Programmazione in C

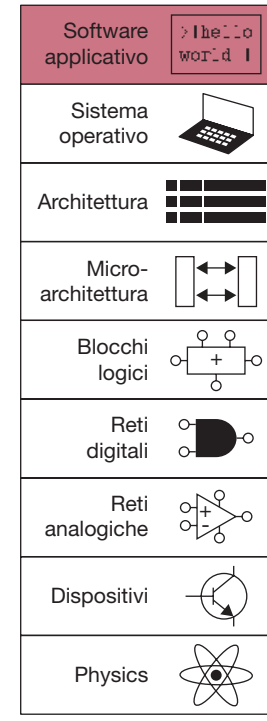
eC

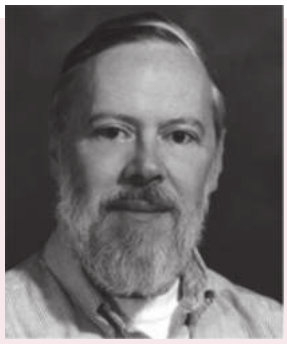
C.1 ■ INTRODUZIONE

Lo scopo principale di questo libro è quello di fornire un quadro di come funzionano i calcolatori a diversi livelli, a partire dai transistori con cui sono realizzati fino al software che sono capaci di eseguire. I primi cinque capitoli del libro risalgono dai livelli di astrazione più bassi, dai transistori alle porte logiche ai circuiti digitali. I capitoli dal sesto all'ottavo fanno un salto al livello dell'architettura e poi scendono alla microarchitettura per collegare concettualmente hardware e software. Questa appendice sulla programmazione in C andrebbe collocata tra il Capitolo 5 e il Capitolo 6, in quanto dedicata al livello di astrazione più alto dell'intero libro: motiva infatti la trattazione dell'architettura e collega il contenuto del libro alle esperienze di programmazione che il lettore potrebbe già avere. Si è quindi deciso di mettere in appendice questo materiale, che può essere usato o meno in base all'esperienza di ciascun lettore.

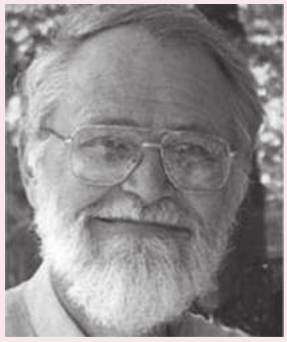
I programmatori usano molti linguaggi diversi per comunicare al calcolatore le operazioni da svolgere. Il calcolatore è capace solo di eseguire istruzioni in **linguaggio macchina**, cioè sequenze di zeri e uni, come discusso nel Capitolo 6. Ma programmare in linguaggio macchina è un'attività lenta e noiosa, e ciò ha spinto i programmatori a utilizzare linguaggi caratterizzati da un più alto livello di astrazione per esprimere le operazioni da svolgere in modo più efficiente. La **Tabella eC.1** elenca alcuni esempi di linguaggi a vari livelli di astrazione.

Uno dei più diffusi linguaggi di programmazione mai sviluppati è denominato C. È stato definito da un gruppo di lavoro che comprendeva Dennis Ritchie e Brian Kernighan ai Laboratori Bell tra il 1969 e il 1973 per consentire di riscrivere il sistema operativo UNIX rispetto alla sua versione originaria in linguaggio assembly. Da molti punti di vista, il C (insieme alla famiglia di linguaggi a lui legati strettamente come il C++, il C# e l'Objective C) è il più diffuso linguaggio esistente. La sua popolarità dipende da una serie di fattori, tra i quali:





Dennis Ritchie, 1941-2011



Brian Kernighan, 1942-

Il C è stato formalmente introdotto nel 1978 dal ben noto libro di Brian Kernighan e Dennis Ritchie *The C Programming Language*. Nel 1989, l'ANSI (*American National Standards Institute*, Istituto Nazionale Americano di Standardizzazione) ha esteso e standardizzato il linguaggio, che è noto con i nomi ANSI C, Standard C e C89. Poco dopo, nel 1990, lo standard è stato adottato dall'ISO (*International Standards Organization*, Organizzazione Internazionale di Standardizzazione) e dall'IEC (*International Electrotechnical Commission*, Commissione Internazionale Elettrotecnica). ISO/IEC hanno aggiornato lo standard nel 1999 in quello che è noto come C99 e che costituisce il riferimento in questo libro.

Tabella eC.1 Linguaggi a livelli di astrazione approssimativamente decrescenti.

Linguaggio	Descrizione
Matlab	Progettato per facilitare l'uso intensivo di funzioni matematiche
Perl	Progettato per *scripting*
Python	Progettato per massimizzare la leggibilità del codice
Java	Progettato per essere eseguito in modo sicuro su ogni calcolatore
C	Progettato per consentire flessibilità e accesso a tutto il sistema, inclusi i driver dei dispositivi
Assembly	Linguaggio macchina reso leggibile dall'uomo
Macchina	Rappresentazione binaria di un programma

- la sua disponibilità per un'enorme varietà di piattaforme hardware, dai supercalcolatori fino ai microcontrollori embedded;
- la sua relativa facilità di uso, con una vastissima base di utenti;
- il suo moderato livello di astrazione, che consente una produttività maggiore del linguaggio assembly ma che dà comunque al programmatore una buona comprensione di come il codice prodotto verrà eseguito;
- il fatto di essere adatto a produrre programmi ad alte prestazioni;
- la sua capacità di interagire direttamente con la struttura hardware.

Questo capitolo è dedicato alla programmazione in C per una serie di ragioni. La più importante è che il C consente al programmatore di accedere direttamente agli indirizzi di memoria, mostrando quindi la stretta correlazione tra hardware e software ampiamente enfatizzata in questo libro. Il C è un linguaggio di programmazione pratico, che tutti gli ingegneri e gli scienziati informatici dovrebbero conoscere. Il suo utilizzo in molti aspetti del progetto e della realizzazione — sviluppi software, programmazione di sistemi embedded, simulazione ecc. — rende la capacità di programmare in C una competenza fondamentale e molto spendibile sul mercato.

I paragrafi seguenti descrivono la sintassi generale di un programma C, discutendo ogni parte del programma comprese le dichiarazioni di intestazioni, di funzioni e di variabili, i tipi di dati e le più comuni funzioni fornite dalle librerie. Il Capitolo 9 (disponibile come supplemento web, come indicato nella Prefazione) descrive un'applicazione pratica utilizzando il C per programmare un calcolatore Raspberry Pi basato su ARM.

RIASSUNTO

- **Programmazione di alto livello:** la programmazione di alto livello è utile in molte fasi della progettazione, dalla scrittura di software di analisi o di simulazione fino alla programmazione di microcontrollori che interagiscono con l'hardware.
- **Accessi di basso livello:** il codice C è molto potente perché oltre ai costrutti di alto livello fornisce la possibilità di effettuare accessi di basso livello a hardware e memoria.

C.2 ■ BENVENUTI AL LINGUAGGIO C

Un programma C è un file di testo che descrive le operazioni che il calcolatore deve eseguire. Il file di testo deve essere **compilato**, cioè tradotto in formato comprensibile alla macchina, ed **eseguito** su un calcolatore. L'Esempio di codice C eC.1 è un semplice (ma usatissimo) primo programma C che scrive

la frase "Hello world!" sulla ***console***, cioè sullo schermo del calcolatore. I programmi C sono generalmente contenuti in uno o più file con estensione ".c". È buona norma di programmazione usare nomi di file che indichino il contenuto del programma: in questo caso, il file potrebbe per esempio chiamarsi `hello.c`.

ESEMPIO DI CODICE C eC.1 UN SEMPLICE PROGRAMMA C

```
// Scrive "Hello world!" sulla console
#include <stdio.h>
int main(void){
  printf("Hello world!\n");
}
```

Visualizzazione sulla console

```
Hello world!
```

Il C è il linguaggio usato per scrivere diffusissimi sistemi come Linux, Windows e iOS. Il C è un linguaggio estremamente potente grazie al suo accesso diretto all'hardware. In confronto ad altri linguaggi come Perl o Matlab, il C non ha molti supporti nativi per operazioni specialistiche come manipolazione di file, *pattern matching*, gestione di matrici, interfacce utente grafiche. Inoltre non ha grossi supporti per prevenire i più comuni errori dei programmatori, come scrivere dati dopo la fine di un array. La sua potenza unita alla mancanza di protezioni ha aiutato gli hacker che possono sfruttare programmi scritti male per accedere ai calcolatori.

C.2.1 Struttura di un programma C

In generale, un programma C è organizzato in una o più **funzioni**. Ogni programma deve contenere almeno la funzione `main`, che costituisce il punto da cui parte l'esecuzione del programma stesso. Molti programmi usano anche altre funzioni, definite nel codice C e/o in una libreria. Le parti del programma `hello.c` sono l'*header* (intestazione), la funzione `main` e il corpo (*body*) della funzione.

Header: `#include <stdio.h>`
L'header include le **librerie di funzioni** necessarie al programma. In questo caso, il programma usa la funzione `printf` che fa parte della libreria standard di I/O: `stdio.h`. *Vedi* il paragrafo C.9 per ulteriori informazioni sulle librerie C.

