*(i + 2) = 50; // Equivale a:  i[2] = 50.
```

In presenza di più dimensioni, il ragionamento è analogo. Nel modello seguente, le lettere ***i*** e ***j*** rappresentano gli indici usati per la scansione, mentre le lettere ***I*** e ***J*** sono la quantità di elementi della dimensione corrispondente. Per esempio, secondo il modello seguente, in un array ***x[10][30]***, la lettera ***J*** corrisponde a 30.

```
x[i][j] == *(x + (i * J) + j)
```

In modo analogo si dovrebbe procedere per dimensioni maggiori:

```
x[i][j][k] == *(x + (i*J*K) + (j*K) + k)
```

Se il compilatore non accetta questo modo di gestire un array, il meccanismo vale per un puntatore dello stesso tipo degli elementi dell'array (che punti all'inizio dell'array stesso). L'esempio seguente mette in evidenza l'uso di un puntatore per scandire un array a due dimensioni.

Listato 66.191. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/tnGTykMU* , *http://ideone.com/ogdeH* .

```
#include <stdio.h>

int main (int argc, char *argv[])
{

  int a[3][4] = {{1,  2,  3,  4},
                 {5,  6,  7,  8},
                 {9, 10, 11, 12}};
  int i, j;
  const int *p = (int *) a;
  int x;

  for (i = 0; i < 3; i++)
    {
      for (j = 0; j < 4; j++)
        {
          x = *(p + i * 4 + j);
          //
          printf ("a[%i][%i]=%i\t", i, j, x);
          //
        }
      printf ("\n");
    }

  return 0;
}
```

I punti più importanti dell'esempio appaiono evidenziati: trattandosi di un array a più di una dimensione, la copia del puntatore avviene con l'ausilio di un cast; la scansione degli indirizzi, a partire dal puntatore ***p*** avviene attraverso una formula, mentre la forma seguente ha

un significato diverso, descritto in un'altra sezione, a proposito dei puntatori a puntatori:

```
…
          x = p[i][j];        // Non è la stessa cosa!
…
```

La versione funzionante dell'esempio mostrato deve fare apparire il testo seguente:

```
a[0][0]=1       a[0][1]=2       a[0][2]=3       a[0][3]=4
a[1][0]=5       a[1][1]=6       a[1][2]=7       a[1][3]=8
a[2][0]=9       a[2][1]=10      a[2][2]=11      a[2][3]=12
```

Naturalmente, quando si usano direttamente i puntatori, è compito esclusivo del programmatore sapere quando l'incremento o il decremento di un puntatore ha significato. Diversamente si rischia di accedere a zone di memoria estranee al contesto di proprio interesse, con risultati imprevedibili.

Prima di concludere l'argomento, vale la pena di tradurre il problema dell'aritmetica dei puntatori in modo opposto, ovvero come indirizzi. Per esempio, dato l'array ***a[]***, a una sola dimensione, si può considerare equivalente la notazione '**&(*a*[*i*])**' rispetto a '***(a + i)***'.

66.5.9 Osservazioni sui puntatori

Ammesso che la variabile ***p*** sia un puntatore a qualcosa, la notazione ***p** equivale a '**p[0]**', così come '***(p+n)**' corrisponde '**p[n]**'. Pertanto, l'uso delle parentesi quadre contenenti un indice, poste dopo il nome di una variabile puntatore, corrisponde alla dereferenziazione che si fa con l'asterisco.

Ammesso che la variabile ***p*** sia un puntatore a qualcosa, la notazione '**&*p**' corrisponde sempre a '***p***', anche se si tratta di un puntatore nullo.

Ammesso che la variabile ***x*** sia tale da potervi assegnare un valore e che possa essere operando di '**&**', la notazione '***&x**' corrisponde sempre a '***x***'.

Ammesso che la variabile ***p*** sia un puntatore a qualcosa, la notazione '***(tipo)p***' individua un'area di memoria che parte dalla posizione indicata dal puntatore e si estende per la dimensione del tipo indicato. In altre parole, si tratta di un cast con il quale si trasforma il tipo di puntatore al volo, ma per questo occorre mostrare un esempio.

Listato 66.194. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/89Ev895Myz* , *http://ideone.com/2A0nj* .

```
#include <stdio.h>
int main (int argc, char *argv[])
{
  int x = 10;
  void *p = &x;
  printf ("%i\n", *(int *) p);
  return 0;
}
```

In questo caso, il puntatore ***p*** è di tipo indefinito ('**void**') e riceve l'indirizzo della variabile ***x***. Successivamente, il valore a cui punta ***p*** viene usato all'interno della funzione ***printf()***, ma prima di essere dereferenziato, viene convertito in un puntatore di tipo '**int ***'.

66.5.10 Stringhe

Le stringhe, nel linguaggio C, non sono un tipo di dati a sé stante; si tratta solo di array di caratteri con una particolarità: l'ultimo carattere è sempre zero, ovvero una sequenza di bit a zero, che si rappresenta simbolicamente come carattere con '**\0**'. In questo modo, si evita di dover accompagnare le stringhe con l'informazione della loro lunghezza.

Pertanto, va osservato che una stringa è sempre un array di caratteri, ma un array di caratteri non è necessariamente una stringa, in quanto per esserlo occorre che l'ultimo elemento sia il caratte-

re '**\0**'. Seguono alcuni esempi che servono a comprendere questa distinzione.

```
char c[20];
```

L'esempio mostra la dichiarazione di un array di caratteri, senza specificare il suo contenuto. Per il momento non si può parlare di stringa, soprattutto perché per essere tale, la stringa deve contenere dei caratteri.

```
char c[] = {'c', 'i', 'a', 'o'};
```

Questo esempio mostra la dichiarazione di un array di quattro caratteri. All'interno delle parentesi quadre non è stata specificata la dimensione perché questa si determina dall'inizializzazione. Anche in questo caso non si può ancora parlare di stringa, perché manca la terminazione.

```
char z[] = {'c', 'i', 'a', 'o', '\0'};
```

Questo esempio mostra la dichiarazione di un array di cinque caratteri corrispondente a una stringa vera e propria. L'esempio seguente è tecnicamente equivalente, solo che utilizza una rappresentazione più semplice:

```
char z[] = "ciao";
```

Pertanto, la stringa rappresentata dalla costante '**"ciao"**' è un array di cinque caratteri, perché, pur senza mostrarlo, include implicitamente anche la terminazione.

L'indicazione letterale di una stringa può avvenire attraverso sequenze separate, senza l'indicazione di alcun operatore di concatenamento. Per esempio, '**"ciao amore\n"**' è perfettamente uguale a '**"ciao " "amore" "\n"**' che viene inteso come una costante unica.

In un sorgente C ci sono varie occasioni di utilizzare delle stringhe letterali (delimitate attraverso gli apici doppi), senza la necessità di dichiarare l'array corrispondente. Però è importante tenere presente la natura delle stringhe per sapere come comportarsi con loro. Per prima cosa, bisogna rammentare che la stringa, anche se espressa in forma letterale, è un array di caratteri; come tale restituisce semplicemente il puntatore del primo di questi caratteri (salvo le stesse eccezioni che riguardano tutti i tipi di array).

```
char *p;
…
p = "ciao";
…
```

L'esempio mostra il senso di quanto affermato: non esistendo un tipo di dati «stringa», si può assegnare una stringa solo a un puntatore al tipo '**char**' (ovvero a una variabile di tipo '**char ***'). L'esempio seguente non è valido, perché non si può assegnare un valore alla variabile che rappresenta un array, dal momento che il puntatore relativo è un valore costante:

```
char z[];
…
z = "ciao";     // Non si può.
…
```

Quando si utilizza una stringa tra gli argomenti della chiamata di una funzione, questa riceve il puntatore all'inizio della stringa. In pratica, si ripete la stessa situazione già vista per gli array in generale.

Listato 66.201. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/qCgdnWnE* , *http://ideone.com/kkzaT* .

```
#include <stdio.h>

void elabora (char *z)
{
  printf (z);
}

int main (void)
{
  elabora ("ciao\n");
  return 0;
}
```


L'esempio mostra una funzione banale che si occupa semplicemente di emettere la stringa ricevuta come parametro, utilizzando ***printf()***. La variabile utilizzata per ricevere la stringa è stata dichiarata come puntatore al tipo '**char**' (ovvero come puntatore di tipo '**char ***'), poi tale puntatore è stato utilizzato come argomento per la chiamata della funzione ***printf()***. Volendo scrivere il codice in modo più elegante si potrebbe dichiarare apertamente la variabile ricevente come array di caratteri di dimensione indefinita. Il risultato è lo stesso.

Listato 66.202. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/83oK83pP4x* , *http://ideone.com/uwq6D* .

```
#include <stdio.h>

void elabora (char z[])
{
  printf (z);
}

int main (void)
{
  elabora ("ciao\n");
  return 0;
}
```

Tabella 66.203. Funzioni comuni per la gestione delle stringhe, definite nel file '`string.h`' (il modificatore '**restrict**' viene descritto in una sezione apposita).

Funzione	Descrizione
`char *strcpy (char *restrict dst, const char *restrict org);` `char *strncpy (char *restrict dst, const char *restrict org, size_t n);`	La funzione ***strcpy()*** copia il contenuto della stringa ***org*** nella stringa ***dst***, compreso il carattere di terminazione <*NUL*>. Perché l'operazione possa avvenire è necessario che le due stringhe non si sovrappongano e che per la stringa di destinazione ci sia abbastanza spazio per i caratteri da copiare. La funzione restituisce il puntatore all'inizio della stringa di destinazione. La funzione ***strncpy()*** si comporta sostanzialmente come ***strcpy()***, con la differenza che copia al massimo ***n*** caratteri, aggiungendo comunque il carattere di terminazione <*NUL*>.

Funzione	Descrizione
`char *strcat (char *restrict dst, const char *restrict org);` `char *strncat (char *restrict dst, const char *restrict org, size_t n);`	La funzione ***strcat()*** accoda alla stringa ***dst*** il contenuto della stringa ***org***, sovrascrivendo il carattere <*NUL*> che concludeva la prima stringa e aggiungendolo comunque alla fine della copia. Perché l'operazione possa avvenire è necessario che le due stringhe non si sovrappongano e, soprattutto, che ci sia abbastanza spazio disponibile dopo la prima stringa da estendere. La funzione restituisce il puntatore alla prima stringa. La funzione ***strncat()*** si comporta sostanzialmente come ***strcat()***, con la differenza che copia al massimo ***n*** caratteri dalla seconda stringa, aggiungendo comunque il carattere di terminazione <*NUL*>.

`int strcmp (const char *str_1, const char *str_2);`
`int strcoll (const char *str_1, const char *str_2);`
`int strncmp (const char *str_1, const char *str_2, size_t n);`
	La funzione ***strcmp()*** confronta due stringhe e restituisce zero nel caso siano uguali, oppure un valore minore di zero se la prima stringa è minore della seconda, oppure un valore maggiore di zero se la prima stringa è maggiore della seconda.
	La funzione ***strcoll()*** funziona sostanzialmente come ***strcmp()***, con la differenza che il confronto ha luogo tenendo conto della configurazione locale (precisamente la categoria '**`LC_COLLATE`**').
	La funzione ***strncmp()*** si comporta sostanzialmente come ***strcmp()***, con la differenza che confronta al massimo ***n*** caratteri.

`char *strchr (const char *str, int c);` `char *strrchr (const char *str, int c);`	La funzione ***strchr()*** cerca nella stringa ***str*** il carattere ***c*** (il carattere che si ottiene riducendo il valore di ***c*** a quello di un tipo '**`char`**'), includendo nella ricerca anche il carattere di terminazione <*NUL*>. La funzione restituisce un puntatore al carattere trovato, oppure restituisce il puntatore nullo se questo non c'è. La funzione ***strrchr()*** si comporta sostanzialmente come ***strchr()***, con la differenza che cerca l'ultima corrispondenza disponibile nella stringa.

Funzione	Descrizione
`char *strpbrk (const char *str_1, const char *str_2);`	La funzione ***strpbrk()*** cerca nella stringa ***str_1*** la prima corrispondenza con uno qualsiasi dei caratteri contenuti nella stringa ***str_2***. Restituisce il puntatore al carattere trovato nella stringa ***str_1*** che soddisfi la condizione; se non trova alcuna corrispondenza restituisce il puntatore nullo.
`size_t strspn (const char *str_1, const char *str_2);` `size_t strcspn (const char *str_1, const char *str_2);`	La funzione ***strspn()*** conta la lunghezza massima della sottostringa iniziale di ***str_1*** che contiene soltanto caratteri dell'insieme contenuto nella stringa ***str_2***. La funzione ***strcspn()*** svolge il compito opposto, di contare la lunghezza massima della sottostringa iniziale di ***str_2***, contenente solo caratteri che **non** fanno parte dell'insieme contenuto in ***str_2***.
`size_t strlen (const char *str);`	La funzione ***strlen()*** restituisce la quantità di caratteri contenuta nella stringa, escluso il carattere di terminazione <*NUL*>.

All'inizio del capitolo, in occasione della descrizione delle costanti letterali per i tipi di dati primitivi, è già descritto il modo con cui si possono rappresentare alcuni caratteri speciali attraverso delle sequenze di escape che vengono annotate qui, nuovamente, per maggiore comodità del lettore, in quanto quelle sequenze sono valide anche nelle stringhe letterali.

Tabella 66.204. Elenco dei modi di rappresentazione delle costanti carattere attraverso codici di escape.

Codice di escape	Descrizione
`\ooo`	Notazione ottale.
`\xhh`	Notazione esadecimale.
`\\`	Una singola barra obliqua inversa ('`\`').
`\'`	Un apice singolo destro.
`\"`	Un apice doppio.
`\?`	Un punto interrogativo. Si usa in quanto le sequenze *trigraph* sono formate da un prefisso di due punti interrogativi.
`\0`	Il codice <*NUL*>.
`\a`	Il codice <*BEL*> (*bell*).
`\b`	Il codice <*BS*> (*backspace*).
`\f`	Il codice <*FF*> (*formfeed*).
`\n`	Il codice <*LF*> (*linefeed*).
`\r`	Il codice <*CR*> (*carriage return*).
`\t`	Una tabulazione orizzontale (<*HT*>).
`\v`	Una tabulazione verticale (<*VT*>).

66.5.11 Parametri della funzione main()

La funzione ***main()***, se viene dichiarata con i suoi parametri tradizionali, permette di acquisire la riga di comando utilizzata per avviare il programma. La dichiarazione completa è la seguente:

```
int main (int argc, char *argv[])
{
  …
}
```

Gli argomenti della riga di comando vengono convertiti in un array di stringhe (cioè di puntatori a '**char**'), in cui il primo elemento è il nome utilizzato per avviare il programma e gli elementi successivi sono gli altri argomenti. Il primo parametro, ***argc***, serve a contenere la quantità di elementi del secondo, ***argv[]***, il quale è l'array di stringhe da scandire. È il caso di annotare che questo array dovrebbe avere sempre almeno un elemento: il nome utilizzato per avviare il programma e, di conseguenza, ***argc*** è sempre maggiore o uguale a uno.[21]

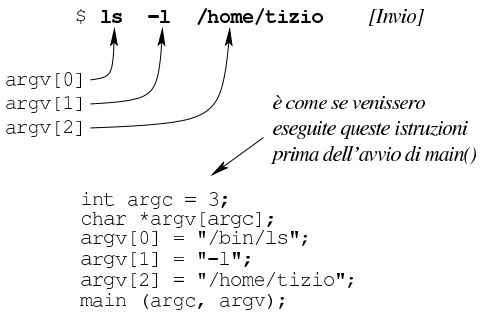
L'esempio seguente mostra in che modo gestire tale array, con la semplice riemissione degli argomenti attraverso lo standard output.