Funzione main: `int main(void)`
Tutti i programmi C devono avere una e una sola funzione `main`. L'esecuzione del programma avviene facendo eseguire al calcolatore il codice contenuto nella funzione `main`, denominato **corpo** (*body*) di `main`. La sintassi per le funzioni è descritta nel paragrafo C.6. Il corpo di una funzione è costituito da una sequenza di **istruzioni** (*statement*). Ogni istruzione termina con un punto e virgola. La notazione `int` indica che la funzione `main` **restituisce** (*return*) come risultato un numero intero, che indica se il programma è stato eseguito correttamente o no.

Corpo: `printf("Hello world!\n");`
Il corpo della funzione `main` contiene una sola istruzione, la chiamata alla funzione `printf`, che visualizza la frase "Hello world!", seguita da un ritorno a capo specificato con la sequenza speciale `"\n"`. Ulteriori dettagli sulle funzioni di I/O sono presenti nel paragrafo C.9.1.

Tutti i programmi seguono il formato generale del semplice programma `hello.c`. Naturalmente, programmi molto complessi possono contenere milioni di linee di codice e occupare centinaia di file.

C.2.2 Esecuzione di un programma C

I programmi C possono essere eseguiti su molte macchine diverse. La **portabilità** è un altro vantaggio del linguaggio C. Il programma deve per prima cosa essere compilato per la macchina desiderata usando il **compilatore C**. Esistono versioni un po' diverse di compilatori C, come per esempio *cc* (*C compiler*) e *gcc* (*GNU C compiler*). Qui si mostra come compilare ed eseguire

Questo capitolo fornisce una comprensione degli aspetti di base della programmazione in C: esistono interi testi dedicati alla descrizione approfondita del linguaggio. Uno dei migliori è il classico testo *The C Programming Language* (titolo della traduzione in italiano *Il linguaggio C*) di Brian Kernighan e Dennis Ritchie, gli sviluppatori del linguaggio C, che fornisce una descrizione concisa di tutti gli aspetti di tale linguaggio. Un altro valido testo è *A Book on C* (titolo della traduzione italiana *C: didattica e programmazione*) di Al Kelley e Ira Pohl.

un programma utilizzando gcc, che può essere scaricato liberamente. È direttamente eseguibile su macchine Linux ed è accessibile nell'ambiente Cygwin per macchine Windows. È anche disponibile per molti sistemi embedded come il Raspberry Pi basato su ARM. Il processo generale di creazione, compilazione ed esecuzione di un file C è il medesimo per ogni programma C.

1. Creare il file di testo, per esempio `hello.c`.
2. In una finestra di tipo Terminale, spostarsi nella cartella che contiene il file `hello.c` e digitare il comando `gcc hello.c` al prompt dei comandi.
3. Il compilatore crea un file eseguibile, che per default si chiama `a.out` (o `a.exe` su macchine Windows).
4. Al prompt dei comandi, digitare `./a.out` (o `./a.exe` su macchine Windows) e premere invio.
5. "Hello world!" apparirà sullo schermo.

RIASSUNTO

- `nomefile.c`: i programmi C hanno tipicamente estensione .c.
- `main`: ogni programma C deve avere una e una sola funzione `main`.
- `#include`: la maggior parte dei programmi C usa funzioni presenti in librerie precostituite. Per usare queste funzioni si deve inserire la direttiva `#include <libreria.h>` all'inizio del file C.
- `gcc nomefile.c`: i file C sono tradotti in codice eseguibile usando compilatori come GNU compiler (gcc) o C compiler (cc).
- **Esecuzione:** dopo la compilazione, i programmi C sono eseguiti digitando il comando `./a.out` (o `./a.exe`) al prompt dei comandi.

C.3 ■ COMPILAZIONE

Un compilatore è un software che legge un programma scritto in un linguaggio di alto livello e lo traduce in un file scritto in linguaggio macchina chiamato eseguibile. Esistono libri interi dedicati ai compilatori, ma se ne fornisce qui una descrizione sintetica. L'attività del compilatore consiste in (1) pre-elaborare il file includendo i riferimenti alle librerie ed espandendo le definizioni di macro, (2) ignorare tutte le informazioni non necessarie come i commenti, (3) tradurre il codice di alto livello in semplici istruzioni native del processore rappresentate in binario, il cosiddetto linguaggio macchina, e (4) compilare tutte le istruzioni in un singolo eseguibile binario che può essere letto ed eseguito dal calcolatore. Il linguaggio macchina è specifico di un certo processore, quindi il programma deve essere compilato per il sistema sul quale dovrà essere eseguito. Il linguaggio macchina del processore ARM è descritto in dettaglio nel Capitolo 6.

C.3.1 Commenti

I programmatori usano i commenti per descrivere il codice a un più alto livello e per chiarire la funzione del codice stesso. Chiunque abbia avuto a che fare con programmi non commentati sa bene l'importanza dei commenti. I programmi C usano due tipi di commenti: i commenti a riga singola iniziano con la coppia di caratteri `//` e terminano alla fine della linea; i commenti a linea multipla iniziano con la coppia di caratteri `/*` e terminano con la coppia di caratteri `*/`. Sebbene fondamentali per l'organizzazione e la chiarezza del programma, i commenti vengono ignorati dal compilatore.

```
// Questo è un esempio di commento a riga singola.
/* Questo è un esempio di commento
   a riga multipla. */
```

Un comento all'inizio di ogni file C è utile per ricordare l'autore del file, le date di creazione e di modifica e lo scopo del programma. Il seguente commento potrebbe per esempio essere inserito all'inizio del file `hello.c`.

```
// hello.c
// 1 Gennaio 2015 - Sarah_Harris@hmc.edu, David_Harris@hmc.edu
//
// Questo programma scrive "Hello world!" sullo schermo
```

C.3.2 `#define`

Si può dare un nome alle costanti usando la direttiva `#define` e poi usare tale nome all'interno del programma. Queste costanti definite per tutto il programma vengono a volte impropriamente chiamate macro. Per esempio, si supponga di scrivere un programma che permette al massimo 5 tentativi per indovinare una carta da gioco: si può usare la direttiva `#define` per identificare tale numero:

```
#define MAXTENTATIVI 5
```

Il carattere `#` indica che questa linea sarà elaborata dal **preprocessore**. Prima della compilazione, il preprocessore sostituisce a ogni occorrenza dell'identificatore `MAXTENTATIVI` il valore `5`. Per convenzione, le direttive `#define` sono situate all'inizio del file e gli identificatori sono scritti in lettere maiuscole. Definire le costanti in una posizione e usare gli identificatori nel programma garantisce la consistenza del programma stesso e la facilità di modifica del valore di ogni costante – basta modificare la linea `#define` invece di dover modificare ogni linea di codice in cui è necessario usare il valore in questione.

L'Esempio di codice C eC.2 mostra come usare la direttiva `#define` per convertire pollici in centimetri. Le variabili `poll` e `cm` sono dichiarate essere dei `float` per indicare che sono numeri in virgola mobile in singola precisione. Se si dovesse usare il fattore di conversione (`POLLINCM`) in un grosso programma, averlo dichiarato con la direttiva `#define` evita errori di battitura (per es. scrivere in un punto 2.53 invece di 2.54) e rende semplice fare modifiche (per es. aggiungere cifre dopo la virgola se necessario).

Per default le costanti in C sono decimali, ma possono anche essere esadecimali (con prefisso "0x") e ottali (con prefisso "0"). Le costanti binarie non sono definite nel C99, ma alcuni compilatori le supportano (con prefisso "0b"). Per esempio, i seguenti assegnamenti sono equivalenti:

```
char x = 37;
char x = 0x25;
char x = 045;
```

ESEMPIO DI CODICE C eC.2 USO DI `#define` PER DICHIARARE COSTANTI

```
// Converte pollici in centimetri
#include <stdio.h>
#define POLLINCM 2.54
int main(void) {
  float poll = 5.5; // 5.5 pollici
  float cm;
  cm = poll * POLLINCM;
  printf("%f pollici = %f centimetri\n", poll, cm);
}
```

Visualizzazione sulla console

```
5.500000 pollici = 13.970000 centimetri
```

Le costanti definite a livello globale evitano i cosiddetti "numeri magici", ovvero valori costanti inseriti nel programma senza alcun nome. L'uso dei numeri magici può causare errori difficili da trovare, per esempio quando un numero viene modificato in un punto del programma e non in un altro.

C.3.3 `#include`

La modularità suggerisce di dividere i programmi in file separati e funzioni separate. Le funzioni di uso frequente possono essere riunite insieme per fa-

cilitarne il riutilizzo. Le dichiarazioni di variabili e di costanti e le definizioni di funzioni presenti in un file di tipo **header** possono essere usate da un altro file inserendo la direttiva al preprocessore `#include`. Le **librerie standard** che forniscono funzioni di uso comune sono utilizzabili in questo modo. Per esempio, la linea seguente è necessaria per usare le funzioni definite della libreria di I/O standard, come la funzione `printf`.

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

L'estensione ".h" del file da includere indica che si tratta di un file header. Sebbene le direttive `#include` possano essere inserite ovunque, purché prima di punti in cui le funzioni, le variabili o gli identificatori sono necessari, per convenzione si mettono all'inizio del file C.