```
#include <stdio.h>

int main (int argc, char *argv[])
{
  int i;

  printf ("Il programma si chiama %s\n", argv[0]);

  for (i = 1; i < argc; i++)
    {
      printf ("argomento n. %i: %s\n", i, argv[i]);
    }
}
```

In alternativa, ma con lo stesso effetto, l'array di puntatori a stringhe può essere definito nel modo seguente, come puntatore di puntatori a caratteri:

```
int main (int argc, char **argv)
{
  …
}
```

Figura 66.208. Schematizzazione di ciò che accade alla chiamata della funzione ***main()***, con un esempio.



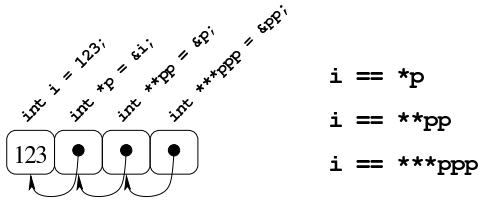
Chi è abituato a utilizzare linguaggi di programmazione più evoluti del C, può trovare strano che non si possa scrivere '**main (int argc, char argv[][])**' e usare di conseguenza l'array. Il motivo per cui ciò non è possibile dipende dal fatto che gli array a più dimensioni sono ottenuti attraverso sottoinsiemi uniformi del tipo dichiarato, così, in questo caso le stringhe dovrebbero essere della stessa dimensione, ma evidentemente ciò non corrisponde alla realtà. Inoltre, la dichiarazione della funzione dovrebbe contenere le dimensioni dell'array che non possono essere note. Pertanto, un array formato da stringhe diseguali, può essere ottenuto solo come array di puntatori al tipo '**char**'.

66.5.12 Puntatori a puntatori

Una variabile puntatore potrebbe fare riferimento a un'area di memoria contenente a sua volta un puntatore per un'altra area. Per dichiarare una cosa del genere, si possono usare più asterischi, come nell'esempio seguente:

```
int i = 123;
int *p = &i;        // Puntatore al tipo "int".
int **pp = &p;      // Puntatore di puntatore al tipo
                    // "int".
int ***ppp = &pp;   // Puntatore di puntatore di
                    // puntatore al tipo "int".
```

Il risultato si potrebbe rappresentare graficamente come nello schema seguente:



Per dimostrare in pratica il funzionamento di questo meccanismo di riferimenti successivi, si può provare con il programma seguente.

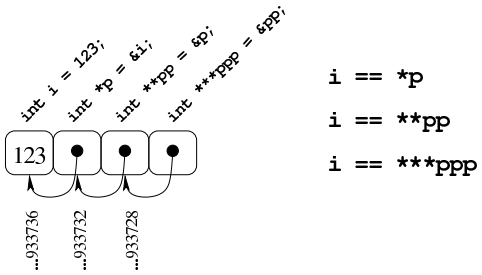
Listato 66.211. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/cettwOsS* , *http://ideone.com/pzBoW* .

```
#include <stdio.h>
int
main (void)
{
  int i = 123;
  int *p = &i;        // Puntatore al tipo "int".
  int **pp = &p;      // Puntatore di puntatore al tipo
                      // "int".
  int ***ppp = &pp;   // Puntatore di puntatore di puntatore
                      // al tipo "int".

  printf ("i, p, pp, ppp: %i, %u, %u, %u\n",
          i, (unsigned int) p, (unsigned int) pp,
          (unsigned int) ppp);

  printf ("i, p, pp, *ppp: %i, %u, %u, %u\n",
          i, (unsigned int) p, (unsigned int) pp,
          (unsigned int) *ppp);

  printf ("i, p, *pp, **ppp: %i, %u, %u, %u\n",
          i, (unsigned int) p, (unsigned int) *pp,
          (unsigned int) **ppp);

  printf ("i, *p, **pp, ***ppp: %i, %i, %i, %i\n",
           i, *p, **pp, ***ppp);

  return 0;
}
```

Eseguendo il programma si dovrebbe ottenere un risultato simile a quello seguente, dove si può verificare l'effetto delle dereferenziazioni applicate alle variabili puntatore:

```
i, p, pp, ppp: 123, 3217933736, 3217933732, 3217933728
i, p, pp, *ppp: 123, 3217933736, 3217933732, 3217933732
i, p, *pp, **ppp: 123, 3217933736, 3217933736, 3217933736
i, *p, **pp, ***ppp: 123, 123, 123, 123
```

Pertanto si può ricostruire la disposizione in memoria delle variabili:



Come si può comprendere facilmente, la gestione di puntatori a puntatore è difficile e va usata con prudenza e solo quando ne esiste effettivamente l'utilità. Va notato anche che si ottiene la dereferenziazione (la traduzione di un puntatore nel contenuto di ciò a cui punta) usando la notazione tipica degli array, ma questo fatto viene descritto nella sezione successiva.

66.5.13 Puntatori a più dimensioni

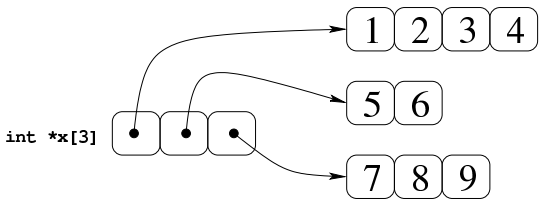
Un array di puntatori consente di realizzare delle strutture di dati ad albero, non più uniformi come invece devono essere gli array a più dimensioni consueti. L'esempio seguente mostra la dichiarazione di tre array di interi, con una quantità di elementi disomogenea, e la successiva dichiarazione di un array di puntatori di tipo '`int *`', a cui si assegnano i riferimenti ai tre array precedenti. Nell'esempio appare poi un tipo di notazione per accedere ai dati terminali che dovrebbe risultare intuitiva, ma se ne possono usare delle altre.

Listato 66.214. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/MPES5c6X* , *http://ideone.com/5bP39* .

```
#include <stdio.h>

int main (void)
{
  int a[] = {1, 2, 3, 4};
  int b[] = {5, 6,};
  int c[] = {7, 8, 9};
  int *x[] = {a, b, c};

  printf ("*x[0] = {%i, %i, %i, %i}\n",
          *x[0], *(x[0]+1), *(x[0]+2), *(x[0]+3));
  printf ("*x[1] = {%i, %i}\n", *x[1], *(x[1]+1));
  printf ("*x[2] = {%i, %i, %i}\n",
          *x[2], *(x[2]+1), *(x[2]+2));

  return 0;
}
```

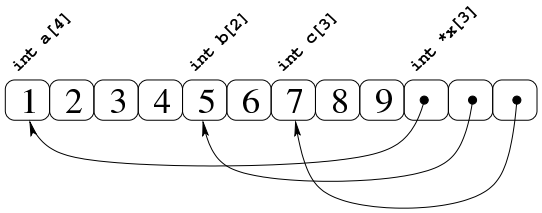
La figura successiva dovrebbe facilitare la comprensione del senso dell'array di puntatori. Come si può osservare, per accedere agli elementi degli array a cui puntano quelli di ***x*** è necessario dereferenziare gli elementi. Pertanto, '`*x[0]`' corrisponde al contenuto del primo elemento del primo sotto-array, '`*(x[0]+1)`' corrisponde al contenuto del secondo elemento del primo sotto-array e così di seguito. Dal momento che i sotto-array non hanno una quantità uniforme di elementi, non è semplice la loro scansione.

Figura 66.215. Schematizzazione semplificata del significato dell'array di puntatori definito nell'esempio.



Si potrebbe obbiettare che la scansione di questo array di puntatori a array può avvenire ugualmente in modo sequenziale, come se fosse un array «normale» a una sola dimensione. Molto probabilmente ciò è possibile effettivamente, dal momento che è probabile che il compilatore disponga le variabili in memoria in sequenza, come si vede nella figura successiva, ma ciò non può essere garantito.

Figura 66.216. La disposizione più probabile delle variabili dell'esempio.



Se invece di un array di puntatori si ha un puntatore di puntatori, il meccanismo per l'accesso agli elementi terminali è lo stesso. L'esempio seguente contiene la dichiarazione di un puntatore a puntatori di tipo intero, a cui viene assegnato l'indirizzo dell'array già descritto. La scansione può avvenire nello stesso modo, ma ne viene proposto uno alternativo e più chiaro, con il quale si comprende cosa si intende per puntatore a più dimensioni.

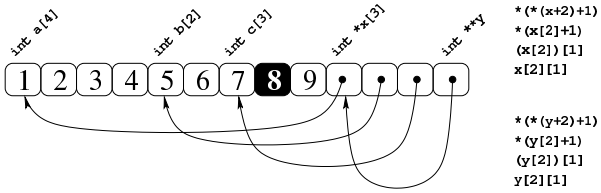
Listato 66.217. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/kDRp85cQ85* , *http://ideone.com/JaqDH* .

```
#include <stdio.h>

int main (void)
{
  int a[] = {1, 2, 3, 4};
  int b[] = {5, 6,};
  int c[] = {7, 8, 9};
  int *x[] = {a, b, c};
  int **y = x;

  printf ("*x[0] = {%i, %i, %i, %i}\n", y[0][0], y[0][1],
                                        y[0][2], y[0][3]);
  printf ("*x[1] = {%i, %i}\n", y[1][0], y[1][1]);
  printf ("*x[2] = {%i, %i, %i}\n", y[2][0], y[2][1],
                                    y[2][2]);

  return 0;
}
```

Come si vede, la variabile ***y*** viene usata come se fosse un array a due dimensioni, ma lo stesso sarebbe valso per la variabile ***x***, in qualità di array di puntatori.

Per capire cosa succede, occorre fare mente locale al fatto che il nome di una variabile puntatore seguito da un numero tra parentesi quadre corrisponde alla dereferenziazione dell'***n***-esimo elemento successivo alla posizione a cui punta tale variabile, mentre il valore puntato in sé corrisponde all'elemento zero (ciò è come dire che ****p*** equivale a '`*p[0]`'). Quindi, scrivere '`*(p+n)`' è esattamente uguale a scrivere '`p[n]`'. Se il valore a cui punta una variabile puntatore è a sua volta un puntatore, per dereferenziarlo occorrono due fasi: per esempio *****p*** è il valore che si ottiene dereferenziando il primo puntatore e quello che si trova nella prima destinazione (quindi *****p*** equivale a '`*p[0]`' e a '`p[0][0]`'). Volendo gestire gli indici si possono considerare equivalenti i puntatori: '`*(*(p+m)+n)`', '`*(p[m]+n)`', '`(p[m])[n]`' e '`p[m][n]`'.

Figura 66.218. Tanti modi alternativi per raggiungere lo stesso elemento.



Seguendo lo stesso ragionamento si possono gestire strutture ad albero più complesse, con più livelli di puntatori, ma qui non vengono

proposti esempi di questo tipo.

Sia l'array di puntatori, sia il puntatore a puntatori, possono essere gestiti con gli indici come se si trattasse di un array a più dimensioni. Pertanto, la notazione '***a*[*m*][*n*]**' può rappresentare l'elemento ***m*,*n*** di un array ***a*** ottenuto secondo la rappresentazione «normale» a matrice, oppure secondo uno schema ad albero attraverso dei puntatori: la differenza sta solo nella presenza o meno di elementi costituiti da puntatori.

66.5.14 Puntatori e funzioni

Nello standard del linguaggio C, la dichiarazione di una funzione è in pratica la definizione di un puntatore al codice della stessa, un po' come accade con gli array.[22] In generale, è possibile dichiarare dei puntatori a un tipo di funzione definito in base al valore restituito e ai tipi di parametri richiesti, attraverso una forma che richiama quella del prototipo di funzione. Il modello seguente è quello della dichiarazione del prototipo:

***tipo nome_funzione* (*tipo_parametro*[*nome_parametro*][,…]);**

Questo è invece il modello della dichiarazione del puntatore:

***tipo* (**nome_puntatore*) (*tipo_parametro*[*nome_parametro*][,…]);**

L'esempio seguente mostra la dichiarazione di un puntatore a una funzione che restituisce un valore di tipo '**int**' e utilizza due parametri di tipo '**int**':

```
int (*f) (int, int);
```

L'esempio seguente è equivalente, con la differenza che si nominano i parametri, anche se ciò è perfettamente inutile, esattamente come nei prototipi delle funzioni:

```
int (*f) (int i, int j);
```

L'assegnamento del puntatore avviene nel modo più semplice possibile, trattando il nome della funzione nello stesso modo in cui si fa con gli array: come un puntatore.

```
int (*f) (int, int);     // Puntatore a funzione.
int prodotto (int, int); // Prototipo di funzione descritta
                         // più avanti.
…
f = prodotto; // Il puntatore «f» contiene il riferimento
              // alla funzione.
```

Una volta assegnato il puntatore, si può eseguire una chiamata di funzione semplicemente utilizzando il puntatore, per cui, i due esempi seguenti sono equivalenti:

```
i = f (2, 3);
```

```
i = prodotto (2, 3);
```

Nel linguaggio C precedente allo standard ANSI, perché il puntatore potesse essere utilizzato in una chiamata di funzione, occorreva indicare l'asterisco, in modo da dereferenziarlo:

```
i = (*f) (2, 3);          // Non serve più.
```

Per concludere viene mostrato un esempio completo, anche se banalizzato: la funzione ***f()*** restituisce un numero intero ottenuto incrementando di una unità l'argomento ricevuto. Questa funzione viene chiamata attraverso un puntatore denominato ***pf***.

Listato 66.225. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/rbXNWbOh*, *http://ideone.com/L6ooG*.

```
#include <stdio.h>

int f (int i)
{
    return (i + 1);
}

int main (void)
{
  int x = 4;
  int y;
  int (*pf) (int i);
  pf = f;
  y = pf (x);
  printf ("%i + 1 = %i\n", x, y);
  return 0;
}
```

Riquadro 66.226. Confusione tra le dichiarazioni.

L'interpretazione umana del linguaggio, a proposito dei puntatori, può essere complicata, pertanto l'uso dei puntatori deve essere fatto con criterio, senza abusarne. Gli esempi seguenti sono solo i più semplici:
`int f (…);` /* dichiarazione della funzione ***f()*** che restituisce un valore intero; */
`int *f (…);` /* dichiarazione della funzione ***f()*** che restituisce un puntatore a un intero; */
`int (*f) (…);` /* dichiarazione del puntatore ***f*** a una funzione che restituisce un intero; */
`int *(*f) (…);` /* dichiarazione del puntatore ***f*** a una funzione che restituisce un puntatore a un intero. */
Ancora più difficile sarebbe dichiarare una funzione che restituisce un array, o peggio, un puntatore a un array.

66.5.14.1 Puntatori a funzione, membri di una struttura

Le strutture sono descritte in un'altra sezione (66.7), tuttavia è opportuno annotare qui in che modo possa essere utilizzato un puntatore a una funzione, quando è un membro di una struttura:

***struttura*.*membro* (*argomenti*);**

(struttura*.*membro*) (*argomenti*);**

I due modelli sono equivalenti e si riferiscono alla chiamata di una funzione, il cui puntatore è costituito dalla variabile ***struttura*.*membro***. È evidente che risulta più comprensibile la prima delle due modalità. A titolo di esempio, ipotizzando la struttura ***totale*** e il membro ***sottrai***, per una funzione che riceve un argomento di tipo intero (precisamente il numero 7), la chiamata potrebbe essere scritta indifferentemente nei due modi successivi:

```
…
totale.sottrai (7);
…
```

```
…
(*totale.sottrai) (7);
…
```

66.5.15 Puntatori a variabili distrutte

L'esempio seguente potrebbe funzionare, ma contiene un errore di principio.

Listato 66.229. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/DRtSxmiS*, *http://ideone.com/egses*.

```
#include <stdio.h>

double *f (void)
{
  double x = 1234.5678;
  return &x;                  // Orrore!
```

```
}

int main (int argc, char *argv[])
{
  double *p;
  p = f ();
  printf ("x = %f\n", *p);
  return 0;
}
```

La funzione ***f()*** dichiara localmente una variabile che inizializza al valore 1234,5678, quindi restituisce il puntatore a questa variabile. A parte il fatto che il compilatore possa segnalare o meno la cosa, non si può utilizzare un puntatore rivolto a un'area di memoria che, almeno teoricamente, non è più allocata. In altri termini, se si costruisce un puntatore a qualcosa, occorre tenere sempre presente il ciclo di vita della sua destinazione e non solo della variabile che contiene tale riferimento.

Purtroppo questa attenzione non viene imposta e, generalmente, il compilatore consente di usare un puntatore a variabili che, formalmente, sono già state distrutte.

66.5.16 Puntatore nullo

Il linguaggio C prescrive che si possa assegnare a una variabile puntatore il valore zero, in qualità di numero intero:

```
…
double *p = 0;
…
```

Il puntatore che contiene il valore zero è indefinito, nel senso che punta a un'area di memoria irraggiungibile. Un puntatore di questo tipo è noto come ***puntatore nullo*** o *null pointer*; inoltre, due puntatori nulli, qualunque sia il tipo di dati a cui si riferiscono, sono uguali in una comparazione. Pertanto si potrebbe verificare la validità di un puntatore nel modo seguente:

```
…
char *p = 0;
…
if (p == 0)
  {
    // Null pointer.
    …
  }
…
```

A ogni modo, lo standard prescrive che nel file '`stddef.h`' sia definita la macro-variabile ***NULL***, a rappresentare formalmente un puntatore nullo:

```
#include <stddef.h>
…
char *p = NULL;
…
if (p == NULL)
  {
    // Null pointer.
    …
  }
…
```

Va osservato che la variabili puntatore, quando acquisiscono un indirizzo in base al verificarsi di certe condizioni, vanno inizializzate opportunamente al valore nullo (come già apparso negli esempi), in modo da poter poi verificare se hanno ottenuto o meno un tale indirizzo.

66.5.17 Utilizzo della memoria in modo dinamico

L'allocazione dinamica della memoria avviene generalmente attraverso la funzione ***malloc()***, oppure ***calloc()***, definite nella libreria standard, secondo i prototipi contenuti nel file '`stdlib.h`'. Se queste riescono a eseguire l'operazione, restituiscono il puntatore alla memoria allocata, altrimenti restituiscono il valore '**`NULL`**'.


```
void *malloc (size_t dimensione);
```

```
void *calloc (size_t quantità, size_t dimensione);
```

La differenza tra le due funzioni sta nel fatto che la prima, ***malloc()***, viene utilizzata per allocare un'area di una certa dimensione, espressa generalmente in byte, mentre la seconda, ***calloc()***, permette di indicare una quantità di elementi e si presta per l'allocazione di array.

Dovendo utilizzare queste funzioni per allocare della memoria, è necessario conoscere la dimensione dei tipi primitivi di dati, ma per evitare incompatibilità conviene farsi aiutare dall'operatore '**`sizeof`**'.

Il valore restituito da queste funzioni è di tipo '**`void *`**' cioè una specie di puntatore neutro, indipendente dal tipo di dati da utilizzare (in quanto il tipo '**`void`**', in sé, rappresenta una variabile di rango nullo, la quale non può contenere alcun dato). Per questo, in linea di principio, prima di assegnare a un puntatore il risultato dell'esecuzione di queste funzioni di allocazione, è opportuno eseguire un cast.

```
int *pi = NULL;
…
pi = (int *) malloc (sizeof (int));

if (pi != NULL)
  {
    // Il puntatore è valido e allora procede.
    …
  }
else
  {
    // La memoria non è stata allocata e si fa qualcosa
    // di alternativo.
    …
  }
```

Come si può osservare dall'esempio, il cast viene eseguito con la notazione '**`(int *)`**' che richiede la conversione esplicita in un puntatore a '**`int`**'. Lo standard C non richiede l'utilizzo di questo cast, quindi l'esempio si può ridurre al modo seguente:

```
…
pi = malloc (sizeof (int));
…
```

La memoria allocata dinamicamente deve essere liberata in modo esplicito quando non serve più. Infatti, il linguaggio C non offre alcun meccanismo di ***raccolta della spazzatura*** o *garbage collector*. Per questo si utilizza la funzione ***free()*** che richiede semplicemente il puntatore e non restituisce alcunché.

```
void free (void *puntatore);
```

È necessario evitare di deallocare più di una volta la stessa area di memoria, perché ciò potrebbe provocare effetti imprevedibili.

```
int *pi = NULL;
…
pi = (int *) malloc (sizeof (int));

if (pi != NULL)
  {
    // Il puntatore è valido e allora procede.
    …
    free (pi); // Libera la memoria
    pi = NULL; // e per sicurezza azzera il puntatore.
    …
  }
else
  {
    // La memoria non è stata allocata e si fa qualcosa
```

```
    // di alternativo.
    …
}
```

Lo standard prevede una funzione ulteriore, per la riallocazione di memoria: ***realloc()***. Questa funzione si usa per ridefinire l'area di memoria con una dimensione differente:

```
void *realloc (void *puntatore, size_t dimensione);
```

In pratica, la riallocazione deve rendere disponibili gli stessi contenuti già utilizzati, salvo la possibilità che questi siano stati ridotti nella parte terminale. Se invece la dimensione richiesta nella riallocazione è maggiore di quella precedente, lo spazio aggiunto può contenere dati casuali. Va osservato che la collocazione in memoria, successiva alla riallocazione, può essere differente da quella precedente. Il funzionamento di ***realloc()*** non è garantito, pertanto occorre verificare nuovamente, dopo il suo utilizzo, che il puntatore ottenuto sia ancora valido.

66.5.18 Puntatori «ristretti»

Lo standard del linguaggio C prevede il modificatore '**restrict**' per le variabili puntatore, da usare come nell'esempio seguente:

```
…
int *restrict p;
…
```

L'utilizzo di tale modificatore equivale a una dichiarazione di intenti (ovvero una promessa) che il programmatore fa al compilatore, nei riguardi del puntatore. Precisamente si dichiara che il puntatore viene usato per accedere ad aree di memoria in modo esclusivo, nel senso che nell'ambito del contesto a cui si fa riferimento, non esistono altri accessi alle stesse aree per mezzo di altri puntatori o di altre variabili. Partendo da questo presupposto, il compilatore può ottimizzare il risultato della compilazione semplificando il codice finale.

La definizione formale del significato di questo modificatore è molto complessa e il compilatore non è in grado di segnalarne un uso improprio. Ciò significa che va usata questa possibilità con prudenza, solo quando si ritiene di averne capito il senso e l'utilità.

Come esempio iniziale si può osservare il prototipo della funzione standard ***strcpy()***:

```
char *strcpy (char *restrict dst, const char *restrict org);
```

Ci sono due parametri costituiti da stringhe che non devono risultare sovrapposte e in questo caso, il vincolo '**restrict**' è appropriato per esprimere il concetto: se entrambi i puntatori delle stringhe sono dichiarati con il modificatore '**restrict**', è evidente che le stringhe rispettive non devono sovrapporsi.

L'impegno che il programmatore prende utilizzando il modificatore '**restrict**' è finalizzato solo al favorire l'ottimizzazione della compilazione.

La promessa che un programmatore fa dichiarando un puntatore '**restrict**' è limitata al campo di azione del puntatore stesso. Per esempio, tornando all'esempio del prototipo della funzione ***strcpy()***, lì si intende che i parametri vengono usati nella funzione senza sovrapposizioni, ma, dato il contesto, rimane il fatto che le stringhe fornite come argomento della chiamata debbano già rispettare il vincolo di non essere sovrapposte.

Esempio 66.237. Viene allocata un'area di memoria composta da 100 elementi della grandezza di un intero normale. I primi 50 elementi vengono scanditi con il puntatore ***r1*** mentre quelli restanti con il puntatore ***r2***. Nell'esempio, agli elementi '***r1*[*i*]**' viene assegnato il valore di '***r2*[*i*]+1**', anche se il fatto in sé non ha una grande importanza.

```
int *restrict r1, *restrict r2;
int *m = malloc (100 * sizeof (int));
int i;

r1 = m;      // r1 viene usato per i primi 50 elementi.
r2 = m + 50; // r2 viene usato per i 50 elementi successivi.

for (i = 0; i < 50; i++)
  {
    r1[i] = r2[i] + 1;
  }
```


Esempio 66.238. Viene allocata un'area di memoria composta da 100 elementi della grandezza di un intero normale. Gli elementi pari vengono scanditi con il puntatore ***r1*** mentre quelli dispari con il puntatore ***r2***. Nell'esempio, agli elementi '***r1*[*j*]**' viene assegnato il valore di '***r2*[*j*]+1**', anche se il fatto in sé non ha una grande importanza.

```
int *restrict r1, *restrict r2;
int *m = malloc (100 * sizeof (int));
int i;
int j;

r1 = m;          // r1 viene usato per gli elementi con
                 // indice pari.
r2 = m + 1;      // r2 viene usato per gli elementi con
                 // indice dispari.

for (i = 0; i < 50; i++)
  {
    j = i * 2;
    r1[j] = r2[j] + 1;
  }
```

Se il compilatore non riconosce il modificatore '**restrict**' significa solo che non è in grado di ottimizzare il codice in un certo modo, ma non è necessario modificare il proprio programma per togliere la parola chiave relativa, perché è sufficiente sfruttare una macrovariabile del precompilatore, a cui non si assegna alcun valore:

```
…
#define restrict
…
```

Oppure, se ciò non è possibile, la si dichiara come un commento privo di contenuto:

```
…
#define restrict /**/
…
```

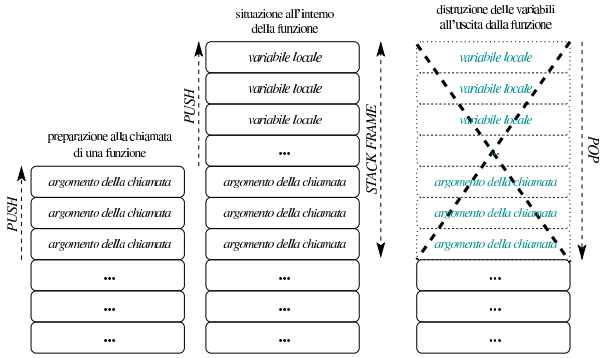
66.6 Le funzioni

Per comprendere come «funzionano» le funzioni nel linguaggio C, occorre fare mente locale all'uso della pila dei dati con il linguaggio macchina. Qui si chiariscono alcuni concetti, partendo dal ripasso della pila dei dati.

66.6.1 Pila dei dati

Dal punto di vista del linguaggio macchina, generalmente si dispone di una pila di dati che si sviluppa a partire da un certo indirizzo di memoria, utilizzando di volta in volta indirizzi inferiori della stessa. Attraverso la pila dei dati, prima della chiamata di una funzione, gli argomenti vengono passati alla stessa aggiungendoli alla pila; successivamente, all'interno della funzione, tutte le variabili locali vengono ottenute facendo crescere ulteriormente la pila. Al termine dell'esecuzione della funzione, la pila viene ridotta allo stato precedente alla chiamata, espellendo le variabili locali e i parametri della chiamata.

Figura 66.241. Semplificazione del meccanismo attraverso cui si passano gli argomenti a una funzione e si gestiscono le variabili locali.



Naturalmente, dal momento che la pila di dati viene gestita attraverso la memoria centrale, la quale consente un accesso diretto ai dati, tramite un indirizzo, nella pila si possono gestire dati di tutti i tipi, volendo anche degli array. A proposito degli array, quando questi sono creati all'interno delle funzioni, pertanto attraverso l'uso della pila dei dati, al compilatore non è necessario sapere preventivamente le dimensioni di questi, perché lo spazio che usano nella memoria è allocato dinamicamente, tramite la pila.

66.6.2 Dichiarazione e chiamata di una funzione

La dichiarazione di una funzione prevede l'indicazione del tipo di variabili che compongono i parametri, allo scopo di far sapere al compilatore in che modo inserire gli argomenti nella pila, al momento della chiamata. Si osservi l'esempio seguente in cui si dichiara una funzione con due parametri molto semplici: un intero normale e un intero di dimensione «doppia».

```
void f (int x, long long int y)
{
    …
    …
}
```

Partendo dal presupposto che la pila dei dati sia gestita a blocchi di «parole» del microprocessore, si può ipotizzare ragionevolmente in che modo siano impilati gli argomenti della chiamata. Si suppone di chiamare la funzione nel modo seguente e che la parola sia da 32 bit:

```
…
f (0x13579BDF, 0x123456789ABCDEF);
…
```

Alla chiamata della funzione, i parametri dovrebbero apparire nella pila come nella figura successiva, trascurando il problema dell'inversione eventuale dei byte:

Come si vede, gli argomenti vengono impilati in ordine inverso, in modo tale che il primo argomento appaia all'inizio della pila.

Ci sono molti dettagli da definire sul come vadano impilati gli argomenti di una chiamata; in particolare è da chiarire in che modo vadano trattati i dati la cui dimensione è inferiore alla parola del microprocessore, così come per quelli che si articolano in strutture. Questi dettagli vanno chiariti quando si vogliono scrivere funzioni da usare assieme a codice scritto in linguaggio assemblatore, oppure anche per altri linguaggi, se per quelli si utilizzano compilatori non conformi a quello usato per il C.

66.6.3 Elenco indefinito di parametri

Il linguaggio C ammette che le funzioni siano dichiarate con almeno un parametro esplicito e un elenco indefinito di parametri successivi. In altre parole, si ammette che ci sia un parametro certo e un elenco, eventuale, di altri parametri sconosciuti. Questo avviene, per esempio, con funzioni standard quali ***printf()***:

```
int printf (const char *formato, …);
```

Quando si chiama una funzione del genere, gli argomenti successivi al primo, se riguardano valori numerici, vengono «promossi» in modo tale da avere una dimensione minima di riferimento. Per la precisione, i valori interi di rango inferiore a quello di un intero comune, sono convertiti al livello di intero '**int**' (con segno o senza, in base alle caratteristiche di partenza); i valori in virgola mobile, se sono espressi secondo un formato di rango inferiore a '**double**', vengono trasformati semplicemente in '**double**'. Gli interi e i valori in virgola mobile di rango superiore, rimangono invariati.

È da osservare che, se si tenta di passare come argomento un valore che occupa uno spazio inferiore alla dimensione della parola del microprocessore, pur dichiarando tutti i parametri è molto probabile che il compilatore debba utilizzare ugualmente una parola intera, riempiendo in qualche modo lo spazio restante con dati nulli; pertanto, in presenza di parametri di dimensione non stabilita, è più che appropriata la promozione predefinita degli argomenti a valori multipli della parola.

Viene mostrato un esempio di programma contenente una funzione con un numero indefinito di parametri, nella quale, gli argomenti della chiamata vengono comunque estratti dalla pila dei dati, conoscendo le dimensioni usate nella chiamata. L'esempio funziona con un compilatore GNU C e serve solo per comprendere il meccanismo, ma per il momento non rappresenta il modo corretto di agire a questo proposito.

Listato 66.245. Per provare il codice attraverso un servizio *pastebin*: http://ideone.com/JzZOP .

```
#include <stdio.h>

void f (int w,...)
{
  //
  // Traduce l'indirizzo di «w» nel puntatore «p».
  //
  char *p = (char *) &w;
  //
  // Sposta il puntatore all'inizio del secondo parametro.
  //
  p = p + sizeof w;
  //
  // Mostra il valore del primo e del secondo parametro.
  //
  printf ("w = %i; ", w);
  printf ("x = %Lf; ", *((long double *)p));
  //
  // Sposta il puntatore all'inizio del terzo parametro.
  //
  p = p + sizeof (long double);
  //
  // Mostra il terzo parametro.
```

```
  //
  printf ("y = %lli; ", *((long long int *)p));
  //
  // Sposta il puntatore all'inizio del quarto parametro.
  //
  p = p + sizeof (long long int);
  //
  // Mostra il quarto parametro.
  //
  printf ("z = %i\n", *((int *)p));
  //
  return;
}

int main (int argc, char *argv[])
{
  f (10, (long double) 12.34, (long long int) 13, 14);
  return 0;
}
```

Come si vede, per raggiungere gli argomenti successivi al primo, conoscendo le loro caratteristiche, si scandisce in pratica la memoria occupata dalla pila dei dati, prendendo come riferimento l'indirizzo del primo parametro, il quale costituisce il riferimento certo. Si misura la dimensione del primo parametro e si aggiusta il puntatore in modo da posizionarsi dopo la fine di questo, sapendo che da lì in poi si trovano gli argomenti successivi. Il puntatore è di tipo '**`char *`**', in modo da poterlo gestire a unità di «caratteri», conformemente al valore prodotto dall'operatore '**`sizeof`**'. Se tutto funziona come previsto, il programma mostra correttamente il messaggio seguente:

```
w = 10; x = 12.340000; y = 13; z = 14
```

Il modo corretto di estrapolare i valori dei parametri non dichiarati richiede l'uso di alcune macroistruzioni della libreria standard, contenute nel file di intestazione '`stdarg.h`'. Si osservi come va trasformato l'esempio già apparso per rispettare la formalità standard:

Listato 66.247. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/22g6f9rU* , *http://ideone.com/Xuy6s* .