Si possono includere anche file header scritti dal programmatore usando i doppi apici (" ")invece delle parentesi angolari (< >). Per esempio, un file header denominato `miefunzioni.h` può essere incluso con la seguente direttiva:

```
#include "miefunzioni.h"
```

Al momento della compilazione, i file indicati fra parentesi angolari vengono ricercati nelle cartelle di sistema, mentre i file indicati fra doppi apici vengono ricercati nella stessa cartella contenente il file C. Se i file header creati dall'utente si trovano in una cartella differente, si deve specificare anche il percorso (*path*) per raggiungere tale cartella a partire dalla cartella corrente.

RIASSUNTO

- **Commenti**: il linguaggio C consente commenti a riga singola (`//`) e a riga multipla (`/* */`).
- `#define NOME valore`: la direttiva `#define` consente di usare un identificatore (`NOME`) ovunque nel programma. Prima della compilazione, tutte le istanze di `NOME` vengono sostituite con `valore`.
- `#include`: la direttiva `#include` permette di utilizzare in un programma funzioni di uso comune. Per le librerie predefinite, inserire all'inizio del file linee del tipo `#include <libreria.h>`. Per includere file header definiti dall'utente, il nome deve essere inserito tra doppi apici specificando il percorso relativo alla cartella corrente, per esempio `#include "mielibrerie/miefunzioni.h"`.

C.4 ■ VARIABILI

Le variabili nei programmi C hanno un tipo, un nome, un valore e una locazione di memoria. Una dichiarazione di variabile definisce il tipo e il nome della variabile. Per esempio, la dichiarazione seguente definisce che la variabile è di tipo char (un tipo che occupa un solo byte) e che il suo none è `x`. Il compilatore decide dove posizionare nella memoria questa variabile da un byte.

```
char x;
```

I nomi delle variabili tengono conto di maiuscolo e minuscolo e possono essere scelti liberamente. Non possono però essere nessuna delle parole riservate del C (come `int`, `while` ecc.), non possono iniziare con una cifra (per es., `int 1x` non è una dichiarazione valida) e non possono includere caratteri speciali come `\`, `*`, `?` o `-`. È consentito il carattere di sottolineatura o *underscore* (_).

Il C vede la memoria come una sequenza di byte, ciascuno associato a un numero univoco che ne indica la posizione ovvero l'**indirizzo**, come mostrato nella **Figura eC.1**. Una variabile occupa uno o più byte di memoria, e l'indirizzo di una variabile da più byte è quello del byte con indirizzo minore. Il tipo della variabile indica come interpretare i byte: un numero intero, in virgola mobile, o altro tipo. Il resto di questo paragrafo descrive i tipi di dati primitivi del linguaggio C, la dichiarazione di variabili globali e locali e l'inizializzazione delle variabili.

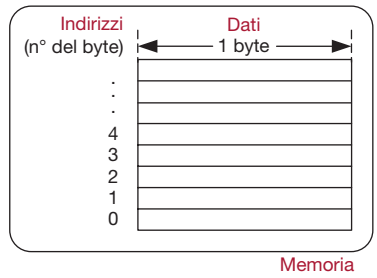


Figura eC.1
La memoria come viene vista dal C.

C.4.1 Tipi di dati primitivi

Il linguaggio C ha un certo numero di tipi di dati primitivi, o predefiniti. A grandi linee, tali tipi si possono classificare in numeri interi, numeri in virgola mobile e caratteri. Una variabile intera è un numero in complemento a due o senza segno compreso in un intervallo finito di valori. Una variabile in virgola mobile usa la rappresentazione IEEE per codificare numeri reali in un certo intervallo e con una certa precisione. Un carattere può essere visto come un valore ASCII o un intero a 8 bit.[1] La **Tabella eC.2** elenca la dimensione e l'intervallo di ogni tipo di dato primitivo. Gli interi possono essere di 16, 32 o 64 bit. Usano il complemento a due a meno che siano qualificati come `unsigned`. La dimensione del tipo `int` dipende dalla macchina e generalmente corrisponde alla dimensione nativa di parola della macchina. Per esempio, su un processore ARM a 32 bit, la dimensione di un `int` o di un `unsigned int` è di 32 bit. I numeri in virgola mobile possono essere di 32 o 64 bit, ovvero singola o doppia precisione. I caratteri sono di 8 bit.

L'Esempio di codice C eC.3 mostra le dichiarazioni di variabili di vari tipi. Come mostrato nella **Figura eC.2**, `x` richiede un byte di dati, `y` due byte e `z` quattro byte. Il compilatore decide dove memorizzare questi byte, ma ogni tipo richiede sempre la stessa quantità di dati. In questo esempio, gli indirizzi di `x`, `y` e `z` sono rispettivamente 1, 2 e 4. I nomi delle variabili tengono conto di maiuscolo e minuscolo, quindi per esempio `x` e `X` sono due variabili diverse (anche se naturalmente non è il caso di usarle entrambe nello stesso programma giusto per cercare di confondersi!).

Tabella eC.2 **Tipi di dati primitivi e loro dimensione.**

Tipo	Dimensione (in bit)	Minimo	Massimo
`char`	8	$-2^{-7} = -128$	$2^7 - 1 = 127$
`unsigned char`	8	0	$2^8 - 1 = 255$
`short`	16	$-2^{15} = -32\,768$	$2^{15} - 1 = 32\,767$
`unsigned short`	16	0	$2^{16} - 1 = 65\,535$
`long`	32	$-2^{31} = -2\,147\,483\,648$	$2^{31} - 1 = 2\,147\,483\,647$
`unsigned long`	32	0	$2^{32} - 1 = 4\,294\,967\,295$
`long long`	64	-2^{63}	$2^{63} - 1$
`unsigned long long`	64	0	$2^{64} - 1$
`int`	dipende dalla macchina		
`unsigned int`	dipende dalla macchina		
`float`	32	$\pm 2^{-126}$	$\pm 2^{127}$
`double`	64	$\pm 2^{-1023}$	$\pm 2^{1022}$

[1] Tecnicamente, lo standard C99 definisce un carattere come "una codifica in bit che sta in un byte" senza specificare che il byte sia di 8 bit. Tuttavia i sistemi attuali assumono come definizione di byte una sequenza di 8 bit.

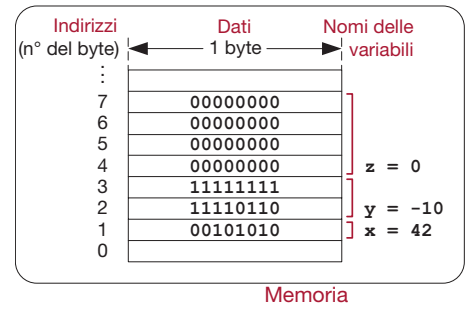


Figura eC.2
Le variabili dell'Esempio di codice C eC.3 memorizzate in memoria.

La natura dipendente dalla macchina del tipo di dato `int` è al tempo stesso un vantaggio e un problema. L'aspetto positivo è il fatto che un `int` corrisponde alla dimensione nativa di parola del processore, quindi può essere prelevato ed elaborato in modo efficiente. L'aspetto negativo è il fatto che i programmi che usano dati `int` possono comportarsi in modo diverso su calcolatori diversi. Ecco cosa succede se un programma di gestione di conti correnti bancari usa variabili di tipo `int` per memorizzare i depositi espressi in centesimi di dollaro (o di euro): se viene compilato su un PC a 64 bit, le variabili hanno tutto lo spazio necessario per trattare anche il più ricco dei correntisti; ma se viene portato su un microcontrollore a 16 bit, il massimo deposito che è in grado di trattare prima di generare traboccamento è pari a 327.67 dollari (o euro): praticamente una banca per poveretti…

ESEMPIO DI CODICE C eC.3 ESEMPI DI TIPI DI DATI

```
// Esempi di alcuni tipi di dati e loro rappresentazione binaria
unsigned char x = 42;     // x = 00101010
short y = -10;            // y = 11111111 11110110
unsigned long z = 0;      // z = 00000000 00000000 00000000 00000000
```

C.4.2 Variabili globali e locali

Le variabili globali e locali differiscono per dove sono dichiarate e per dove sono visibili. Una variabile globale viene dichiarata al di fuori di tutte le funzioni, tipicamente all'inizio del programma, e può essere utilizzata da tutte le funzioni. Le variabili globali vanno usate con molta parsimonia perché violano il principio di modularità e rendono i programmi di grosse dimensioni più difficili da leggere. Se però una variabile deve essere utilizzata da più funzioni va dichiarata globale.

Una variabile locale viene dichiarata all'interno di una funzione e può essere usata solo da quella funzione. Quindi due funzioni possono avere entrambe una variabile con lo stesso nome senza interferire a vicenda. Le variabili locali sono dichiarate all'inizio della funzione. Cessano di esistere quando la funzione termina e vengono ricreate quando la funzione viene nuovamente chiamata. Non conservano quindi il proprio valore tra una chiamata della funzione e la successiva.

Gli Esempi di codice C eC.4 ed eC.5 confrontano due programmi che usano variabili globali il primo e locali il secondo. Nell'Esempio di codice C eC.4 la variabile `max` può essere utilizzata da qualsiasi funzione. Usare una variabile locale come mostrato nell'Esempio di codice C eC.5 è preferibile perché si mantiene una ben definita interfaccia di modularità.