```
#include <stdio.h>
#include <stdarg.h>

void f (int w,...)
{
  //
  // Dichiara le variabili che servono a contenere
  // gli argomenti privi di parametri formali.
  //
  long double x;
  long long int y;
  int z;
  //
  // Dichiara il puntatore ai parametri.
  //
  va_list ap;
  //
  // Posiziona il puntatore dopo il primo parametro,
  // ovvero dopo l'ultimo parametro dichiarato
  // esplicitamente.
  //
  va_start (ap, w);
  //
  // Estrapola il secondo argomento della chiamata (portando
  // avanti il puntatore di conseguenza.
  //
  x = va_arg (ap, long double);
  //
  // Mostra il valore del primo e del secondo argomento
  // ottenuto dalla chiamata della funzione.
  //
  printf ("w = %i; ", w);
  printf ("x = %Lf; ", x);
  //
  // Estrapola il terzo argomento.
  //
  y = va_arg (ap, long long int);
```



```
  //
  // Mostra il terzo argomento.
  //
  printf ("y = %lli; ", y);
  //
  // Estrapola il quarto argomento.
  //
  z = va_arg (ap, int);
  //
  // Mostra il quarto e ultimo argomento.
  //
  printf ("z = %i\n", z);
  //
  // Conclude la scansione degli argomenti.
  //
  va_end (ap);
  //
  return;
}

int main (int argc, char *argv[])
{
  f (10, (long double) 12.34, (long long int) 13, 14);
  return 0;
}
```

Come si vede, è necessario incorporare il file di intestazione '`stdarg.h`' della libreria standard. All'inizio della funzione si dichiara una variabile di tipo '**`va_list`**' per scandire l'elenco di parametri: si tratta evidentemente di un puntatore (molto probabilmente al tipo '**`char`**'). Subito dopo si inizializza la variabile da usare per la scansione con la macroistruzione '**`va_start`**' che ha l'apparenza di una funzione. A '**`va_start`**' viene passata la variabile da usare come puntatore per gli argomenti e l'ultimo parametro dichiarato espressamente nella funzione, allo scopo di aggiornare il puntatore e di portarlo all'inizio del primo argomento privo di un parametro esplicito. Successivamente si utilizza la macroistruzione '**`va_arg`**', anche questa con l'apparenza di una funzione, per estrapolare l'argomento a cui punta la variabile di tipo '**`va_list`**', usata per lo scopo, aggiornando conseguentemente la variabile-puntatore, in modo da essere pronta per l'argomento successivo. Al termine si usa '**`va_end`**', la quale può essere indifferentemente una macroistruzione o una funzione vera e propria, allo scopo di concludere l'uso del puntatore dichiarato per la scansione dei parametri.

Le macroistruzioni '**`va_start`**' e '**`va_arg`**' non potrebbero essere realizzate in forma di funzioni. Infatti, '**`va_start`**' utilizza apparentemente come argomento l'ultimo parametro della funzione, ma per calcolare la posizione del parametro successivo servirebbe invece l'indirizzo di tale variabile. In modo analogo, la macroistruzione '**`va_arg`**' richiede l'indicazione del tipo di dati da estrarre, mentre una funzione vera potrebbe accettare solo la dimensione restituita dall'operatore '**`sizeof`**'; inoltre restituisce un valore dello stesso tipo, mentre una funzione vera può restituire un solo tipo prestabilito.

Nell'esempio non si vede cosa accade quando si trasmette un argomento costituito da un carattere ('**`char`**'). In tal caso bisogna tenere in considerazione l'effetto della promozione a intero; pertanto, la macroistruzione ***`va_arg`*** va usata indicato un tipo '**`int`**' (e non un tipo '**`char`**'). Lo stesso dicasi per i valori in virgola mobile, che vanno estratti prevedendo un formato '**`double`**', anche se nell'argomento originale dovesse trattarsi di '**`float`**' (e ammesso che l'argomento non sia espresso in un formato ancora più grande).

66.6.4 Annotazioni su «printf()» e altre funzioni simili

Da quanto descritto a proposito della promozione dei valori numerici, interi o in virgola mobile, si comprende che le rappresentazioni di valori numerici vanno fatte preferibilmente a partire da interi di tipo '**`int`**' o da valori in virgola mobile di tipo '**`double`**'. Si osservino gli esempi seguenti:

- `printf ("%hd\n", 123);`
  in linea di principio, lo specificatore di conversione '**%hd**' attende un valore di tipo '**short int**', ma il valore 123 che gli viene fornito è implicitamente di tipo '**int**';

- `printf ("%hd\n", (short int) 123);`
  esattamente come nell'esempio precedente e a nulla serve il tentativo di indicare un cast nell'argomento della chiamata alla funzione;

- `printf ("%c\n", 'A');`
  lo specificatore di conversione '**%c**' si attende un valore di tipo '**char**' (con o senza segno), ma il carattere '**A**' che gli viene fornito è implicitamente di tipo '**int**'.

Nel caso della funzione ***scanf()***, questi problemi non ci sono, perché gli argomenti variabili sono costituiti tutti da puntatori ad aree di memoria che devono essere in grado di contenere le informazioni da inserire.

66.6.5 Costante predefinita «__func__»

Lo standard del linguaggio prescrive che, se all'interno di una funzione viene usato il nome '**__func__**', questo si deve tradurre nel nome della funzione che lo contiene. In pratica, il compilatore che incontra questo nome, dichiara automaticamente, all'interno della funzione, la costante seguente:

```
static const char __func__[] = "nome_funzione";
```

L'esempio seguente mostra in che modo se ne potrebbe fare uso.

Listato 66.251. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/1I0KIqiq* , *http://ideone.com/A9est* .

```
#include <stdio.h>
void f (void)
{
  printf ("Sono nella funzione \"%s\".\n", __func__);
}
int main (int argc, char *argv[])
{
  f ();
  return 0;
}
```

Una volta compilato il programma, eseguendolo si ottiene:

```
Sono nella funzione "f".
```

66.7 Struttura, unione, campo, enumerazione, costante composta

Fino a questo punto sono stati incontrati solo i tipi di dati primitivi, oltre agli array di questi (incluse le stringhe). Nel linguaggio C, come in altri, è possibile definire dei tipi di dati aggiuntivi, derivati dai tipi primitivi.

Qui si usa la convenzione di nominare le strutture, le unioni e le enumerazioni con una lettera iniziale maiuscola. Per quanto riguarda invece i tipi di dati derivati, ottenuti con l'istruzione '**typedef**', si segue l'uso comune di aggiungere l'estensione '**_t**'.

66.7.1 Enumerazioni

È possibile dichiarare una variabile di tipo enumerativo, costituita tecnicamente da un intero, la quale può rappresentare solo un insieme prestabilito di valori, indicati simbolicamente attraverso delle definizioni. I valori simbolici che possono essere rappresentati sono tradotti in un numero intero, ma il programmatore non dovrebbe avere la necessità di avere a che fare direttamente con tali valori numerici corrispondenti. In altri termini, il tipo enumerativo è una forma di rappresentazione di un intero attraverso costanti mnemoniche.


```
enum nome { costante [, costante ]...}
```

La sintassi indicata mostra il modo in cui si definisce un tipo del genere: all'interno di parentesi graffe si elencano i nomi delle costanti che possono essere assegnate a una variabile di questo tipo. Tuttavia, alle costanti si può associare un valore intero in modo esplicito; pertanto, la costante può essere espressa così:

```
nome_simbolico [=n]
```

Si osservi l'esempio seguente che comunque non rappresenta un programma completo:

```
…
enum Colore { nero, marrone, rosso, arancio, giallo, verde,
              blu, viola, grigio, bianco, argento=100,
              oro };
…
enum Colore c;    // Dichiara la variabile «c».
…
c = marrone + 1;  // Assegna a «c» il valore successivo a
                  // «marrone»; in pratica assegna il valore
                  // «rosso».
…
if (c <= rosso);                      // Se il colore va dal
  {                                   // nero al rosso,
    printf ("Non mi piace: %i\n", c); // visualizza un
  }                                   // messaggio e mostra
                                      // anche il numero
                                      // corrispondente.
…
```

All'inizio viene dichiarato il tipo enumerativo '**Colore**', come insieme di colori principali, definiti simbolicamente per nome. Va osservato che nel caso dell'argento, viene associato espressamente il valore 100.

In mancanza di associazioni esplicite tra il valore simbolico e valore numerico, il compilatore associa al primo dei simboli il valore zero e dà a quelli successivi un numero ottenuto incrementando di una unità quello precedente. Nel caso dell'esempio, nero corrisponde a zero, marrone a uno, rosso a due e così di seguito fino al bianco. Il colore argento è definito espressamente (quindi dal nove del bianco si salta al 100 dell'argento) e il colore dell'oro viene determinato implicitamente come pari a ***argento***+1, ovvero uguale a 101.

Seguendo l'esempio si vede la dichiarazione della variabile ***c*** di tipo '**enum Colore**'. In pratica, viene dichiarata una variabile di tipo intero, in grado di contenere i valori dell'enumerazione '**Colore**'.

Successivamente si assegna alla variabile ***c*** la somma tra la costante ***marrone*** (pari a uno) e il numero uno. In pratica si assegna il valore due, ma in base al contesto si intende di avere assegnato ***rosso***.

Alla fine dell'esempio si vede un confronto tra la variabile ***c*** e un colore di quelli definiti simbolicamente. Di fatto si sta confrontando il valore della variabile con il numero due, ma in pratica sembra di valutare la cosa solo sul piano della sequenza ideale che è stata attribuita a quei colori.

La dichiarazione di una variabile enumerativa coincide quindi con la dichiarazione di un insieme di costanti simboliche, le quali non possono essere ridefinite. Pertanto, non è possibile dichiarare due variabili diverse che condividono costanti simboliche con lo stesso nome, a meno di essere in un campo di azione differente:

```
enum Colori { nero, marrone, rosso, arancio, giallo, verde,
              blu, viola, grigio, bianco };

enum Bianco_e_nero { nero, bianco };    // Non si può.
```

Le costanti simboliche definite attraverso le enumerazioni, possono essere usate anche al di fuori delle variabili dichiarate espressamente per questo scopo, purché possano ragionevolmente contenerne il valore. È anche evidente che al posto delle enumerazioni definite in questo modo sia possibile gestire direttamente le costanti. L'esem-

pio seguente riporta i passi equivalenti di quanto già visto all'inizio della sezione:

```
…
const int nero    = 0;
const int marrone = 1;
const int rosso   = 2;
const int arancio = 3;
const int giallo  = 4;
const int verde   = 5;
const int blu     = 6;
const int viola   = 7;
const int grigio  = 8;
const int bianco  = 9;
const int argento = 100;
const int oro     = 101;
…
int c;            // Dichiara la variabile «c».
…
c = marrone + 1;  // Assegna a «c» il valore successivo a
                  // «marrone»; in pratica assegna il valore
                  // «rosso».
…
if (c <= rosso);                    // Se il colore va dal
  {                                 // nero al rosso,
    printf ("Non mi piace: %i\n", c); // visualizza un
  }                                 // messaggio e mostra
                                    // anche il numero
                                    // corrispondente.
…
```

66.7.2 Strutture

Gli array sono sequenze di elementi uguali, tutti adiacenti nel modello di rappresentazione della memoria, ideale o reale che sia. In modo simile si possono definire strutture di dati più complesse in cui gli elementi adiacenti siano di tipo differente. Gli elementi che compongono una struttura sono i suoi ***membri***. In pratica, una struttura è una sorta di mappa di accesso a un'area di memoria, attraverso i suoi membri.

La variabile contenente una struttura si comporta in modo analogo alle variabili di tipo primitivo, per cui, la variabile che è stata creata a partire da una struttura, rappresenta tutta la zona di memoria occupata dalla struttura stessa e non solo il riferimento al suo inizio. Questa distinzione è importante, per non fare confusione con il comportamento relativo agli array che sono sostanzialmente solo dei puntatori.

La dichiarazione di una struttura si articola in due fasi: la dichiarazione del tipo e la dichiarazione delle variabili che utilizzano quella struttura.

```
struct Datario { int giorno; int mese; int anno; };
```

L'esempio mostra la dichiarazione della struttura '**`Datario`**' (ovvero del tipo '**`struct Datario`**') composta da tre interi dedicati a contenere rispettivamente: il giorno, il mese e l'anno. In questo caso, trattandosi di tre elementi dello stesso tipo, sarebbe stato possibile utilizzare un array, ma come è possibile vedere in seguito, una struttura può essere conveniente anche in queste situazioni.

È importante osservare che le parentesi graffe sono parte dell'istruzione di dichiarazione della struttura e non rappresentano un blocco di istruzioni. Per questo motivo appare il punto e virgola finale, cosa che potrebbe sembrare strana, specialmente quando la struttura si articola su più righe come nell'esempio seguente:

```
struct Datario {
    int giorno;
    int mese;
    int anno;
};              // Il punto e virgole finale è necessario.
```

La dichiarazione delle variabili che utilizzano la struttura può avvenire contestualmente con la dichiarazione della struttura, oppure in un momento successivo. L'esempio seguente mostra la dichiarazione del tipo '**`struct Datario`**', seguito da un elenco di variabili che utilizzano quel tipo: ***inizio*** e ***fine***.


```
struct Datario {
    int giorno;
    int mese;
    int anno;
} inizio, fine;
```

Tuttavia, il modo più elegante per dichiarare delle variabili a partire da una struttura è quello seguente:

```
struct Datario inizio, fine;
```

Quando una variabile è stata definita come organizzata secondo una certa struttura, si accede ai suoi componenti attraverso l'indicazione del nome della variabile stessa, seguita dall'operatore punto ('`.`') e dal nome dell'elemento particolare.

```
inizio.giorno = 1;
inizio.mese = 1;
inizio.anno = 2012;
…
fine.giorno = inizio.giorno;
fine.mese = inizio.mese +1;
fine.anno = inizio.anno;
```

Una struttura può essere dichiarata in modo anonimo, definendo immediatamente tutte le variabili che fanno uso di quella struttura. La differenza sta nel fatto che la struttura non viene nominata nel momento della dichiarazione e, dopo la definizione dei suoi elementi, devono essere elencate tutte le variabili in questione. Evidentemente, non c'è la possibilità di riutilizzare questa struttura per altre variabili definite in un altro punto, ma soprattutto, come viene mostrato in seguito, diventa impossibile indicare il tipo di struttura come parametro formale di una funzione.

```
struct {
    int giorno;
    int mese;
    int anno;
} inizio, fine;
```

66.7.3 Assegnamento, inizializzazione, campo di azione e puntatori delle strutture

Nella sezione precedente si è visto come accedere ai vari componenti della struttura, attraverso una notazione che utilizza l'operatore punto. Volendo è possibile assegnare a una variabile di questo tipo l'intero contenuto di un'altra che appartiene alla stessa struttura:

```
inizio.giorno = 1;
inizio.mese = 1;
inizio.anno = 2012
…
fine = inizio;
fine.mese++;
```

L'esempio mostra l'assegnamento alla variabile ***fine*** di tutta la variabile ***inizio***. Questo è ammissibile solo perché si tratta di variabili dello stesso tipo, cioè di strutture di tipo '**`Datario`**' (come deriva dagli esempi precedenti). Se invece si trattasse di variabili costruite a partire da strutture differenti, anche se realizzate nello stesso modo, con gli stessi membri, ciò non sarebbe ammissibile.

```
…
struct Datario {int giorno; int mese; int anno;};
struct Giorno  {int giorno; int mese; int anno;};
…
struct Datario ingresso = {31, 12, 2007};
struct Giorno  uscita;
uscita = ingresso;      // Errore: i dati sono incompatibili
…
```

Nel momento della dichiarazione di una struttura, è possibile anche inizializzarla utilizzando una forma simile a quella disponibile per gli array:

```
struct Datario inizio = { 1, 1, 2012 };
```

Oppure, per essere precisi e non dipendere dall'ordine dei campi nella struttura:

```
struct Datario inizio = { .giorno=1, .mese=1, .anno=2012 };
```

Dal momento che le strutture sono tipi di dati nuovi, per poterne fare uso occorre che la dichiarazione relativa sia accessibile a tutte le parti del programma che hanno bisogno di accedervi. Probabilmente, il luogo più adatto è al di fuori delle funzioni, eventualmente anche in un file di intestazione realizzato appositamente.

Ciò dovrebbe bastare a comprendere che le variabili che contengono una struttura vengono passate regolarmente attraverso le funzioni, purché la dichiarazione del tipo corrispondente sia precedente ed esterno alla descrizione delle funzioni stesse.

```
…
struct Datario { int giorno; int mese; int anno; };
…
void elabora (struct Datario oggi)
{
    …
}
```

L'esempio seguente che rappresenta un programma completo, serve a dimostrare che, nella chiamata di una funzione, la struttura viene passata per valore (e non per riferimento come avviene con gli array).

Listato 66.267. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/YZqBySyx* , *http://ideone.com/ffO5U* .

```
#include <stdio.h>

struct Datario {int giorno; int mese; int anno;};

void f (struct Datario d)
{
    unsigned int indirizzo = (int) &d;
    d.giorno = 28;
    d.mese = 2;
    d.anno = 2007;
    printf ("data %i-%i-%i inserita all'indirizzo %u\n",
             d.giorno, d.mese, d.anno, indirizzo);
}

int main (void)
{
    struct Datario data = {31, 12, 2007};
    unsigned int ind = (int) &data;
    f (data);
    printf ("data %i-%i-%i inserita all'indirizzo %u\n",
             data.giorno, data.mese, data.anno, ind);
    return 0;
}
```

Se si esegue il programma si ottiene un messaggio simile a quello seguente, dove si vede che gli l'indirizzi delle variabili contenenti la struttura, prima della chiamata della funzione e all'interno della stessa, sono differenti:

```
data 28-2-2007 inserita all'indirizzo 3212916960
data 31-12-2007 inserita all'indirizzo 3212916992
```

D'altro canto, se la variabile fosse la stessa, le modifiche fatte all'interno della funzione sarebbero visibili anche dopo la chiamata.

Così come nel caso dei tipi primitivi, anche con le strutture si possono creare dei puntatori. La loro dichiarazione avviene in modo intuitivo, come nell'esempio seguente:

```
struct Datario *p_data_fattura;
…
p_data_fattura = &inizio;
…
```

Quando si utilizza un puntatore a una struttura, diventa un po' più difficile fare riferimento ai vari componenti della struttura stessa, perché l'operatore punto ('`.`') che serve a unire il nome della struttura a quello dell'elemento, ha priorità rispetto all'asterisco che si utilizza per dereferenziare il puntatore:

```
*p_data_fattura.giorno = 15; // Non è valido!
```

L'esempio appena mostrato, non è ciò che sembra, perché l'asterisco posto davanti viene valutato dopo l'elemento

'**`p_data_fattura.giorno`**', il quale non esiste. Per risolvere il problema si possono usare le parentesi, come nell'esempio seguente:

```
(*p_data_fattura).giorno = 15; // Corretto.
```

In alternativa si può usare l'operatore '**`->`**', fatto espressamente per i puntatori a una struttura:

```
p_data_fattura->giorno = 15; // Corretto.
```

L'esempio seguente è una variante di quello già presentato in precedenza per dimostrare il passaggio per valore delle variabili che contengono una struttura. Ma in questo caso, il passaggio dei dati avviene esplicitamente per riferimento.

Listato 66.273. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/vW8ktxAg* , *http://ideone.com/dYBHZ* .

```
#include <stdio.h>

struct Datario {int giorno; int mese; int anno;};

void f (struct Datario *d)
{
    unsigned int indirizzo = (int) d;
    d->giorno = 28;
    d->mese = 2;
    d->anno = 2007;
    printf ("data %i-%i-%i inserita all'indirizzo %u\n",
             d->giorno, d->mese, d->anno, indirizzo);
}

int main (void)
{
    struct Datario data = {31, 12, 2007};
    unsigned int ind = (int) &data;
    f (&data);
    printf ("data %i-%i-%i inserita all'indirizzo %u\n",
             data.giorno, data.mese, data.anno, ind);
    return 0;
}
```

In tal caso, gli indirizzi della struttura appaiono uguali e le modifiche applicate all'interno della funzione si riflettono nella variabile originale:

```
data 28-2-2007 inserita all'indirizzo 3214580384
data 28-2-2007 inserita all'indirizzo 3214580384
```

66.7.4 Scostamento all'interno delle strutture

Il file '`strdef.h`' della libreria standard definisce una macroistruzione che, attraverso la parvenza di una funzione, consente di misurare lo scostamento di un membro della struttura, rispetto all'inizio della stessa:

```
offsetof (tipo, membro)
```

Si osservi l'esempio seguente.

Listato 66.275. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/wW7KtJB2* , *http://ideone.com/QJrQv* .

```
#include <stdio.h>
#include <stddef.h>

struct Elenco {
    char  uno;
    short due;
    int   tre;
};

int main (int argc, char *argv[])
{
    size_t offset = offsetof (struct Elenco, due);
    printf ("Il membro \"due\" si trova %i byte dopo "
            "l'inizio della struttura.\n", offset);
    return 0;
}
```

Come si può vedere, la macroistruzione ***offsetof*** produce un risultato di tipo '**size_t**'. Supponendo che il compilatore allinei i membri della struttura secondo multipli di due byte, il messaggio emesso dal programma potrebbe essere così:

```
Il membro "due" si trova 2 byte dopo l'inizio della struttura.
```

Pertanto, in questo caso, dopo il membro '**uno**' c'è un byte inutilizzato prima del membro '**due**'.

È il caso di ribadire che '**offsetof**' è una macroistruzione, ottenuta tramite le funzionalità del precompilatore. Diversamente, è probabile che sia impossibile realizzare una funzione che si comporti nello stesso modo apparente.

66.7.5 Unioni

L'unione permette di definire un tipo di dati accessibile in modi diversi, gestendolo come se si trattasse contemporaneamente di tipi differenti. La dichiarazione è simile a quella della struttura; quello che bisogna tenere a mente è che si fa riferimento alla stessa area di memoria; pertanto, lo spazio occupato è pari a quello del membro più grande.

```
union Livello {
    char c;
    int  i;
};
```

Si immagini, per esempio, di voler utilizzare indifferentemente una serie di lettere alfabetiche, oppure una serie di numeri, per definire un livello di qualcosa («A» equivalente a uno, «B» equivalente a due, ecc.). Le variabili generate a partire da questa unione, possono essere gestite nei modi stabiliti, come se fossero una struttura, ma condividendo la stessa area di memoria.

```
union Livello carburante;
```

L'esempio mostra in che modo si possa dichiarare una variabile di tipo '**union Livello**', riferita all'omonima unione. Il bello delle unioni sta però nella possibilità di combinarle con le strutture.

```
struct Livello {
    char tipo;
    union {
        char c;         // Usato se tipo == 'c'.
        int  i;         // Usato se tipo == 'n'.
    };
};
```

L'esempio non ha un grande significato pratico, ma serve a chiarire le possibiltà. La variabile ***tipo*** serve ad annotare il tipo di informazione contenuta nell'unione, se di tipo carattere o numerico. L'unione viene dichiarata in modo anonimo come appartenente alla struttura.

L'esempio successivo, che è completo, permette di verificare l'ordine con cui vengono memorizzati i byte in memoria. L'unione dichiarata parte dal presupposto che un numero '**short int**' utilizzi l'equivalente di due caratteri:

Listato 66.280. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/K6to4aa3* , *http://ideone.com/YJk2f* .

```
#include <stdio.h>

union Little_big {
    short int i;        // 16 bit
    char c[2];          // 8 bit, 8 bit
};

int main (void)
{
    union Little_big lb;
    lb.i = 0x1234;
    printf ("%x %x%x\n", lb.i, lb.c[0], lb.c[1]);
    return 0;
}
```


Eseguendo il programma in un elaboratore con architettura *little endian* si ottiene il risultato seguente:

```
1234 3412
```

66.7.6 Campi

All'interno di una struttura è possibile definire l'accesso a ogni singolo bit di un tipo di dati determinato, oppure a gruppetti di bit. In pratica viene dato un nome a ogni bit o gruppetto.

```
struct Luci {
    unsigned char
        b0      :1,
        b1      :1,
        b2      :1,
        b3      :1,
        b4      :1,
        b5      :1,
        b6      :1,
        b7      :1;
};
```

L'esempio mostra l'abbinamento di otto nomi ai bit di un tipo '**char**'. Il primo, ***b0***, rappresenta il bit più a destra, ovvero quello meno significativo. Se il tipo '**char**' occupasse una dimensione maggiore di 8 bit, la parte eccedente verrebbe semplicemente sprecata.

```
struct Luci salotto;
…
salotto.b2 = 1;
```

L'esempio mostra la dichiarazione della variabile ***salotto*** come appartenente alla struttura mostrata sopra, quindi l'assegnamento del terzo bit a uno, probabilmente per «accendere» la lampada associata.

Volendo indicare un gruppo di bit maggiore, basta aumentare il numero indicato a fianco dei nomi dei campi, come nell'esempio seguente:

```
struct Prova {
    unsigned char
        b0      :1,
        b1      :1,
        b2      :1,
        stato   :4;
};
```

Nell'esempio appena mostrato, si usano i primi tre bit in maniera singola (per qualche scopo) e altri quattro per contenere un'informazione «più grande». Ciò che resta (probabilmente solo un bit) viene semplicemente ignorato.

66.7.7 Istruzione «typedef»

L'istruzione '**typedef**' permette di definire un nuovo di tipo di dati, in modo che la sua dichiarazione sia più agevole. Lo scopo di tutto ciò sta nell'informare il compilatore; '**typedef**' non ha altri effetti. La sintassi del suo utilizzo è molto semplice:

```
typedef tipo nuovo_tipo;
```

Si osservi l'esempio seguente:

```
typedef int numero_t;
numero_t x, y, z;
```

In questo modo viene definito il nuovo tipo '**numero_t**', corrispondente in pratica a un tipo intero, con il quale si dichiarano tre variabili: ***x***, ***y*** e ***z***. Le tre variabili sono di tipo '**numero_t**'. L'esempio seguente riguarda le enumerazioni:

```
typedef enum Colore { nero, marrone, rosso, arancio,
                      giallo, verde, blu, viola, grigio,
                      bianco } colore_t;
colore_t c, d;
```

In questo caso si definisce il tipo '**colore_t**', corrispondente a un'enumerazione con i nomi dei colori principali. Le variabili ***c*** e ***d***

vengono dichiarate con questa modalità. Dal momento che si usa '**typedef**', si potrebbe definire l'enumerazione in modo anonimo:

```
typedef enum { nero, marrone, rosso, arancio, giallo,
               verde, blu, viola, grigio, bianco } colore_t;
colore_t c, d;
```

L'esempio successivo riguarda le strutture:

```
struct Datario {
    int giorno;
    int mese;
    int anno;
};
typedef struct Datario data_t;
data_t inizio, fine;
```

Attraverso '**typedef**' è stato definito il tipo '**data_t**', facilitando così la dichiarazione delle variabili ***inizio*** e ***fine***. Ma in questo caso, si presta di più una struttura anonima:

```
typedef struct {
    int giorno;
    int mese;
    int anno;
} data_t;
data_t inizio, fine;
```

Tradizionalmente, i nomi dei tipi di dati creati con l'istruzione '**typedef**' hanno estensione '**_t**'.

66.7.8 Costanti letterali composte

È possibile rappresentare un array o una struttura attraverso una costante letterale, nota come ***costante letterale composta***. Formalmente si definisce la costante letterale composta secondo il modello seguente, dove le parentesi graffe fanno parte della definizione:

(*tipo*) { *valore* [, *valore*] }

Per comprenderne l'utilizzo servono degli esempi e il caso più semplice riguarda la definizione degli array:

```
int *p = (int []) {3, 5, 76};
```

In questo modo si dichiara un array di interi, contenente rispettivamente i valori 3, 5 e 76, il cui indirizzo iniziale viene assegnato al puntatore ***p***. La variante seguente fa sì che il contenuto dell'array non possa essere modificato, ma per questo deve rendere altrettanto invariabile il contenuto raggiunto attraverso il puntatore:

```
const int *p = (const int []) {3, 5, 76};
```

Un array in forma letterale può essere trasmesso a una funzione. Quello che segue è un programma completo per dimostrare tale possibilità.

Listato 66.292. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/EkefhL40* , *http://ideone.com/dx4fQ* .

```
#include <stdio.h>

void f (int i[])
{
  printf ("i: %i %i %i\n", i[0], i[1], i[2]);
}

int main (int argc, char *argv[])
{
  f ((int []) {1, 3, 7});
  return 0;
}
```

In pratica, la funzione ***f()*** viene chiamata passando come argomento un array di tre interi, il quale logicamente viene trasmesso solo attraverso il puntatore al primo dei suoi elementi.

In modo analogo si possono rappresentare le strutture, ma in tal caso occorre disporre di un modello di riferimento, come si può vedere nell'esempio seguente che costituisce un altro programma completo.

Listato 66.293. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/1aTWnv01* , *http://ideone.com/PLEi0* .

```
#include <stdio.h>

struct Elenco {
    char  uno;
    short due;
    int   tre;
};

int main (int argc, char *argv[])
{
  struct Elenco e;
  e = (struct Elenco) { 33, 55, 77 };
  printf ("struttura: %i %i %i\n", e.uno, e.due, e.tre);
  return 0;
}
```

Ma naturalmente, i valori della struttura possono essere abbinati esplicitamente ai componenti a cui appartengono:

```
…
  e = (struct Elenco) { .uno=33, .tre=77, .due=55 };
…
```

Come per il caso degli array, anche le strutture rappresentate in forma letterale possono essere usate tra gli argomenti di una funzione. L'esempio seguente fa la stessa cosa di quello appena mostrato, con la differenza che si avvale di una funzione per ottenere lo scopo.

Listato 66.295. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/4DUDe1bG* , *http://ideone.com/33Gx33G* .

```
#include <stdio.h>

struct Elenco {
    char  uno;
    short due;
    int   tre;
};

void f (struct Elenco e)
{
  printf ("struttura: %i %i %i\n", e.uno, e.due, e.tre);
}

int main (int argc, char *argv[])
{
  f ((struct Elenco) { 33, 55, 77 });
  return 0;
}
```

Anche in questo caso, naturalmente, si possono rendere espliciti i componenti della struttura a cui si attribuiscono i valori:

```
…
  f ((struct Elenco) { .uno=33, .tre=77, .due=55 });
…
```

A differenza dell'array, la struttura che si trova tra gli argomenti di una funzione viene passata integralmente; volendo trasmettere solo il suo indirizzo, si può usare l'operatore '**&**', come nell'esempio seguente.

Listato 66.297. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/FJnL564M* , *http://ideone.com/mcodf* .

```
#include <stdio.h>

struct Elenco {
    char  uno;
    short due;
    int   tre;
};

void f (struct Elenco *e)
{
```

```
  printf ("struttura: %i %i %i\n", e->uno, e->due, e->tre);
}

int main (int argc, char *argv[])
{
  f (&(struct Elenco) {.uno=33, .tre=77, .due=55});
  return 0;
}
```

66.8 Tipi di dati speciali, di uso comune

Il linguaggio C prevede un insieme di tipi di dati tradizionali, a cui ci si riferisce con maggiore frequenza, e vari altri tipi, alcuni dei quali è bene conoscere.

66.8.1 Tipo «_Bool»

Lo standard C prevede un tipo particolare per la rappresentazione di valori logici, ovvero solo per i valori zero e uno. Nella tradizione del linguaggio, questo tipo manca e di norma si è rimediato rimpiazzandolo semplicemente con un valore intero, dal tipo '**char**' in su. Dal momento che è frequente l'uso di un tipo personalizzato (o di una macro-variabile del precompilatore) denominato ***bool***, lo standard ha inserito il tipo logico con il nome '**_Bool**', allo scopo di evitare conflitti con il codice esistente.

Il tipo '**_Bool**' può contenere solo i valori zero e uno; pertanto, la conversione di un numero di tipo diverso in un tipo '**_Bool**' avviene traducendo qualunque valore diverso da zero con il numero uno (*Vero*), mentre lo zero mantiene il suo valore invariato (*Falso*).

Lo standard non stabilisce come deve essere rappresentato in memoria il tipo '**_Bool**', anche se si tratta molto probabilmente di un byte intero che viene a essere sacrificato per lo scopo. Data la particolarità di questo tipo, non è detto che si possa utilizzare un puntatore per raggiungere l'area di memoria corrispondente.

Comprendendo il motivo per il quale questo tipo ha ricevuto un nome così particolare, diventa evidente che se lo si vuole utilizzare convenga creare una macro-variabile o un tipo derivato. D'altra parte, lo stesso file '`stdbool.h`' prescrive la definizione della macro-variabile '**bool**'.

In conclusione, se si desidera utilizzare un tipo di dati booleano, conviene fare riferimento alla macro-variabile ***bool***, la quale potrebbe anche essere ridefinita localmente nel proprio programma, se quello che si vuole non è conforme alle previsioni dello standard o delle librerie del proprio compilatore.

66.8.2 Tipo «void»

Il tipo '**void**' rappresenta un'eccezione tra i tipi di dati usati nel linguaggio, in quanto rappresenta formalmente una variabile di rango nullo, e come tale incapace di contenere qualunque valore. La situazione più frequente di utilizzo del tipo '**void**' riguarda le funzioni, quando non devono restituire alcun valore: in tal caso si dichiara che sono di tipo '**void**'.

```
void procedura (int x)
{
  …
  return;
}
```

L'esempio mostra una funzione che, non dovendo restituire alcun valore, viene dichiarata di tipo '**void**'. Come si vede, l'istruzione '**return**' va usata, in questo caso, senza l'indicazione di un valore.

Quando una funzione non richiede parametri, si deve indicare esplicitamente questo fatto con la parola chiave '**void**', come dire che esiste sì un parametro, ma di rango nullo e come tale privo di qualunque informazione:

```
int funzione (void)
{
  …
}
```


In questo esempio, la funzione restituisce un valore intero, ma non fa uso di alcun parametro.

Il cast di tipo '**void**' può servire per annullare il risultato di un'espressione, quando ciò che interessa della stessa sono solo i suoi «effetti collaterali». In altri termini, quando un'espressione esegue qualche tipo di operazione, ma complessivamente si vuole scartare il risultato che viene generato, si può usare un cast di tipo '**void**'. Per esempio, quando si vuole usare una funzione, la quale restituirebbe un valore, del quale non si vuole fare alcun uso, si può indicare nella chiamata un cast al tipo '**void**', anche se di norma ciò non è necessario:

```
…
    (void) mia_funzione (…);
…
```

È possibile definire un puntatore generico al tipo '**void**', sapendo che questo è convertibile in tutti gli altri tipi di puntatore, con un cast appropriato e che è sempre possibile fare anche l'inverso:

```
    …
    void *p;
    …
    p = (void *) &a;
    …
```

Il puntatore nullo può essere definito, sia come un valore intero pari a zero, sia come tale valore tradotto in un puntatore generico, ovvero '**void ***':

```
    …
    int NULL = 0;
    …
```

```
    …
    void *NULL = (void *) 0;
    …
```

Si osservi che un puntatore generico ('**void ***') non può essere incrementato o decrementato, perché fa riferimento a un'unità di memoria di dimensione nulla. Pertanto, per usare un puntatore del genere, quando si vuole scandire la memoria, prima va convertito in un puntatore di rango appropriato.

66.8.3 Tipo «size_t»

Secondo lo standard il tipo '**size_t**' è definito nel file '`stddef.h`', ma in pratica, dal momento che viene usato dall'operatore '**sizeof**', potrebbe essere incorporato direttamente nel compilatore, tra i tipi fondamentali. A ogni modo si tratta normalmente di un tipo equivalente a un '**unsigned long int**', destinato però a contenere la dimensione di qualcosa, intesa come intervallo tra due indirizzi (tra due puntatori), ma espressa come valore assoluto.

Listato 66.304. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/65fb65zpFR* , *http://ideone.com/9UNUA* .

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[])
{
    double a = 1.1;
    double b = 2.2;
    char * A = (char *) &a;
    char * B = (char *) &b;
    size_t s = abs (A - B);
    printf ("distanza: %i\n", s);
    return 0;
}
```

L'esempio mostra la dichiarazione di due variabili e di due puntatori alle variabili. Tuttavia, i puntatori sono di tipo '**char ***', in modo che la sottrazione tra i due dia la distanza in byte. Volendo, per non fare riferimento a un tipo particolare di puntatore, si potrebbe usare il tipo '**void**', ottenendo lo stesso risultato.