ESEMPIO DI CODICE C eC.4 VARIABILI GLOBALI

```
// Usa una variabile globale per trovare e visualizzare il massimo di 3 numeri
int max;       // variabile globale che contiene il valore massimo

void trovaMax(int a, int b, int c) {
  max = a;
  if (b > max) {
    if (c > b) max = c;
    else max = b;
  } else if (c > max) max = c;
}

void visualizzaMax(void) {
  printf("Il numero massimo e': %d\n", max);
}

int main(void) {
  trovaMax(4, 3, 7);
  visualizzaMax();
}
```

La **visibilità** (*scope*) di una variabile è il contesto nel quale può essere utilizzata. Per esempio, la visibilità di una variabile locale è la funzione nella quale è stata dichiarata. Tale variabile non è visibile in nessun'altra parte del programma.

ESEMPIO DI CODICE C eC.5 VARIABILI LOCALI

```
// Usa una variabile locale per trovare e visualizzare il massimo di 3 numeri

int restituisciMax(int a, int b, int c) {
  int risultato = a; // variabile locale che contiene il valore massimo
  if (b > risultato) {
    if (c > b) risultato = c;
    else risultato = b;
  } else if (c > risultato) risultato = c;
  return risultato;
}

void visualizzaMax(int m) {
  printf("Il numero massimo e': %d\n", m);
}

int main(void) {
  int max;

  max = restituisciMax(4, 3, 7);
  visualizzaMax(max);
}
```

C.4.3 Inizializzazione delle variabili

Una variabile deve essere **inizializzata** – cioè le si deve assegnare un valore – prima di poter essere letta. Quando una variabile viene dichiarata, le viene riservato il corretto numero di byte di memoria, ma tali byte contengono l'ultimo valore che vi era stato scritto prima, quindi in pratica un valore casuale dal punto di vista della variabile in questione. Tutte le variabili, sia globali sia locali, possono essere inizializzate quando vengono dichiarate oppure nel corpo del programma. L'Esempio di codice C eC.3 mostra variabili inizializzate al momento della loro dichiarazione. L'Esempio di codice C eC.4 mostra invece come le variabili sono inizializzate prima di essere usate ma dopo la loro dichiarazione: la variabile globale `max` viene inizializzata dalla funzione `trovaMax` prima di essere letta dalla funzione `visualizzaMax`. Leggere variabili non inizializzate è un tipico errore di programmazione, spesso difficile da trovare.

RIASSUNTO

- **Variabili:** ogni variabile è definita dal suo tipo di dato, dal suo nome e dalla sua locazione di memoria. La variabile viene definita dall'istruzione `tipodidato nome`.
- **Tipi di dati:** il tipo di dato definisce la dimensione (numero di byte) e la rappresentazione (interpretazione del contenuto dei byte) della variabile. La Tabella eC.2 elenca i tipi di dati predefiniti del C.
- **Memoria:** il C vede la memoria come una sequenza di byte. La memoria memorizza le variabili e associa a ogni variabile un indirizzo (numero del primo byte).
- **Variabili globali:** sono dichiarate al di fuori di tutte le funzioni, e possono essere utilizzate da tutte le funzioni.
- **Variabili locali:** sono dichiarate all'interno di una funzione e possono essere usate solo da quella funzione.
- **Inizializzazione delle variabili:** ogni variabile deve essere inizializzata prima di essere letta. L'inizializzazione può avvenire al momento della dichiarazione oppure in seguito.

C.5 ■ OPERATORI

La forma più tipica di istruzione C è un'**espressione** del tipo:

```
y = a + 3;
```

L'espressione include operatori (come + o *) che agiscono su uno o più operandi come variabili o costanti. Il C supporta gli operatori indicati nella **Tabella eC.3**, raggruppati per categoria ed elencati in ordine decrescente di precedenza. Per esempio, gli operatori di tipo moltiplicativo hanno la precedenza su quelli di tipo additivo. All'interno della stessa categoria, gli operatori vengono valutati nell'ordine in cui compaiono nel programma.

Tabella eC.3 Operatori elencati in ordine decrescente di precedenza.

Categoria	Operatore	Descrizione	Esempio
Unari	++	post-incremento	a++; // a = a+1
	--	post-decremento	x--; // x = x-1

	&	indirizzo di memoria di una variabile	x = &y; //x = indirizzo
			//di memoria di y

	~	NOT bit a bit	z = ~a;
	!	NOT booleano	!x
	-	negazione	y = -a;
	++	pre-incremento	++a; // a = a+1
	--	pre-decremento	--x; // x = x-1

	(tipo)	cast di una variabile a (tipo)	x = int(c); //cast di c a int
			//e assegnamento a x

	(dimensione di)	dimensione di una variabile o di un tipo in byte	long int y; x = sizeof(y); // x = 4
Moltiplicativi	*	moltiplicazione	y = x * 12;
	/	divisione	z = 9 / 3; // z = 3
	%	resto di divisione	z = 5 % 2; // z = 1
Additivi	+	addizione	y = a + 2;
	-	sottrazione	y = a - 2;
Traslazioni bit a bit	<<	traslazione a sinistra	z = 5 << 2; // z = 0b00010100
	>>	traslazione a destra	x = 9 >> 3; // x = 0b00000001
Relazionali	= =	uguale	y == 2
	!=	diverso	x != 7
	<	minore	y < 12
	>	maggiore	val > max
	<=	minore o uguale	z <= 2
	>=	maggiore o uguale	y >= 10
Bit a bit	&	AND bit a bit	y = a & 15;
	^	XOR bit a bit	y = 2 ^ 3;
	|	OR bit a bit	y = a | b;
Logici	&&	AND booleano	x && y
	||	OR booleano	x || y

Ternario	? :	operatore ternario	y = x ? a : b //se x vale TRUE y=a
			// altrimenti y=b

(*continua*)

Tabella eC.3 Operatori elencati in ordine decrescente di precedenza. (*continua*)

Categoria	Operatore	Descrizione	Esempio
Assegnamenti	=	assegnamento	`x = 22;`
	+=	somma e assegnamento	`y += 3;  // y = y + 3`
	-=	sottrazione e assegnamento	`z -= 10;  // z = z - 10`
	*=	moltiplicazione e assegnamento	`x *= 4;  // x = x * 4`
	/=	divisione e assegnamento	`y /= 10;  // y = y / 10`
	%=	resto e assegnamento	`x %= 4;  // x = x % 4`
	>>=	traslazione a destra bit a bit e assegnamento	`x >>= 5;  // x = x>>5`
	<<=	traslazione a sinistra bit a bit e assegnamento	`x <<= 2;  // x = x<<2`
	&=	AND bit a bit e assegnamento	`y &= 15;  // y = y & 15`
	|=	OR bit a bit e assegnamento	`x |= y;  // x = x | y`
	^=	XOR bit a bit e assegnamento	`x ^= y;  // x = x ^ y`

Gli operatori unari, detti anche monadici, hanno un solo operando, l'operatore ternario ne ha tre e tutti gli altri ne hanno due. L'operatore ternario seleziona il secondo oppure il terzo operando a seconda del valore TRUE (vero, in C un valore diverso da zero) oppure FALSE (falso, in C il valore zero) del primo operando. L'Esempio di codice C eC.6 mostra come calcolare `y = max(a,b)` usando l'operatore ternario oppure con la versione equivalente ma più verbosa che fa uso dell'istruzione `if/else`.

La verità, tutta la verità, nient'altro che la verità.

Il C considera una variabile TRUE (vera) se il suo valore è diverso da zero e FALSE (falsa) se è uguale a zero. Gli operatori logici e l'operatore ternario, come pure le istruzioni di controllo del flusso come `if e while`, dipendono dal valore vero o falso di una variabile. Gli operatori relazionali e quelli logici producono un risultato che vale 1 in caso di TRUE e 0 in caso di FALSE.

ESEMPIO DI CODICE C eC.6 (a) OPERATORE TERNARIO (b) ISTRUZIONE `if/else` EQUIVALENTE

```
(a) y = (a > b) ? a : b; // parentesi non necessarie, usate solo per chiarezza

(b) if (a > b) y = a;
    else y = b;
```

L'assegnamento semplice usa l'operatore =. Il C consente anche l'uso di assegnamenti compositi, cioè assegnamenti eseguiti dopo una semplice operazione come addizione (+=) o moltiplicazione (*=). Negli assegnamenti compositi, la variabile nella parte sinistra viene usata sia come operando sia come destinazione del risultato. L'Esempio di codice C eC.7 mostra questi e altri esempi di operazioni in C. I valori binari nei commenti sono indicati con il prefisso "`0b`".

ESEMPIO DI CODICE C eC.7 ESEMPI DI USO DEGLI OPERATORI

Espressione		Risultato	Note
`44 / 14`		`3`	La divisione intera effettua troncamento
`44 % 14`		`2`	Resto di 44 diviso 14
`0x2C && 0xE`	`//0b101100 && 0b1110`	`1`	AND logico
`0x2C || 0xE`	`//0b101100 || 0b1110`	`1`	OR logico
`0x2C & 0xE`	`//0b101100 & 0b1110`	`0xC (0b001100)`	AND bit a bit
`0x2C | 0xE`	`//0b101100 | 0b1110`	`0x2E (0b101110)`	OR bit a bit
`0x2C ^ 0xE`	`//0b101100 ^ 0b1110`	`0x22 (0b100010)`	XOR bit a bit
`0xE << 2`	`//0b1110 << 2`	`0x38 (0b111000)`	Traslazione a sinistra di 2
`0x2C >> 3`	`//0b101100 >> 3`	`0x5 (0b101)`	Traslazione a destra di 3

(*continua*)

ESEMPIO DI CODICE C eC.7 ESEMPI DI USO DEGLI OPERATORI (*continua*)

Espressione	Risultato	Note
`x = 14;` `x += 2;`	`x=16`	

`y = 0x2C;  // y = 0b101100`
`y &= 0xF;  // y &= 0b1111`
	`y=0xC (0b001100)`	

`x = 14; y = 44;` `y = y + x++;`	`x=15, y=58`	Incrementa `x` dopo averla usata
`x = 14; y = 44;` `y = y + ++x;`	`x=15, y=59`	Incrementa `x` prima di usarla

C.6 ■ CHIAMATE DI FUNZIONE

La modularità è un aspetto chiave per una buona programmazione. Un grosso programma deve essere diviso in parti più piccole chiamate funzioni che, come i moduli hardware, hanno ingressi, uscite e comportamento ben definiti. L'Esempio di codice C eC.8 mostra la funzione `somma3`. La dichiarazione di funzione inizia con tipo restituito: `int`, seguito dal nome: `somma3`, e dai valori di ingresso (detti "argomenti" o "parametri") elencati tra partentesi: (`int a, int b, int c`). Le parentesi graffe racchiudono il corpo della funzione, che può contenere zero o più istruzioni. L'istruzione `return` indica il valore che la funzione deve restituire al chiamante, e che può quindi essere considerato l'uscita della funzione. Una funzione può restituire un solo valore.

ESEMPIO DI CODICE C eC.8 FUNZIONE

```
// Restituisce la somma delle tre variabili di ingresso
int somma3(int a, int b, int c) {
  int risultato = a + b + c;
  return risultato;
}
```

Dopo la seguente chiamata alla funzione `somma3`, la variabile `y` contiene il valore 42.

```
int y = somma3 (10, 15, 17);
```

Sebbene una funzione possa avere valori di ingresso e di uscita, tali valori non sono necessari. L'Esempio di codice C eC.9 mostra una funzione senza ingressi né uscite. La parola chiave `void` prima del nome della funzione indica che non viene restituito alcun valore. La stessa parola `void` tra parentesi indica che la funzione non ha argomenti di ingresso.

Se non si indica nulla tra le parentesi, questo significa nessun argomento. Si sarebbe quindi potuto scrivere anche:

```
void mostraPrompt()
```

ESEMPIO DI CODICE C eC.9 FUNZIONE `mostraPrompt` SENZA INGRESSI NÉ USCITE

```
// Visualizza un messaggio di prompt sulla console
void mostraPrompt(void)
{
  printf ("Inserire un numero da 1 a 3: ");
}
```

Una funzione deve essere dichiarata nel codice prima di essere usata. Per far questo si può scrivere la funzione chiamata nella parte iniziale del file. Per questo motivo la funzione `main` viene di solito posizionata alla fine del file, dopo tutte le funzioni da lei chiamate. Oppure si può inserire nel programma un **prototipo** della funzione prima della posizione in cui la funzione viene definita. Il prototipo della funzione è la prima linea della funzione vera e propria, in cui sono dichiarati il tipo da restituire, il nome della funzione e gli

ingressi alla funzione. Per esempio, i prototipi delle funzioni presenti negli Esempi di codice C eC.8 ed eC.9 sono:

```
int somma3(int a, int b, int c);
void mostraPrompt(void);
```

L'Esempio di codice C eC.10 mostra come usare i prototipi di funzione. Anche se le funzioni sono definite dopo `main`, la presenza dei prototipi all'inizio del file ne consente l'utilizzo da parte di `main`.

Se si ordinano opportunamente le funzioni, i prototipi possono non servire. Diventano però inevitabili in alcuni casi, come quando la funzione `f1` chiama la funzione `f2`, che a sua volta chiama `f1`. È quindi un buono stile di programmazione inserire i prototipi di tutte le funzioni di un programma all'inizio del file in C o in un file header.

ESEMPIO DI CODICE C eC.10 PROTOTIPI DI FUNZIONE

```
#include <stdio.h>

// prototipi di funzione
int somma3(int a, int b, int c);
void mostraPrompt(void);

int main(void)
{
  int y = somma3(10, 15, 20);
  printf("risultato di somma3: %d\n", y);
  mostraPrompt();
}

int somma3(int a, int b, int c) {
  int risultato = a+b+c;
  return risultato;
}

void printPrompt(void) {
  printf ("Inserire un numero da 1 a 3: ");
}
```

Visualizzazione sulla console

```
risultato di somma3: 45
Inserire un numero da 1 a 3:
```

Come nel caso delle variabili, anche i nomi di funzione tengono conto di maiuscolo e minuscolo, non possono essere nessuna delle parole riservate del C, non possono includere caratteri (a eccezione della sottolineatura _) e non possono iniziare con una cifra. Di solito i nomi di funzioni contengono un verbo per indicare cosa fanno tali funzioni.
È importante adottare uno stile consistente nell'uso di lettere maiuscole e minuscole per i nomi di funzioni e variabili per non dover passare il tempo a verificare se si è scritto correttamente. Due stili tipici sono "a cammello", nel quale si scrive maiuscola l'iniziale di ogni parola tranne la prima (come per esempio `mostraPrompt`), con una forma che ricorda appunto la schiena a gobbe del cammello, oppure con inserimento di caratteri di sottolineatura fra le parole (come per esempio `mostra_prompt`). Gli autori sostengono che il secondo stile affatica oltremodo il dito che deve premere il tasto della sottolineatura, quindi preferiscono lo stile a cammello, ma la cosa importante è adeguarsi allo stile in uso nella propria organizzazione.

La funzione `main` viene sempre dichiarata restituire un `int`, che comunica al sistema operativo il motivo della sua terminazione. Il valore zero indica la condizione normale, valori diversi da zero indicano condizioni di errore. Se `main` arriva alla fine senza incontrare nessuna istruzione `return`, restituisce il valore zero. Si noti che molti sistemi operativi non comunicano automaticamente all'utente il valore restituito dal programma.

C.7 ■ ISTRUZIONI DI CONTROLLO DEL FLUSSO DI ESECUZIONE

Il C ha istruzioni di controllo del flusso di due tipi: istruzioni condizionali e cicli. Un'istruzione condizionale esegue un'istruzione solo se una condizione è vera. Un ciclo esegue ripetutamente un'istruzione fintantoché una condizione è vera.

C.7.1 Istruzioni condizionali

Le istruzioni `if`, `if/else` e `switch/case` sono istruzioni condizionali comunemente usate in linguaggi di alto livello come il C.

Istruzione if

Un'istruzione `if` esegue l'istruzione immediatamente seguente quando l'espressione tra parentesi è TRUE (cioè diversa da zero). La forma generale è:

```
if (espressione)
  istruzione
```


Le parentesi graffe, { }, sono usate per raggruppare una o più istruzioni in un'*istruzione composita* ovvero un blocco di istruzioni.

L'Esempio di codice C eC.11 mostra come usare l'istruzione `if` in C. Se la variabile `controllo` è uguale a 1, la variabile `valore` viene forzata a 1. Si può eseguire una sequenza di istruzioni racchiudendo la sequenza fra parentesi graffe, come mostrato nell'Esempio di codice C eC.12.

ESEMPIO DI CODICE C eC.11 ISTRUZIONE `if`

```
int controllo = 0;

if (controllo == 1)
  valore = 1;
```

ESEMPIO DI CODICE C eC.12 ISTRUZIONE `if` CON UNA SEQUENZA DI ISTRUZIONI

```
// Se importo >= $2, visualizza prompt e dispensa merendina

if (importo >= 2) {
  printf ("Selezionare merendina: ");
  dispensaMerendina = 1;
}
```

Istruzione if/else

L'istruzione `if/else` esegue una di due istruzioni a seconda del valore della condizione, come mostrato sotto. Quando `espressione` è TRUE si esegue `istruzione1`, altrimenti si esegue `istruzione2`.

```
if (espressione)
  istruzione1
else
  istruzione2
```

L'Esempio di codice C eC.6(b) mostra l'uso dell'istruzione `if/else`: il codice forza `y = a` se `a` è maggiore di `b`, altrimenti `y = b`.

Istruzione switch/case

L'istruzione switch/case esegue una di varie istruzioni a seconda delle condizioni, come mostrato nella forma generale qui riportata:

```
switch (variabile) {
  case (espressione1): istruzione1 break;
  case (espressione2): istruzione2 break;
  case (espressione3): istruzione3 break;
  default:             istruzione4
}
```

Per esempio, se `variabile` è uguale a `espressione2` l'esecuzione prosegue a partire da `istruzione2` fino a incontrare la parola chiave `break`, che provoca l'uscita dall'istruzione `switch/case`. Se nessuna condizione è verificata, si esegue l'istruzione associata a `default`.

Se si omette la parola chiave `break`, l'esecuzione comincia dove si trova la condizione che assume valore TRUE e prosegue in tutti i casi seguenti: generalmente non è questo che si vuole, ma la dimenticanza di `break` è un errore frequente, soprattutto tra i programmatori inesperti.

L'Esempio di codice C eC.13 mostra un'istruzione `switch/case` che in base al valore della variabile `opzione` aggiorna l'ammontare `importo` di soldi introdotti. L'istruzione `switch/case` equivale a una serie di istruzioni `if/else` annidate, come mostrato dal codice equivalente dell'Esempio di codice C eC.14.