Listato 66.305. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/GFoNtYU5* , *http://ideone.com/bp9Qi* .

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[])
{
    double a = 1.1;
    double b = 2.2;
    void * A = (void *) &a;
    void * B = (void *) &b;
    size_t s = abs (A - B);
    printf ("distanza: %i\n", s);
    return 0;
}
```

Va osservato che il risultato mostrato dall'esecuzione dell'esempio compilato, dipende dal compilatore. In pratica, è il compilatore che decide come collocare in memoria le variabili; se si presume che siano adiacenti, si dovrebbe ottenere una distanza di otto byte.

66.8.4 Tipo «ptrdiff_t»

Per rappresentare la differenza tra due indirizzi, tenendo conto del segno, si usa il tipo '**ptrdiff_t**', definito anch'esso nel file '`stddef.h`'. Molto probabilmente si tratta di un tipo equivalente a un '**long int**'. Viene ripreso l'esempio già mostrato, senza calcolare il valore assoluto della differenza tra indirizzi.

Listato 66.306. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/WaunziA8* , *http://ideone.com/pzJC8* .

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
int main (int argc, char *argv[])
{
    double a = 1.1;
    double b = 2.2;
    void * A = (void *) &a;
    void * B = (void *) &b;
    ptrdiff_t s = (A - B);
    printf ("differenza: %i\n", s);
    return 0;
}
```

66.8.5 Tipo «va_list»

Il tipo '**va_list**' è definito dallo standard nel file di intestazione '`stdarg.h`', allo scopo di agevolare la scansione degli argomenti variabili, passati alle funzioni. Lo standard è vago sul significato che deve avere il tipo '**va_list**', ma in pratica dovrebbe trattarsi di un puntatore al tipo '**char**'.[23] Tuttavia il suo utilizzo rimane relegato alla scansione degli argomenti variabili, come descritto nella sezione 66.6.3. Viene comunque riportata qui la copia di un esempio che ne mostra l'uso.

Listato 66.307. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/5kUJnQxn* , *http://ideone.com/2W05Y* .

```
#include <stdio.h>
#include <stdarg.h>

void
f (int w,...)
{
    long double x;      // Dichiara le variabili che servono
    long long int y;    // a contenere gli argomenti per i
    int z;              // quali mancano i parametri formali.

    va_list ap;         // Dichiara il puntatore agli
                        // argomenti.

    va_start (ap, w);   // Posiziona il puntatore dopo la
                        // fine di «w».

    x = va_arg (ap, long double); // Estrae l'argomento
                                  // successivo portando
                                  // avanti il puntatore
                                  // di conseguenza.

    printf ("w = %i; ", w);     // Mostra il valore del
                                // primo parametro.
    printf ("x = %Lf; ", x);    // Mostra il valore
                                // dell'argomento successivo.

    y = va_arg (ap, long long int);  // Estrapola e mostra
    printf ("y = %lli; ", y);        // il terzo argomento.

    z = va_arg (ap, int);            // Estrapola e mostra
    printf ("z = %i\n", z);          // il quarto argomento.

    va_end (ap);                // Conclude la scansione.

    return;
}

int main (int argc, char *argv[])
{
    f (10, (long double)12.34, (long long int)13, 14);
    return 0;
}
```

66.8.6 Tipo «wchar_t»

Per rappresentare un carattere esteso, ovvero un carattere dell'insieme universale, non è sufficiente il tipo '**char**' e per questo esiste invece il tipo '**wchar_t**' (*wide character type*), definito nel file '`stddef.h`'.

Il tipo '**wchar_t**' è un intero, usato generalmente senza segno, di rango sufficiente a rappresentare tutti i caratteri che si intende di poter ammettere. È da osservare che per rappresentare l'insieme completo dei caratteri già definiti sono necessari anche più di 32 bit.

Il tipo '**wchar_t**' si usa sostanzialmente come il tipo '**char**', anche per ciò che riguarda gli array e le stringhe (che per essere tali devono essere terminate con il carattere nullo), ma si tratta sempre di una gestione interna, perché la rappresentazione richiede invece una trasformazione nella forma prevista dalla configurazione locale (sezione 66.9).

66.8.7 Tipo «wint_t»

Molte delle funzioni standard che in qualche modo hanno a che fare con un carattere singolo (perché ne ricevono il valore come argomento o perché restituiscono il valore di un carattere), lo fanno trattando il carattere come un tipo '**int**', ovvero, trattando il carattere senza segno e promuovendolo al rango di un intero normale. Questo sistema permette di distinguere tra tutti i caratteri dell'insieme ridotto e un valore ulteriore, rappresentato dalla macro-variabile ***EOF***, usata per rappresentare un errore in base al contesto.

Nella gestione dei caratteri estesi ci sono funzioni analoghe che svolgono lo stesso tipo di adattamento, ma in tal caso il valore del carattere viene gestito in qualità di '**wint_t**', il quale può rappresentare tutti i caratteri che sono ammessi dal tipo '**wchar_t**', con l'aggiunta del valore corrispondente a '**WEOF**' (diverso da tutti gli altri).

Il tipo '**wint_t**' e la macro-variabile ***WEOF*** sono definiti nel file '`wchar.h`'. Il tipo '**wint_t**' è, evidentemente, un intero di rango tale da consentire la rappresentazione di tutti i valori necessari.

66.8.8 Tipo «time_t»

Diverse funzioni dichiarate nel file '`time.h`' fanno riferimento al tipo '**time_t**' che rappresenta la quantità di unità di tempo trascorsa a partire da un'epoca di riferimento.

Frequentemente si tratta di un valore numerico intero che rappresenta la quantità di secondi trascorsi dall'epoca di riferimento (nei sistemi Unix è di norma l'ora zero del 1 gennaio 1970); inoltre, in un elaboratore che gestisca correttamente i fusi orari, è normale che questo valore sia riferito al tempo universale coordinato.

66.8.9 Tipo «struct tm»

La libreria standard, nel file '`time.h`', prescrive che sia definito il tipo '**`struct tm`**', con il quale è possibile rappresentare tutte le informazioni relative a un certo tempo, secondo le convenzioni umane:

```
struct tm {
    int tm_sec;     // Secondi:                 da 0 a 60.
    int tm_min;     // Minuti:                  da 0 a 59.
    int tm_hour;    // Ora:                     da 0 a 23.
    int tm_mday;    // Giorno del mese:         da 1 a 31.
    int tm_mon;     // Mese dell'anno:          da 0 a 11.
    int tm_year;    // Anno dal 1900.
    int tm_wday;    // Giorno della settimana: da 0 a 6
                    // con lo zero corrispondente alla
                    // domenica.
    int tm_yday;    // Giorno dell'anno:        da 0 a 365.
    int tm_isdst;   // Ora estiva. Contiene un valore
                    // positivo se è in vigore l'ora estiva;
                    // zero se l'ora è quella «normale»
                    // ovvero quella invernale;  un valore
                    // negativo se l'informazione non è
                    // disponibile.
};
```

66.8.10 Tipo «FILE»

Il tipo '**`FILE`**' rappresenta una variabile strutturata con tutte le informazioni necessarie a individuare un flusso di file aperto. Di norma vengono usati puntatori, ovvero variabili di tipo '**`FILE *`**', per tutte le operazioni di accesso relative a flussi di file aperti, tanto che nel gergo comune si confondono le cose e tali puntatori sono chiamati generalmente *stream*.

66.8.11 Tipo «fpos_t»

Alcune funzioni individuano la posizione di accesso ai file attraverso un insieme di dati. In quei casi, per rappresentare tale insieme di dati si usano variabili strutturate di tipo '**`fpos_t`**'.

66.9 Configurazione locale

La libreria standard del linguaggio C prevede la gestione della configurazione locale, attraverso l'indicazione di una stringa da associare a una ***categoria***, dove la categoria rappresenta il contesto particolare della configurazione locale a cui si vuole fare riferimento.

La stringa con cui si indica il tipo di configurazione desiderato, contiene le informazioni sulla lingua, la nazionalità e soprattutto la codifica da usare per la rappresentazione delle ***sequenze multibyte***. La codifica scelta condiziona l'insieme di caratteri che possono essere gestiti, sia attraverso le sequenze multibyte, sia attraverso i caratteri estesi.

66.9.1 Configurazione locale nei sistemi Unix e simili

In un sistema Unix o simile, la configurazione locale viene definita impostando alcune variabili di ambiente. Si tratta precisamente di variabili il cui nome inizia per '**`LC_`**…', dove in particolare la variabile '**`LC_ALL`**', se usata, prevale su tutte, mentre la variabile '**`LANG`**' (se '**`LC_ALL`**' non viene usata) serve per la configurazione predefinita di tutte le altre variabili '**`LC_`**…' che non fossero state dichiarate espressamente. A queste variabili di ambiente si associa una stringa secondo il formato seguente:

lingua_nazionalità . codifica

Per esempio, la configurazione '**`de_CH.UTF-8`**' rappresenta la configurazione di lingua tedesca per la Svizzera, con una codifica UTF-8.

Ogni variabile di ambiente '**`LC_`**…', esclusa '**`LC_ALL`**', rappresenta una categoria, ovvero un contesto particolare a cui applicare la configurazione locale. Per esempio, pur volendo gestire i numeri con una rappresentazione europea (con la virgola per i decimali), si potrebbe voler gestire le valute in dollari americani. Pertanto ci potrebbe essere un uso contrastante delle variabili '**`LC_NUMERIC`**' e '**`LC_MONETARY`**'.

66.9.2 Configurazione locale nel linguaggio C

Il linguaggio C non gestisce la configurazione locale attraverso le variabili di ambiente, perché non è detto che il sistema in cui si trova a operare il programma le preveda. Tuttavia definisce le categorie della configurazione locale attraverso macro-variabili (dichiarate nel file '`locale.h`') con gli stessi nomi e significati usati per le variabili di ambiente dei sistemi Unix e simili (vale anche il fatto che la macro-variabile ***LC_ALL*** si riferisca simultaneamente a tutte le categorie previste). Le macro-variabili in questione riguardano solo le categorie ***LC_…***, mentre la variabile di ambiente ***LANG*** non ha alcun corrispondente nel linguaggio e non rappresenta precisamente una categoria, ma solo un valore predefinito.

La configurazione locale di partenza per un programma scritto in linguaggio C è proprio la configurazione '**`C`**', la quale coincide sostanzialmente con la modalità di funzionamento tradizionale del linguaggio, con una codifica ASCII o equivalente. Per impostare la configurazione locale si usa la funzione ***setlocale()*** secondo il modello seguente:

```
char *setlocale (int categoria, const char *configurazione);
```

Il primo parametro è un numero intero che si indica normalmente attraverso una macro-variabile ***LC_…***; il secondo è una stringa, contenente la definizione della configurazione, per esempio '**`it_IT.UTF-8`**'. Se la funzione è nelle condizioni di accettare la configurazione richiesta, restituisce un puntatore alla stringa che definisce la configurazione stessa; altrimenti dà solo il puntatore nullo.

Come accennato, all'avvio ogni programma si trova a funzionare come se fosse stata usata la configurazione '**`C`**', ovvero come se fosse stata usata la funzione ***setlocale()*** così:

```
setlocale (LC_ALL, "C");
```

Per richiedere una configurazione più attuale e più utile, conviene specificare qualcosa che preveda la codifica UTF-8, con la quale è possibile rappresentare qualunque carattere della codifica universale:

```
setlocale (LC_ALL, "fr_CH.UTF-8");
```

Tuttavia, se il sistema operativo ha una gestione della configurazione locale, così come avviene nei sistemi Unix e simili, è meglio far sì che il programma erediti tale configurazione. Per ottenere questo, si usa la funzione ***setlocale()*** lasciando una stringa nulla (nel senso di vuota) al posto della configurazione richiesta:

```
setlocale (LC_ALL, "");
```

Per interrogare la configurazione locale attiva per una certa categoria (o per tutte se si fa riferimento a '**`LC_ALL`**'), è sufficiente fornire il puntatore nullo al posto della stringa. L'esempio seguente è completo e si vede anche l'incorporazione del file '`locale.h`', contenente il prototipo della funzione ***setlocale()*** e la dichiarazione delle macro-variabili ***LC_…***.

Listato 66.312. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/iomnt08Q* , *http://ideone.com/1Vk3V* .

```
#include <stdio.h>
#include <locale.h>
int main (int argc, char *argv[])
{
  setlocale (LC_ALL, "");
  char *loc;
  loc = setlocale (LC_ALL, NULL);
  printf ("LC_ALL: \"%s\"\n", loc);
  return 0;
}
```

Il programma potrebbe emettere il risultato seguente:

```
LC_ALL: "it_IT.UTF-8"
```

66.9.3 Caratteri multibyte e caratteri estesi

All'origine del linguaggio C esisteva una corrispondenza biunivoca tra carattere e byte. Attualmente, questa corrispondenza riguarda solo i caratteri dell'insieme minimo, il quale di norma coincide con quello della codifica ASCII. Per rappresentare caratteri che vanno al di fuori dell'insieme minimo, si usano due metodi nel linguaggio: le sequenze multibyte, in cui un carattere è rappresentato attraverso una sequenza di più byte o comunque attraverso l'inserzione di codici che cambiano di volta in volta il sottoinsieme di riferimento, e i caratteri estesi che richiedono una unità di memorizzazione con un rango maggiore del byte. L'esempio seguente mostra l'uso di una stringa multibyte:

```
    printf ("€àèìòαστ\n");
```

È il contesto che fa capire la natura della stringa. In pratica, il file sorgente che contiene i caratteri deve essere scritto utilizzando una qualche codifica che preveda l'uso di più byte per rappresentare un carattere. La stessa codifica è quella che il programma deve usare durante il funzionamento per interpretare correttamente la stringa multibyte fornita.

In questo caso particolare, la funzione ***printf()*** non ha nemmeno bisogno di rendersi conto della codifica; semplicemente, se il programma funziona secondo la configurazione corretta, la visualizzazione del messaggio avviene come previsto.

Esistono diversi modi di gestire delle sequenze multibyte per rappresentare caratteri particolari, ma alcune sono più difficili da amministrare, perché richiedono il passaggio a sottoinsiemi di caratteri differenti attraverso l'uso di codici speciali, a cui si fa riferimento con il termine *shift*. In pratica, in tali condizioni, quando deve essere interpretata una stringa contenente sequenze multibyte, le funzioni devono tenere traccia dello stato di questa interpretazione, per sapere a quale sottoinsieme particolare di caratteri si sta facendo riferimento. Pertanto, l'interruzione e la ripresa di tale interpretazione devono essere motivo di preoccupazione per il programmatore. Fortunatamente la tendenza è quella di usare la codifica UTF-8 per la rappresentazione dell'insieme universale dei caratteri, per tutte le lingue e tutte le nazionalità. Tale codifica ha il vantaggio di non richiedere la conservazione di uno stato (*shift status*), in quanto l'interpretazione di ogni carattere è indipendente dai precedenti: quello che è importante è evitare di spezzare l'interpretazione di un carattere a metà, ma anche se fosse, i caratteri successivi verrebbero individuati correttamente.

Dall'esempio mostrato si intende che una stringa multibyte si rappresenta letteralmente nello stesso modo di una stringa normale, con la differenza che la sua lunghezza in «caratteri», nel senso di unità '**`char`**', è maggiore dei caratteri che rappresenta. quindi, eventualmente, nel dimensionare un array di caratteri, occorre tenere conto di questo particolare.

Per rappresentare un carattere che va al di fuori dell'insieme minimo del linguaggio C, si può usare un carattere esteso, ovvero un valore intero di rango maggiore rispetto al tipo '**`char`**'. Si tratta precisamente del tipo '**`wchar_t`**' (*wide char*) che in condizioni normali va dai 16 ai 32 bit;.

Evidentemente, il rango del tipo '**`wchar_t`**' condiziona la quantità di caratteri che possono essere rappresentati. Per una rappresentazione abbastanza completa dell'insieme universale serve almeno un tipo '**`wchar_t`**' da 32 bit.

Si può rappresentare una costante letterale di tipo '**`wchar_t`**' mettendo anteriormente il prefisso '**`L`**'. Per esempio, '**`L'€'`**' viene convertito dal compilatore in un carattere esteso che rappresenta numericamente il simbolo dell'euro. In modo analogo è possibile costruire array di elementi '**`wchar_t`**', per contenere stringhe estese (stringhe di caratteri '**`wchar_t`**' concluse da un valore nullo di terminazione, come per le stringhe normali). Anche per rappresentare le stringhe estese in modo letterale si può usare il prefisso '**`L`**'. Per esempio, '**`L"àèìòù"`**' viene tradotto dal compilatore in una stringa estesa.