ESEMPIO DI CODICE C eC.13 ISTRUZIONE `switch/case`

```
// Aggiorna importo in base al valore di opzione

switch (opzione) {
  case 1:  importo = 100; break;
  case 2:  importo = 50;  break;
  case 3:  importo = 20;  break;
  case 4:  importo = 10;  break;
  default: printf("Errore: opzione sconosciuta.\n");
}
```

ESEMPIO DI CODICE C eC.14 ISTRUZIONI `if/else` ANNIDATE

```
// Aggiorna importo in base al valore di opzione

if      (opzione == 1)  importo = 100;
else if (opzione == 2)  importo = 50;
else if (opzione == 3)  importo = 20;
else if (opzione == 4)  importo = 10;
else printf("Errore: opzione sconosciuta.\n");
```

C.7.2 Cicli

`while`, `do/while` e `for` sono comuni costrutti ciclici usati in molti linguaggi di alto livello incluso il C. Questi cicli eseguono ripetutamente un'istruzione fintantoché una condizione è soddisfatta.

Ciclo while

Il ciclo while esegue ripetutamente un'istruzione fino a quando la condizione diventa non più verificata, come mostrato nella forma generale qui riportata:

```
while (condizione)
  istruzione
```

Il ciclo `while` nell'Esempio di codice C eC.15 calcola il fattoriale di $9 = 9 \times 8 \times 7 \times \ldots \times 1$. Si noti che `condizione` viene verificata prima di eseguire `istruzione`. In questo esempio l'istruzione è un'istruzione composita, quindi servono le parentesi graffe.

ESEMPIO DI CODICE C eC.15 CICLO `while`

```
// Calcola 9! (il fattoriale di 9)
int i = 1, fatt = 1;

// moltiplica i numeri da 1 a 9
while (i < 10) { // i cicli while verificano la condizione all'inizio
  fatt *=1;
  i++;
}
```

Ciclo do/while

Il ciclo `do/while` è simile al ciclo while, ma `condizione` viene verificata solo dopo aver eseguito una volta `istruzione`. Il formato generale è il seguente, con il punto e virgola dopo la condizione.

```
do
  istruzione
while (condizione);
```

Il ciclo `do/while` dell'Esempio di codice C eC.16 chiede all'utente di indovinare un numero. Il programma verifica la condizione (ovvero se il numero

proposto dall'utente è uguale al numero corretto) solo dopo aver eseguito il corpo del ciclo `do/while` una volta. Questo costrutto è utile nei casi come questo in cui si deve comunque fare qualcosa (nell'esempio chiedere all'utente il tentativo di numero) prima di verificare la condizione.

ESEMPIO DI CODICE C eC.16 CICLO `do/while`

```
// Chiede all'utente di indovinare un numero e verifica che sia quello corretto
#define MAXTENTATIVI 3
#define NUMCORRETTO 7
int tentativo, numTentativi = 0;

do {
  printf("Indovina un numero 0 and 9. Hai ancora %d tentativi.\n",
        (MAXTENTATIVI-numTentativi));
  scanf("%d", &tentativo);   // legge il dato inserito dall'utente
  numTentativi++;
} while ( (numTentativi < MAXTENTATIVI) & (tentativo != NUMCORRETTO) );
// i cicli do/while verificano la condizione dopo la prima iterazione

if (tentativo == NUMCORRETTO)
  printf("Hai indovinato il numero corretto!\n");
```

Ciclo for

Il ciclo `for`, come i cicli `while` e `do/while`, esegue ripetutamente `istruzione` finché `condizione` diventa non più verificata. Ma il ciclo `for` fa uso di una **variabile di ciclo** che tipicamente viene usata per tenere traccia del numero di ripetizioni del ciclo. Il formato generale è il seguente:

```
for (inizializzazione; condizione; operazione di ciclo)
  istruzione
```

Il codice `inizializzazione` viene eseguito una sola volta, prima di iniziare il ciclo `for`. La `condizione` viene verificata all'inizio di ogni iterazione del ciclo: se risulta non TRUE si esce dal ciclo. L'`operazione di ciclo` viene eseguita alla fine di ogni iterazione. L'Esempio di codice C eC.17 mostra il calcolo del fattoriale di 9 con il ciclo `for`.

ESEMPIO DI CODICE C eC.17 CICLO `for`

```
// Calcola 9!
int i = 1;     // variabile di ciclo
int fatt = 1;

for (i=1; i<10); i++)
  fatt *=1;
```

Mentre i cicli `while` e `do/while` degli Esempi di codice C eC.15 ed eC.16 includono codice per incrementare e verificare le variabili di ciclo (rispettivamente `i` e `numTentativi`), il ciclo `for` incorpora queste istruzioni nel suo formato. Un ciclo `for` può essere espresso in modo equivalente, ma meno efficiente, come:

```
inizializzazione;
while (condizione) {
  istruzione
  operazione di ciclo;
}
```

RIASSUNTO

- **Istruzioni di controllo del flusso:** il C rende disponibili istruzioni di controllo del flusso di due tipi: istruzioni condizionali e cicli.
- **Istruzioni condizionali:** le istruzioni condizionali eseguono un'istruzione quando una condizione risulta TRUE. Il C prevede le seguenti istruzioni condizionali: `if`, `if/else` e `switch/case`.
- **Cicli:** i cicli eseguono ripetutamente un'istruzione fino a quando una condizione diventa FALSE. Il C prevede i cicli `while`, `do/while` e `for`.

C.8 ■ ALTRI TIPI DI DATI

Oltre a numeri interi e in virgola mobile di varia dimensione, il C include altri tipi particolari di dati come i puntatori, gli array, le stringhe e le strutture. Tali tipi di dati sono introdotti in questo paragrafo, assieme all'allocazione dinamica della memoria.

C.8.1 Puntatori

Un puntatore è l'indirizzo di una variabile. L'Esempio di codice C eC.18 mostra come usare i puntatori. `salario1` e `salario2` sono variabili di tipo intero, e `punt` è una variabile che può contenere l'indirizzo di un intero. Il compilatore assegna a queste variabili delle posizioni arbitrarie in memoria RAM, in base all'ambiente di esecuzione. Per fare un esempio concreto, si supponga che il programma sia compilato in un sistema a 32 bit con `salario1` memorizzato agli indirizzi 0x780-73, `salario2` agli indirizzi 0x74-77 e `punt` agli indirizzi 0x78-7B. La **Figura eC.3** mostra la memoria e il suo contenuto dopo l'esecuzione del programma.

Nella dichiarazione di variabili, l'asterisco (`*`) prima del nome di una variabile indica che tale variabile è un puntatore a una variabile del tipo dichiarato. Quando si usa una variabile puntatore, l'operatore `*` serve a **dereferenziare** il puntatore, cioè a restituire il valore presente all'indirizzo di memoria contenuto nel puntatore. L'operatore `&` (indirizzo di) serve invece a restituire l'indirizzo della variabile cui viene applicato.

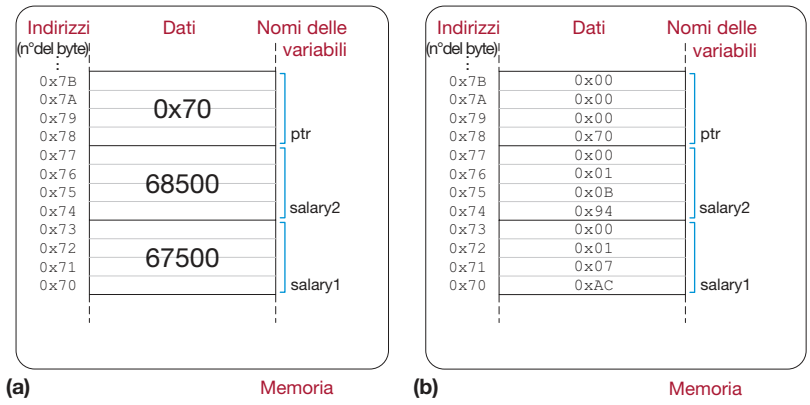


Figura eC.3
Contenuto della memoria dopo l'esecuzione dell'Esempio di codice C eC.18 mostrato (a) per valore e (b) per byte in una memoria little endian.

Dereferenziare un puntatore che punta a una locazione di memoria non esistente o situata al di fuori dello spazio accessibile al programma comporta normalmente l'interruzione forzata del programma, nota come *segmentation fault*.

ESEMPIO DI CODICE C eC.18 PUNTATORI

```
// Esempio di operazioni con i puntatori
int salario1, salario2;  // numeri a 32 bit
int *punt;               // puntatore che contiene indirizzo di variabile int
salario1 = 67500;        // salario1 = $67,500 = 0x000107AC
punt = &salario1;        // punt = 0x0070, l'indirizzo di salario1
salario2 = *punt + 1000; /* dereferenzia punt per avere il contenuto
                            dell'indirizzo 70 = $67,500, quindi somma
                            $1,000 e assegna a salario2 $68,500 */
```

I puntatori sono particolarmente utili quando una funzione deve modificare una variabile invece di restituire semplicemente un valore. Dal momento che una funzione non può modificare direttamente i valori che le sono stati passati in ingresso, può definire come ingresso un puntatore a una variabile. Questo metodo è detto **passaggio dei parametri per riferimento** invece del **passaggio dei parametri per valore** utilizzato sinora. L'Esempio di codice C eC.19 mostra come passare la variabile `x` per riferimento, in modo che la funzione `perquattro` possa modificare tale variabile.

ESEMPIO DI CODICE C eC.19 PASSAGGIO PER RIFERIMENTO DI UNA VARIABILE DI INGRESSO

```
// Moltiplica per 4 il valore referenziato da a
#include <stdio.h>
void perquattro(int *a)
{
*a = *a * 4;
}

int main(void)
{
  int x = 5;
  printf("x prima: %d\n", x);
  perquattro(&x);
  printf("x dopo: %d\n", x);
  return 0;
}
```

Visualizzazione sulla console

```
x prima: 5
x dopo: 20
```

Un puntatore all'indirizzo 0 è detto ***null pointer*** e indica che il puntatore non sta referenziando alcun valore significativo. Viene indicato con la parola chiave `NULL` nei programmi.

C.8.2 Array

Un array è un gruppo di variabili simili memorizzate a indirizzi di memoria consecutivi. Gli elementi sono numerati da 0 a $N - 1$, dove N rappresenta la lunghezza dell'array. L'Esempio di codice C eC.20 dichiara un array di nome `voti` che memorizza i risultati agli esami di tre studenti. Viene riservato uno spazio di memoria in grado di contenere tre variabili di tipo `long`, quindi $3 \times 4 = 12$ byte. Se `voti` inizia all'indirizzo 0x40, l'indirizzo del primo elemento (cioè `voti[0]`) è 0x40, quello del secondo 0x44 e quello del terzo 0x48, come mostrato nella **Figura eC.4**. Il nome della variabile (in questo caso `voti`) è per definizione il puntatore al primo elemento. È responsabilità del programmatore non accedere a elementi oltre la fine dell'array: il C non effettua alcun

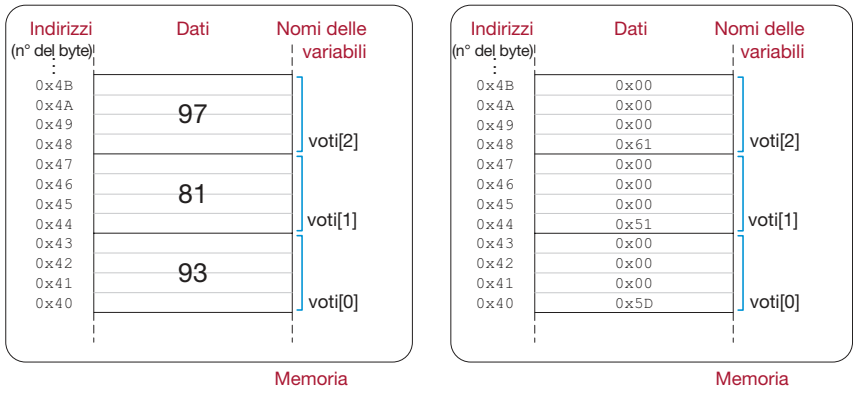


Figura eC.4
L'array voti memorizzato in memoria.

controllo sui limiti degli array, quindi un programma che scriva dopo la fine dell'array viene compilato senza errori ma al momento dell'esecuzione andrà a sovrascrivere indebitamente altre parti di memoria.

ESEMPIO DI CODICE C eC.20 DICHIARAZIONE DI ARRAY

```
long voti[3];  // array di tre elementi da 4 byte ciascuno
```

Gli elementi di un array possono essere inizializzati sia al momento della dichiarazione usando le parentesi graffe `{}`, come mostrato nell'Esempio di codice C eC.21, o singolarmente nel corpo del codice, come mostrato nell'Esempio di codice C eC.22. Il singolo elemento dell'array viene indicato mediante le parentesi quadre `[]`. Il contenuto della memoria nella quale è memorizzato l'array è mostrato nella Figura eC.4. L'inizializzazione dell'array mediante le parentesi graffe può essere fatta una sola volta, all'atto della dichiarazione. I cicli `for` sono normalmente usati per assegnare e leggere dati negli array, come mostrato nell'Esempio di codice C eC.23.

ESEMPIO DI CODICE C eC.21 INIZIALIZZAZIONE DI UN ARRAY ALLA DICHIARAZIONE USANDO { }

```
long voti[3]={27, 21, 30};  // voti[0]=27; voti[1]=21; voti[2]=30;
```

ESEMPIO DI CODICE C eC.22 INIZIALIZZAZIONE DI UN ARRAY USANDO ASSEGNAMENTI

```
long voti[3]

voti[0]=27;
voti[1]=21;
voti[2]=30;
```

ESEMPIO DI CODICE C eC.23 INIZIALIZZAZIONE DI UN ARRAY MEDIANTE CICLO `for`

```
// L'utente inserisce i voti dei 3 studenti nell'array
long voti[3];
int i, voto;

printf("Inserire i voti dei 3 studenti.\n");
for (i=0; i<3; i++) {
  printf("Inserire un voto e premere Invio.\n");
  scanf("%d", &voto);
  voti[i] = voto;
}
printf("Voti: %d %d %d\n", voto[0], voto[1], voto[2]);
```

Quando si dichiara un array, la sua lunghezza deve essere una costante in modo tale che il compilatore possa allocare la necessaria quantità di memoria. Quando però si passa un array come argomento di ingresso a una funzione, non è necessario specificarne la lunghezza in quanto alla funzione serve sapere solo l'indirizzo iniziale dell'array. L'Esempio di codice C eC.24 mostra come passare un array a una funzione. L'argomento di ingresso `arr` è semplicemente l'indirizzo del primo elemento di un array. Spesso si passa alla funzione come argomento anche il numero di elementi dell'array. Nella funzione, un argomento di ingresso di tipo `int[]` indica che si tratta di un array di interi. Alla funzione si possono passare array di qualsiasi tipo.

ESEMPIO DI CODICE C eC.24 PASSAGGIO DI UN ARRAY COME ARGOMENTO DI INGRESSO

```
// Inizializza un array di 5 elementi, calcola la media e la visualizza.
#include <stdio.h>

// Restituisce la media di un array (arr) di lunghezza lun
float calcolaMedia(int arr[], int lun) {
  int i;
  float media, totale = 0;
  for (i=0; i < lun; i++)
    totale += arr[i];
  media = totale / lun;
return media;
}

int main(void) {
  int dati[4] = {78, 14, 99, 27};
  float valormedio;
  valormedio = calcolaMedia (dati, 4);
  printf("il valor medio e': %f.\n", valormedio);
}
```

Visualizzazione sulla console

```
Il valor medio è: 54.500000.
```

Un argomento array è equivalente al puntatore all'inizio dell'array, quindi si sarebbe potuta definire la funzione `calcolaMedia` anche nel modo seguente:

```
float calcolaMedia(int *arr, int lun);
```

Anche se funzionalmente equivalente, si preferisce però la modalità `tipodato[]` per passare un array come argomento a una funzione, perché rende evidente il fatto che l'argomento è appunto un array.

Una funzione può restituire un solo valore, ovvero la variabile di ritorno. Se però riceve come argomento un array può restituire molti valori modificando tale array. L'Esempio di codice C eC.25 ordina un array dal valore più piccolo al più grande e lascia il risultato nell'array stesso. I tre prototipi di funzione sotto riportati sono equivalenti: l'indicazione della lunghezza dell'array nella dichiarazione della funzione (cioè `int valori[100]`) viene ignorata.

```
void ordina(int *valori, int lun);
void ordina(int valori[], int lun);
void ordina(int valori[100], int lun);
```

ESEMPIO DI CODICE C eC.25 PASSAGGIO DI UN ARRAY E DELLA SUA LUNGHEZZA COME ARGOMENTO

```
// Ordina gli elementi dell'array valori di lunghezza lun dal basso all'alto
void ordina(int valori[], int lun)
{
  int i, j, temp;
  for (i=0; i<lun; i++) {
    for (j=i+1; j<lun; j++) {
      if (valori[i] > valori[j]) {
        temp = valori[i];
        valori[i] = valori[j];
        valori[j] = temp;
      }
    }
  }
}
```

Gli array possono avere più dimensioni. L'Esempio di codice C eC.26 usa un array bidimensionale per memorizzare i voti presi da dieci studenti in otto compiti in classe. Si ricordi che l'inizializzazione di un array con le parentesi `{}` può essere fatta solo al momento della sua dichiarazione.