Listato 66.315. Per provare il codice attraverso un servizio *pastebin*: *http://ideone.com/eWHAz* .

```
#include <stdio.h>
#include <locale.h>
#include <wchar.h>
int main (int argc, char *argv[])
{
  setlocale (LC_ALL, "en_US.UTF-8");
  wchar_t wc = L'ß';
  wchar_t wcs[] = L"€àèìòαστ";
  printf ("%lc, %ls\n", wc, wcs);
  return 0;
}
```

L'esempio mostra l'uso delle costanti letterali riferite a caratteri e stringhe estese. In particolare, va osservato l'uso della funzione ***printf()***, in cui si indicano lo specificatore di conversione '**`%lc`**' per tradurre un carattere esteso e '**`%ls`**' per una stringa estesa. Ecco il risultato che si attende di visualizzare da quel programma:

```
ß, €àèìòαστ
```

A questo punto è bene sia chiaro un concetto logico ma non sempre evidente: per gestire caratteri al di fuori dell'insieme minimo, è necessario definire la configurazione locale con una codifica che sia tale da permetterlo. Pertanto, se non si usa la funzione ***setlocale()*** (così come invece avviene nell'esempio), si sta lavorando con la configurazione predefinita '**`C`**', per la quale non ci sono sequenze multibyte e diventa inutile l'uso del tipo '**`wchar_t`**'. Pertanto, se nell'esempio mancasse l'uso appropriato della funzione ***setlocale()***, non si otterrebbe la visualizzazione del testo come previsto.

66.9.4 Concatenamento eterogeneo

Il concatenamento di stringhe espresse in forma di costanti letterali, avviene, per le stringhe estese, esattamente come per le stringhe tradizionali, con l'eccezione che il concatenamento eterogeneo è ammissibile e implica sempre l'interpretazione di stringhe estese:

```
…
    wcp = "ciao amore" L"€àèìòαστ";
…
```

In questo caso, la variabile ***wcp*** riceve il puntatore a una stringa estesa contenente precisamente la sequenza «ciao amore€àèìòαστ», conclusa in modo appropriato.

Questo meccanismo consente, tra le altre cose, di concatenare delle macro-variabili che si espandono in stringhe letterali normali, in ogni circostanza, senza doverle duplicare per distinguerle in base al contesto.

66.9.5 Conversione tra caratteri multibyte e caratteri estesi

Un gruppo di funzioni dichiarate come prototipo nel file '`stdlib.h`' è importante per gestire la conversione tra caratteri multibyte e caratteri estesi. Le funzioni più importanti sono precisamente ***mbstowcs()***

(*Multibyte string to wide character string*) e ***wcstombs()*** (*Wide character string to multibyte string*), con lo scopo di convertire stringhe da multibyte a caratteri estesi e viceversa.

```
size_t mbstowcs (wchar_t *restrict wcs,
                 const char *restrict s,
                 size_t n);
```

```
size_t wcstombs (char *restrict s,
                 const wchar_t *restrict wcs,
                 size_t n);
```

La funzione '**mbstowcs**' si usa per convertire una stringa contenente sequenze multibyte in una stringa estesa, ovvero un array di elementi '**wchar_t**'. L'ultimo parametro rappresenta la quantità massima di caratteri estesi che devono essere inseriti nella stringa estesa di destinazione, contando anche il carattere nullo di terminazione. Il valore restituito è la quantità di caratteri che sono stati inseriti, escludendo il carattere nullo di terminazione, se c'è.

Listato 66.318. Per provare il codice attraverso un servizio *pastebin*: *http://ideone.com/Nx3eA* .

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
  setlocale (LC_ALL, "en_US.UTF-8");
  wchar_t wca[] = L"€€€€€€€€€€";
  wchar_t wcb[] = L"€€€€€€€€€€";
  size_t q;

  q = mbstowcs (wca, "äåâ", 3);
  printf ("mbstowcs: %i: \"%ls\"\n", q, wca);

  q = mbstowcs (wcb, "äåâ", 6);
  printf ("mbstowcs: %i: \"%ls\"\n", q, wcb);

  return 0;
}
```

L'esempio mostra la dichiarazione di due stringhe estese contenenti 10 caratteri estesi (oltre al carattere di terminazione della stringa). La funzione ***mbstowcs()*** viene usata la prima volta per tradurre la stringa multibyte '**L"äåâ"**' nei caratteri estesi corrispondenti all'inizio della prima delle due stringhe estese. Però, viene posto il limite al trasferimento di soli tre caratteri. Così facendo, il carattere di terminazione della stringa multibyte non viene convertito. Nel secondo caso, invece, si richiede il trasferimento di sei caratteri estesi, ma questo si ferma quando viene incontrato il carattere nullo di terminazione.

Entrambe le chiamate alla funzione ***mbstowcs()*** restituiscono il valore tre, perché sono solo tre i caratteri trasferiti, che siano diversi da quello di terminazione, ma nel secondo caso si può apprezzare la differenza nella stringa estesa risultante:

```
mbstowcs: 3: "äåâ€€€€€€€"
mbstowcs: 3: "äåâ"
```

La funzione ***wcstombs()*** funziona in modo opposto, per convertire una stringa estesa in una stringa multibyte. In questo caso, l'ultimo parametro rappresenta la quantità di byte che si vogliono ottenere con il trasferimento, incluso quello che rappresenta la terminazione della stringa. Logicamente, come nel caso dell'altra funzione, si ottiene la quantità di byte ottenuti dal trasferimento, ma senza contare il carattere nullo di terminazione.

Listato 66.320. Per provare il codice attraverso un servizio *pastebin*: *http://ideone.com/inlVK* .

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
```



```
int main (void)
{
  setlocale (LC_ALL, "en_US.UTF-8");
  char mba[] = "**********";
  char mbb[] = "**********";
  size_t n;

  n = wcstombs (mba, L"äåâ", 6);
  printf ("wcstombs: %i: \"%s\"\n", n, mba);

  n = wcstombs (mbb, L"äåâ", 9);
  printf ("wcstombs: %i: \"%s\"\n", n, mbb);

  return 0;
}
```

Questo nuovo esempio è analogo al precedente, ma invertendo il ruolo delle stringhe: questa volta la stringa estesa viene convertita in una stringa multibyte. Nel caso particolare della codifica UTF-8, ognuna delle lettere che si vedono nella stringa estesa si traduce in una sequenza di due byte; pertanto, la conversione richiede che siano convertiti almeno sette byte, per includere anche il carattere nullo di terminazione. Si può vedere che nel primo caso il carattere nullo non viene convertito, pertanto la stringa di destinazione continua ad apparire della lunghezza originale, pur con la prima parte sovrascritta. Naturalmente, rimangono solo quattro asterischi perché la sequenza multibyte necessaria a rappresentare quelle tre lettere è complessivamente di sei byte.

```
wcstombs: 6: "äåâ****"
wcstombs: 6: "äåâ"
```

La conversione, in un verso o nell'altro, può fallire. Se queste funzioni incontrano dei problemi, restituiscono l'equivalente di `-1` tradotto secondo il tipo '**size_t**' (in pratica, utilizzando una rappresentazione dei valori negativi in complemento a due, si ottiene il valore positivo massimo che la variabile possa rappresentare, essendo '**size_t**' senza segno).

Ci sono altri dettagli sull'uso di queste funzioni, ma si possono approfondire leggendo la sezione 69.9.11 e le pagine di manuale *mbstowcs(3) wcstombs(3)*.

66.10 Organizzazione dei file sorgenti

Quando si scrive un programma che non sia estremamente banale, diventa importante organizzare i file dei sorgenti in un modo gestibile. Se l'esperienza di programmazione da cui si proviene, quando ci si rivolge al C, è quella dei linguaggi interpretati, si può essere tentati di scrivere tutto il proprio programma in un file solo, ma questo approccio può essere controproducente. D'altra parte, per dividere il lavoro in più file, occorre che tale suddivisione abbia un senso pratico, conforme alla filosofia del linguaggio.

66.10.1 File di intestazione

La direttiva '**#include**' del precompilatore consente di incorporare un altro file, scritto secondo le regole del linguaggio, come se il suo contenuto facesse parte del file incorporante. Tradizionalmente questi file che vengono incorporati sono «file di intestazione», a cui si dà un'estensione diversa, '`.h`', proprio per distinguerne lo scopo.

Un file di intestazione, perché sia utile e non serva a creare maggiore confusione, può contenere la dichiarazione di macro-variabili, di macroistruzioni, di tipi derivati, di prototipi di funzione e di variabili pubbliche. Non ha senso inserire il codice completo delle funzioni all'interno di un file di intestazione, perché queste verrebbero replicate inutilmente nei file-oggetto, ogni volta che viene incorporato il file stesso.

Se si rispetta questo principio, un file di intestazione può essere incorporato in diversi file, garantendo un uso uniforme di quanto dichiarato al suo interno, senza duplicazioni inutili nel risultato della

compilazione, anche se ciò che contiene tale file viene usato solo parzialmente o non viene usato affatto.

Un file di intestazione deve contenere ciò che serve alla soluzione di un certo tipo di problematica, ben delimitata. In particolare, dovrebbe contenere tutti i prototipi delle funzioni che servono, o possono servire, per quel tale problema.

66.10.2 Funzioni pubbliche

Le funzioni che devono poter essere usate in varie parti del programma è bene siano pubbliche (come avviene in modo predefinito) e che siano descritte come prototipo in un file di intestazione appropriato. Per quanto possibile, le funzioni potrebbero essere scritte in file indipendenti, ovvero: un file distinto per ogni funzione.

Dal momento che le funzioni potrebbero avere bisogno di usare macro-variabili o macroistruzioni definite nel file di intestazione che ne dichiara i prototipi, nei file di queste funzioni dovrebbe apparire l'inclusione del file di intestazione rispettivo.

66.10.3 Funzioni e variabili private

Le funzioni dichiarate con la parola chiave '**`static`**' sono visibili solo all'interno del file-oggetto in cui vanno a finire. Queste funzioni statiche sono utili in quanto vengono chiamate da una sola o da poche funzioni; in tal caso, questo gruppo di funzioni è costretto a convivere nello stesso file.

Lo stesso problema riguarda le variabili che devono essere utilizzate da più funzioni, ma che non devono essere visibili alle altre, perché anche in questo caso si rende necessario il mettere tale insieme nello stesso file.

66.10.4 Esempio di «stdlib.h»

Per comprendere il senso di quanto appena descritto in modo così sintetico, è utile osservare l'organizzazione della libreria C standard, anche se poi nella realtà i contenuti dei file che la compongono non sono sempre facili da interpretare. A ogni modo, qui viene proposto il caso di quella parte della libreria C che fa capo al file di intestazione '`stdlib.h`'.

Per cominciare, già dal nome del file scelto come esempio, va osservato che un file di intestazione realizzato in modo conforme alla filosofia del linguaggio rappresenta una «libreria» di qualcosa, anche se, per le funzioni, contiene solo i prototipi. Ecco, in breve, come potrebbe essere fatto il file '`stdlib.h`', omettendo alcune porzioni ridondanti per i fini della spiegazione:

```
#ifndef _STDLIB_H
#define _STDLIB_H       1
#define NULL 0
typedef unsigned long int size_t;
typedef unsigned int wchar_t;
#include <limits.h>
typedef struct {int quot; int rem;} div_t;
…
#define RAND_MAX        INT_MAX
…
int   atoi    (const char *nptr);
…
int   rand    (void);
void  srand   (unsigned int seed);
void *malloc  (size_t size);
void *realloc (void *ptr, size_t size);
void  free    (void *ptr);
#define calloc(nmemb, size) (malloc ((nmemb) * (size)))
…
#endif
```

Si può osservare che l'interpretazione del contenuto del file è subordinata al fatto che la macro-variabile ***_STDLIB_H*** non sia già stata dichiarata, mentre altrimenti viene dichiarata. In pratica, con questo meccanismo, se per qualunque ragione un file si trova a incorporare più volte il file di intestazione, il compilatore considera quel contenuto solo la prima volta.

Nell'esempio si vedono dichiarazioni di macro-variabili, di macroistruzioni (***calloc()*** è, in questo caso, una macroistruzione), di tipi di dati derivati. Secondo il buon senso, tutte queste cose devono servire alle funzioni di cui sono presenti i prototipi, ma soprattutto per ciò che riguarda i prototipi. Per esempio, la macro-variabile ***NULL*** viene dichiarata nel file di intestazione perché è il valore che potrebbe essere restituito da funzioni come ***malloc()*** e deve essere uniformato; il tipo derivato '**`size_t`**' viene dichiarato perché viene usato dalla funzione ***malloc()*** e da altre; il file '`limits.h`' viene incorporato perché definisce il valore della macro-variabile ***INT_MAX*** che in questo caso viene usato per definire ***RAND_MAX***, la quale deve essere uniformata per l'uso con la funzione ***rand()***.

La funzione ***atoi()*** è utile per dimostrare in che modo mettere ogni funzione nel proprio file indipendente. Per esempio, quello che segue potrebbe essere il file '`atoi.c`':

```
#include <stdlib.h>:
#include <ctype.h>:
int
atoi (const char *nptr)
{
  int i;
  int sign = +1;
  int n;

  for (i = 0; isspace (nptr[i]); i++)
    {
      ;       // Si limita a saltare gli spazi iniziali.
    }

  if (nptr[i] == '+')
    {
      sign = +1;
      i++;
    }
  else if (nptr[i] == '-')
    {
      sign = -1;
      i++;
    }

  for (n = 0; isdigit (nptr[i]); i++)
    {
      n = (n * 10) + (nptr[i] - '0'); // Accumula il valore.
    }

  return sign * n;
}
```

Come si vede, questa versione di ***atoi()*** si avvale delle funzioni ***isspace()*** e ***isdigit()***, dichiarate nel file '`ctype.h`' che viene aggiunto di conseguenza all'elenco delle inclusioni. Questa inclusione non è stata fatta nel file di intestazione '`stdlib.h`', perché l'uso delle funzioni ***isspace()*** e ***isdigit()*** è dovuto soltanto a una scelta realizzativa di ***atoi()*** e non perché la libreria '`stdlib.h`' dipenda necessariamente da '`ctype.h`'.

Per realizzare le funzioni ***rand()*** e ***srand()*** deve essere condivisa una variabile, la quale può essere nascosta prudentemente al resto del programma. Pertanto serve un file unico che incorpori entrambe le funzioni:

```
#include <stdlib.h>
static unsigned int _srand = 1; // Il rango di «_srand»
                                // deve essere maggiore o
                                // uguale a quello di
                                // «RAND_MAX» e di
                                // «unsigned int».
int
rand (void)
{
  _srand = _srand * 1234567 + 12345;
  return _srand % ((unsigned int) RAND_MAX + 1);
}

void
srand (unsigned int seed)
{
  _srand = seed;
}
```

66.10.5 Parametri delle macroistruzioni

Quando si dichiara una macroistruzione, si usano delle macro-variabili interne che rappresentano i parametri per la «chiamata» di questa specie di funzione. Dal momento che il codice che costituisce la macroistruzione può avvalersi di altre macro-variabili già dichiarate e dato che di norma queste hanno nomi che utilizzano lettere maiuscole, è bene che quelle interne siano scritte con sole lettere minuscole. In pratica, conviene fare come nella macroistruzione già apparsa nella sezione precedente:

```
#define calloc(nmemb, size) (malloc ((nmemb) * (size)))
```

Al contrario, facendo come nell'esempio successivo, il rischio che sia già stata dichiarata la macro-variabile ***SIZE*** oppure ***NMEMB*** è più alto:

```
#define calloc(NMEMB, SIZE) (malloc ((NMEMB) * (SIZE)))
```

66.10.6 Compilazione

I vari file con estensione '`.c`' possono essere compilati separatamente, per ottenere altrettanti file-oggetto da collegare successivamente (i file '`.h`' devono essere incorporati da file '`.c`', pertanto non vanno compilati da soli). Per esempio, per un certo gruppo di file collocato in una certa directory, si potrebbe usare un file-make simile a quello seguente:

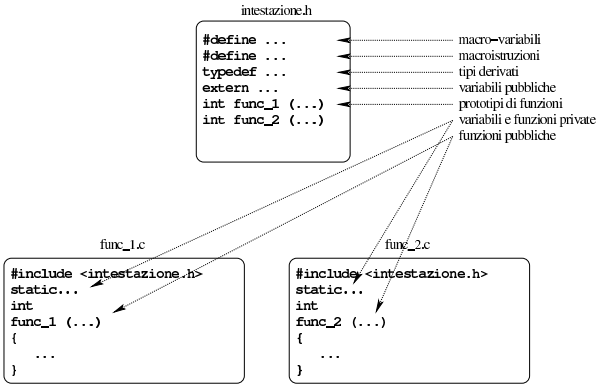
```
sorgenti = uno due tre
#
all: $(sorgenti)
#
clean:
        @rm *.o 2> /dev/null
#
$(sorgenti):
        @echo $@.c
        @gcc -Wall -Werror -o $@.o -c $@.c -I../include
```

In pratica, si presume che nella directory in cui si trova il file-make, ci siano i file '`uno.c`', '`due.c`' e '`tre.c`', per i quali si vogliono ottenere altrettanti file-oggetto, con l'estensione appropriata. Si presume anche che i file di intestazione a cui i sorgenti fanno riferimento si trovino nella directory '`../include/`'.

Compilando in questo modo i file che contengono il minimo indispensabile (possibilmente una sola funzione per ciascuno), quando si verificano errori è più semplice concentrare l'attenzione per correggerli.

Quando si dispone dei file-oggetto si può passare al collegamento (*link*), ma anche in questa fase possono emergere dei problemi di tipo diverso: di solito si tratta di una funzione che viene chiamata, della quale esiste solo il prototipo e quindi non si trova in alcun file-oggetto. Naturalmente, il collegamento deve avvenire una volta sola, con tutti i file-oggetto che compongono il programma.

Figura 66.328. Indicazioni generali per la stesura di un insieme di file sorgenti ordinato.



66.11 K&R

Il linguaggio C, nella sua versione originale, nota come «K&R» (Kernigham e Ritchie), aveva delle caratteristiche che, fortunatamente, sono state perdute. Generalmente non è necessario conoscere le particolarità del vecchio linguaggio C, ma può capitare di leggere del vecchio codice, oppure può succedere di dover usare un vecchio compilatore.

Qui si annotano le caratteristiche principali della sintassi K&R, rispetto al linguaggio C, nella sua forma attuale.

66.11.1 Prototipi e chiamate delle funzioni

La differenza più importante tra la sintassi K&R e il linguaggio C attuale, sta nel modo di dichiarare i prototipi delle funzioni. Il prototipo di una funzione K&R non contiene la definizione dei tipi dei parametri (e tantomeno permette di attribuire loro dei nomi). Per esempio, il prototipo

```
int funzione (char a, short b, int c, long d, float e, double f);
```

si riduce, secondo la sintassi K&R, semplicemente nella dichiarazione seguente:

```
int funzione ();
```

Nella sintassi K&R, la mancanza di un prototipo vero e proprio, fa sì che nella chiamata di una funzione occorra essere molto precisi con i tipi degli argomenti; in altri termini, tutto quello che non ha il rango di '**`int`**', va controllato attentamente. Per esempio, supponendo che il rango di '**`long int`**' sia effettivamente maggiore di quello di '**`int`**', la chiamata seguente provoca certamente dei problemi:

```
x = funzione ('a', 123, 456, 789, 12.3, 45.6);
```

Si comprende che l'argomento attuale 789 sia effettivamente di tipo '**`int`**', mentre la funzione si attende un rango maggiore, con risultati non prevedibili. Per quanto riguarda invece i tipi '**`char`**' e '**`short int`**', va osservato che la sintassi K&R prevede la promozione automatica a '**`int`**', inoltre, per il tipo '**`float`**' è prevista la promozione a '**`double`**'.

Come si può intuire, **anche la quantità prevista degli argomenti di una chiamata non è determinabile per il compilatore**, con le conseguenze che si possono immaginare.

66.11.1.1 La macroistruzione «_PROTOTYPE» di Minix

Il codice del sistema operativo Minix è nato in un momento in cui si potevano incontrare sia compilatori C che riconoscevano e richiedevano l'uso di prototipi di funzione con l'indicazione dei parametri, sia di compilatori che potevano accettare solo la sintassi K&R. Per ovviare a questo problema, il codice del sistema Minix adotta l'uso di una macroistruzione, denominata ***_PROTOTYPE***, dichiarata così:

```
#if _ANSI
…
#define _PROTOTYPE(function, params)    function params
…
#else
…
#define _PROTOTYPE(function, params)    function()
…
#endif
```

Successivamente, quando viene il momento di dichiarare un prototipo, questo viene scritto come nell'esempio seguente:

```
_PROTOTYPE( int printf, (const char *_format, ...)              );
_PROTOTYPE( int scanf,  (const char *_format, ...)              );
```

66.11.2 Dichiarazione delle funzioni

Il modello sintattico che descrive la dichiarazione delle funzioni secondo il C di K&R, potrebbe essere espresso come nello schema seguente:

```
tipo nome_funzione (par_1[, par_2]…)
tipo par_1;
[tipo par_2;]
…
{
    …
}
```

L'esempio seguente mostra la dichiarazione di una certa funzione, secondo la sintassi attuale del linguaggio C:

```
int
funzione (int i, int j)
{
    int k;
    k = i + j;
    return (k);
}
```

Così sarebbe invece secondo la sintassi K&R:

```
int
funzione (i, j)
int i;
int j;
{
    int k;
    k = i + j;
    return (k);
}
```

Tra l'altro, ciò può far incorrere in un errore, che il compilatore non segnala:

```
int
funzione (i, j)
int i;
int j;
{
    int i;
    int k;
    k = i + j;
    return (k);
}
```

In questo caso, la variabile ***i*** viene dichiarata anche nel corpo della funzione, oscurando il contenuto del parametro ***i***.

66.11.3 Operatori composti di assegnamento

Nel linguaggio C comune si possono utilizzare degli operatori di assegnamento composti, come nell'esempio seguente in cui si vuole incrementare di due unità la variabile ***i***:

```
i += 2;
```

Nella sintassi K&R, scrivere '**+=**' oppure '**+ =**' non fa differenza, mentre nello standard attuale del linguaggio ciò non è più ammissibile.

Nelle primissime versioni della sintassi K&R, gli operatori composti erano invertiti, pertanto, avrebbe potuto essere scritto:

```
i =+ 2; /* da non fare mai! */
```

Si può osservare che nella sintassi K&R non è possibile usare il segno '**+**' al di fuori della somma, perché non avrebbe alcuna utilità (dal momento che «+***x***» è uguale a ***x***); pertanto, il fatto che si possa anche scrivere '**i = + 2;**', non dovrebbe creare difficoltà. Tuttavia, scrivendo l'istruzione seguente, c'è da domandarsi cosa si intenda veramente:

```
i =- 2; /* da non fare mai! */
```

La variabile ***i*** viene ridotta di due unità, oppure le viene assegnato semplicemente il valore -2?

66.11.4 Tipi numerici

Nella sintassi K&R, le costanti numeriche in ottale possono contenere anche le cifre 8 e 9, senza che il compilatore si allarmi di ciò. Inoltre, le costanti numeriche rappresentano sempre un numero di tipo intero normale ('**int**'), a meno che gli si aggiunga la lettera '**L**', per indicare che si tratta di un tipo '**long int**'.

I tipi numerici disponibili sono minori rispetto allo standard attuale del linguaggio C, mancando il tipo '**long long int**' e il tipo '**long double**'. Inoltre, nella sintassi K&R non è previsto l'uso dello specificatore '**unsigned**'.

66.11.5 Tipo «void *»

Per la sintassi K&R, il tipo '**void ***' è equivalente al tipo '**char ***'. Pertanto, l'incremento di un tale puntatore porta ai byte successivi, mentre così non può avvenire secondo la sintassi attuale.

66.11.6 Direttive del preprocessore

Nella sintassi K&R, le direttive del preprocessore devono avere il cancelletto ('**#**') esattamente nella prima colonna; inoltre non sono ammissibili direttive nulle, in cui il cancelletto sta da solo.

Le direttive '**#elif**', '**#error**' e '**#pragma**' non sono disponibili. Gli operandi '**defined**', '**#**' e '**##**' non sono disponibili.

66.11.7 Altre osservazioni su K&R

- Sulla funzione ***main()*** non si specifica cosa debba o possa restituire.
- Non sono disponibili le sequenze triplici, o *trigraph*, e di conseguenza non viene riconosciuta la sequenza '**\?**' nelle costanti carattere o nelle stringhe.
- La porzione significativa dei nomi degli identificatori (nomi di funzioni, di variabili, ecc.) è di soli otto caratteri.
- Le sequenze '**\a**', '**\?**' e '**\x**', nelle costanti carattere e nelle stringhe, non sono riconosciute; al contrario, sono ammissibili le sequenze '**\8**' e '**\9**', che invece non dovrebbero, trattandosi di riferimenti a valori in ottale.
- Le costanti stringa, potrebbero risultare modificabili, mentre la sintassi attuale del linguaggio non lo deve consentire.
- L'operatore '**&**', usato per ottenere il puntatore a una variabile, non può essere usato con i nomi degli array.
- L'inizializzazione di una variabile, mentre viene dichiarata, può essere fatta omettendo il segno '**=**'. Per esempio, al posto di scrivere '**int x = 3 + 4;**', si può scrivere '**int x 3 + 4;**'.
- Nella struttura di controllo '**switch**', l'espressione che viene valutata per la scelta dell'azione da compiere deve essere di tipo '**int**'.
- Le enumerazioni non sono disponibili.

66.11.8 Unproto

«

Se per qualche ragione si deve usare un compilatore C che è rimasto a standard precedenti al 1987, viene in aiuto il programma Unproto,[24] di Wietse Venema, che va inserito tra il preprocessore C e il compilatore vero e proprio.

Unproto è in grado di trasformare il risultato prodotto dal preprocessore in un codice C compatibile con la sintassi K&C, sia per la questione legata ai prototipi e la dichiarazione delle funzioni, sia per altri problemi meno appariscenti.

Unproto è anche incluso nella distribuzione Dev86, ovvero gli strumenti di sviluppo per 8088/8086, dove il compilatore BCC di Bruce Evans se ne avvale automaticamente.

66.12 Riferimenti

«

- Brian W. Kernigham, Dennis M. Ritchie, *The C programming language*, Prentice-Hall 1978, prima edizione ISBN 0-13-110163-3; seconda edizione 0-13-110362-8, 0-13-110370-9; edizione italiana, Pearson, ISBN 88-7192-200-X, *http://cm.bell-labs.com/cm/cs/cbook/*).
- Eric Giguere, *The ANSI Standard: A Summary for the C Programmer*, 1987, *http://www.ericgiguere.com/articles/ansi-c-summary.html*
- Open Standards, *C - Approved standards*, *http://www.open-std.org/jtc1/sc22/wg14/www/standards*
- *ISO/IEC 9899:TC2*, *http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf*
- Richard Stallman e altri, *GNU coding standards*, *http://www.gnu.org/prep/standards/*
- Autori vari, *GCC manual*, *http://gcc.gnu.org/onlinedocs/gcc/* , *http://gcc.gnu.org/onlinedocs/gcc.pdf*
- Douglas Walls, *How to use the restrict qualifier in C*, *http://developers.sun.com/solaris/articles/cc_restrict.html*
- *SUPER-UX C Programmer's Guide, DIFFERENCES BETWEEN SUPER-UX ANSI C AND K&R C*, *http://static.cray-cyber.org/Documentation/NEC_SX_R10_1/G1AF02E/CHAP1.HTML#1.3*
- Wietse Zweitze Venema, *Wietse's tools and papers, Unproto*, *ftp://ftp.porcupine.org/pub/security/index.html* , *ftp://ftp.porcupine.org/pub/unix/unproto5.shar.Z*
- Robert de Bath, *Dev86: a cross development C compiler, assembler and linker environment for the production of 8086 executables*, *http://homepage.ntlworld.com/robert.debath/dev86/*

[1] È bene osservare che un'istruzione composta, ovvero un raggruppamento di istruzioni tra parentesi graffe, non è concluso dal punto e virgola finale.

[2] In particolare, i nomi che iniziano con due trattini bassi ('__'), oppure con un trattino basso seguito da una lettera maiuscola ('_***X***') sono riservati.

[3] Tuttavia, le estensioni POSIX prevedono la possibilità di avere tre parametri: '`int main (int argc, char *argv[], char *envp[])`'.

[4] Il linguaggio C, puro e semplice, non comprende alcuna funzione, benché esistano comunque molte funzioni più o meno standardizzate, come nel caso di ***printf()***.

[5] Quando il linguaggio C viene usato secondo lo standard POSIX, ovvero ciò che definisce le caratteristiche di un sistema operativo Unix, il byte deve essere precisamente di 8 bit, senza altre possibilità.

[6] Sono esistiti anche elaboratori in grado di indirizzare il singolo bit in memoria, come il Burroughs B1900, ma rimane il fatto che il linguaggio C si interessi di raggiungere un byte intero alla volta.

[7] Il qualificatore '`signed`' si può usare solo con il tipo '`char`', dal momento che il tipo '`char`' puro e semplice può essere con o senza segno, in base alla realizzazione particolare del linguaggio che dipende dall'architettura dell'elaboratore e dalle convenzioni del sistema operativo.

[8] La distinzione tra valori con segno o senza segno, riguarda solo i numeri interi, perché quelli in virgola mobile sono sempre espressi con segno.

[9] Come si può osservare, la dimensione è restituita dall'operatore '`sizeof`', il quale, nell'esempio, risulta essere preceduto dalla notazione '`(int)`'. Si tratta di un cast, perché il valore restituito dall'operatore è di tipo speciale, precisamente si tratta del tipo '`size_t`'. Il cast è solo precauzionale perché generalmente tutto funziona in modo regolare senza questa indicazione.

[10] Per la precisione, il linguaggio C stabilisce che il «byte» corrisponda all'unità di memorizzazione minima che, però, sia anche in grado di rappresentare tutti i caratteri di un insieme minimo. Pertanto, ciò che restituisce l'operatore ***sizeof()*** è, in realtà, una quantità di byte, solo che non è detto si tratti di byte da 8 bit.

[11] Gli operandi di '`? :`' sono tre.

[12] Lo standard prevede il tipo di dati '`_Bool`' che va inteso come un valore numerico compreso tra zero e uno. Ciò significa che il tipo '`_Bool`' si presta particolarmente a rappresentare valori logici (binari), ma ciò sempre secondo la logica per la quale lo zero corrisponde a *Falso*, mentre qualunque altro valore corrisponde a *Vero*.

[13] Per la precisione, i parametri di una funzione corrispondono alla dichiarazione di variabili di tipo automatico.

[14] Questa descrizione è molto semplificata rispetto al problema del campo di azione delle variabili in C; in particolare, quelle che qui vengono chiamate «variabili globali», non hanno necessariamente un campo di azione esteso a tutto il programma, ma in condizioni normali sono limitate al file in cui sono dichiarate. La questione viene approfondita in modo più adatto a questo linguaggio nella sezione 66.3.

[15] In pratica, ***EXIT_SUCCESS*** equivale a zero, mentre ***EXIT_FAILURE*** equivale a uno.

[16] Lo standard non impone che si tratti di file veri e propri; tuttavia, in un sistema Unix o in qualunque altro sistema operativo analogo, questi sarebbero file da cercare secondo criteri stabiliti, come viene descritto.

[17] In fase di collegamento (*link*) può darsi che il programma che svolge questo compito richieda che i file-oggetto siano indicati secondo una certa sequenza logica, ma questo problema, se esiste, è al di fuori della competenza del linguaggio C.

[18] Si ricorda che, in questo contesto, per «file» si intende il risultato dell'elaborazione da parte del precompilatore, il quale a sua volta potrebbe avere fuso assieme diversi file.

[19] Una variabile potrebbe rappresentare un registro del microprocessore e in tal caso non si potrebbe costruire un puntatore alla stessa. Pertanto, l'argomento sui puntatori parte dal presupposto che le variabili a cui eventualmente si vuole fare riferimento tramite un puntatore siano allocate in memoria.

[20] Per dereferenziare un puntatore si usa generalmente l'asterisco davanti al nome, pertanto il valore a cui punta la variabile ***p*** è accessibile attraverso l'espressione ****p***. Tuttavia esiste un altro modo che viene chiarito a proposito dell'aritmetica dei puntatori, per cui lo stesso valore si raggiunge con l'espressione '`p[0]`'.

[21] In contesti particolari è ammissibile che ***argc*** sia pari a zero, a indicare che non viene fornita alcuna informazione; oppure, se gli argomenti vengono forniti ma il nome del programma è assente, ***argv[0][0]*** deve essere pari a <*NUL*>, ovvero al carattere nullo.

[22] L'indirizzo gestito da un puntatore a una funzione, riguarda potenzialmente uno «spazio di indirizzamento» differente rispetto a quello

usato per le variabili. Per esempio, il puntatore ***p1***, riferito a una certa funzione, potrebbe avere lo stesso contenuto numerico del puntatore ***p2*** riferito a una variabile, ma nella memoria reale, i due puntatori raggiungerebbero posizioni differenti. Ciò serve per comprendere che la gestione dei puntatori alle funzioni non può essere confusa con quella dei dati, perché riguarda domini di indirizzamento diversi.

[23] È improbabile che sia utilizzato un tipo '`void *`', perché non sarebbe possibile scandire la memoria, salvo convertirlo ogni volta in un formato '`char *`'.

[24] **Unproto** software libero