ESEMPIO DI CODICE C eC.26 INIZIALIZZAZIONE DI UN ARRAY BIDIMENSIONALE

```
// Inizializza un array bidimensionale alla dichiarazione
int voti[10][8] = { {8, 10, 10, 10, 10, 9, 7, 10},
                    {7, 10, 10, 10, 9, 7, 8, 10},
                    {7, 6, 6, 9, 10, 6, 9, 8},
                    {6, 9, 6, 6, 7, 8, 10, 9},
                    {10, 9, 6, 8, 8, 9, 7, 9},
                    {7, 8, 10, 6, 7, 10, 10, 8},
                    {10, 9, 7, 9, 6, 6, 9, 10},
                    {9, 9, 7, 10, 6, 6, 10, 8},
                    {6, 9, 8, 9, 6, 10, 8, 10},
                    {9, 7, 6, 9, 6, 7, 6, 9} };
```

L'Esempio di codice C eC.27 mostra alcune funzioni che operano sull'array `voti` dell'Esempio di codice C eC.26. Gli array multidimensionali passati come argomenti alle funzioni devono definire tutte le dimensioni tranne la prima. Quindi entrambi i seguenti prototipi di funzione sono accettabili:

```
void visual2dArray(int arr[10][8]);
void visual2dArray(int arr[][8]);
```

ESEMPIO DI CODICE C eC.27 OPERAZIONI SU UN ARRAY BIDIMENSIONALE

```
#include <stdio.h>

// Visualizza il contenuto di un array 10 × 8
void visual2dArray(int arr[10][8])
{
  int i, j;

  for (i=0; i<10; i++) { // per ciascuno dei 10 studenti
    printf("Riga %d\n", i);
    for (j=0; j<8; j++) {
     printf("%d ", arr[i][j]); // visualizza i voti in ciascuno degli 8 compiti
    }
    printf("\n");
  }
}
```

```
// Calcola il voto medio di un array 10 × 8
float calcolaMedia(int arr[10][8])
{
  int i, j;
  float media, totale = 0;
  // calcola il valor medio di un array bidimensionale
  for (i=0; i<10; i++) {
    for (j=0; j<8; j++) {
      totale += arr[i][j]; // somma i valori presenti nell'array
    }
  }
  media = totale/(10*8);
  printf("La media e': %f\n", media);
  return media;
}
```

Si noti che dal momento che l'array è un puntatore all'elemento iniziale, non è possibile copiare o confrontare interi array con gli operatori = e ==: si deve invece usare un ciclo per copiare o confrontare i singoli elementi uno alla volta.

C.8.3 Caratteri

Un carattere (`char`) è una variabile di 8 bit. Può essere visto sia come un numero in complemento a due compreso fra –128 e 127, sia come il codice ASCII di una lettera, una cifra o un simbolo. I caratteri ASCII possono essere specificati come valori numerici (in decimale, esadecimale ecc.) o come caratteri stampabili racchiusi fra singoli apici. Per esempio, la lettera A ha il codice ASCII 0x41, la B il codice 0x42, e così via. Quindi `'A'+3` vale 0x44, cioè `'D'`. La Tabella 6.5 elenca le codifiche ASCII, e la **Tabella eC.4** elenca le notazioni usate in C per indicare caratteri speciali. I caratteri speciali includono il ritorno a inizio riga (o *carriage return*: `\r`), il ritorno a capo (o *newline*: `\n`), la tabulazione orizzontale (`\t`) e la fine di una stringa (`\0`). `\r` è indicato per completezza, ma viene usato molto raramente in C: riporta il carrello (cioè il cursore, ovvero la posizione da cui saranno visualizzati i caratteri successivi) all'inizio della riga corrente, quindi i caratteri successivi sovrascrivono i precedenti. `\n` invece sposta il cursore all'inizio di una nuova riga[2]. Il carattere *NULL* (`'\0'`) indica il termine di una stringa e viene discusso nel paragrafo C.8.4.

Tabella eC.4 Caratteri speciali.

Carattere speciale	Codifica	Descrizione
`\r`	0x0D	ritorno a inizio riga
`\n`	0x0A	ritorno a capo
`\t`	0x09	tabulazione
`\0`	0x00	terminazione di una stringa
`\\`	0x5C	barra rovesciata (*backslash*)
`\"`	0x22	doppio apice
`\'`	0x27	apice singolo
`\a`	0x07	avviso sonoro (*bell*)

Il termine «ritorno di carrello» risale alle macchine da scrivere per le quali era necessario che il carrello, ovvero il cilindro che teneva il foglio di carta, venisse spostato a destra per consentire di scrivere nella parte sinistra del foglio. La leva per il ritorno del carrello, mostrata a sinistra nella fotografia, veniva azionata per spostare a destra il carrello e far avanzare la carta di una riga (operazione denominata *line feed*).

Una macchina da scrivere elettrica usata da Winston Churchill. (http://cwr.iwm.org.uk/server/show/conMediaFile.71979)

[2] I file di testo in ambiente Windows usano `\r\n` per rappresentare la fine di una riga, mentre i sistemi basati su UNIX usano solo `\n`, il che causa non pochi problemi passando file di testo da un sistema all'altro.

C.8.4 Stringhe

Una stringa è un array di caratteri usato per memorizzare un pezzo di testo di lunghezza delimitata ma variabile. Ogni carattere è un byte che contiene il codice ASCII di una lettera, una cifra o un simbolo del pezzo di testo. La dimensione dell'array determina la massima lunghezza della stringa, che però può essere più corta: in C tale lunghezza viene infatti determinata cercando il terminatore NULL (il valore ASCII 0x00) alla fine della stringa.

L'Esempio di codice C eC.28 mostra la dichiarazione di un array di caratteri di 10 elementi denominato `saluto` che contiene la stringa `"Hello!"`. Si supponga che tale array parta dall'indirizzo di memoria 0x50: la **Figura eC.5** mostra il contenuto della memoria da 0x50 a 0x59 contenente la stringa `"Hello!"`: si noti che tale stringa occupa solo i primi sette elementi dell'array, anche se in memoria sono stati riservati dieci elementi.

Le stringhe in C sono dette *null terminated* o *zero terminated* perché la loro lunghezza viene determinata dalla posizione di uno zero usato come terminatore. Al contrario, linguaggi come il Pascal usano il primo byte per specificare la lunghezza della stringa, che può essere al massimo di 255 caratteri. Tale byte viene detto *byte di prefisso* e le stringhe di questo tipo sono denominate *P-string*. Un vantaggio delle stringhe null terminated è il fatto che possono essere di lunghezza qualsiasi. Un vantaggio delle P-string è il fatto che la lunghezza può essere determinata immediatamente senza dover scandire l'intera stringa alla ricerca dello zero terminatore.

ESEMPIO DI CODICE C eC.28 DICHIARAZIONE DI STRINGA

```
char saluto[10]= "Hello!";
```

L'Esempio di codice C eC.29 mostra una dichiarazione alternativa della stringa `saluto`: il puntatore `saluto` contiene l'indirizzo del primo elemento di un array a 7 elementi che contiene tutti i caratteri di `"Hello!"` seguiti dallo zero terminatore. Il codice mostra anche come visualizzare la stringa con il codice di formato `%s`.

ESEMPIO DI CODICE C eC.29 DICHIARAZIONE ALTERNATIVA DI STRINGA

```
char *saluto= "Hello!";
printf ("saluto: %s",saluto);
```

Visualizzazione sulla console

```
saluto: Hello!
```

A differenza delle variabili primitive, una stringa non può essere resa uguale a un'altra stringa con l'operatore di assegnamento, =. Ogni elemento dell'array di caratteri deve essere individualmente copiato dalla stringa sorgente alla stringa destinazione, come per qualsiasi altro array. L'Esempio di codice C eC.30 copia una stringa, `sorg`, in un'altra, `dest`. Le dimensioni delle due stringhe non sono necessarie, perché la fine della stringa `sorg` è indicata dallo zero terminatore. Naturalmente `dest` deve essere lunga abbastanza da contenere tutti i caratteri di `sorg` senza andare a sporcare altri dati. La funzione `strcpy` e altre funzioni di manipolazione stringhe sono disponibili nelle librerie predefinite del C (*vedi* par. C.9.4).

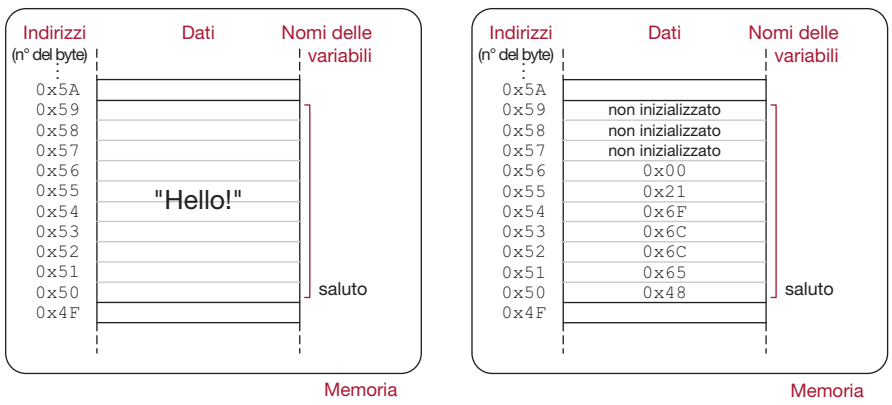


Figura eC.5
La stringa `"Hello!"` memorizzata in memoria.

ESEMPIO DI CODICE C eC.30 COPIA DI STRINGHE

```
// Copia la stringa sorgente, sorg, nella stringa destinazione, dest
void strcpy(char *dest, char *sorg)
{
  int i = 0;

  do {
    dest[i] = sorg[i]; // copia i caratteri uno alla volta
  } while (sorg[i++]); // finché non trova il terminatore null
}
```

C.8.5 Strutture

In C le strutture sono usate per memorizzare gruppi di dati di vario tipo. Il formato generale di dichiarazione di una struttura è:

```
struct nome {
  tipo1 elemento1;
  tipo2 elemento2;
  …
};
```

dove `struct` è la parola chiave che indica che si tratta di una struttura, `nome` è l'etichetta della struttura ed `elemento1` ed `elemento2` sono i componenti della struttura. L'Esempio di codice C eC.31 mostra come usare una struttura per memorizzare le informazioni di contatto di una persona. Il programma può poi dichiarare la variabile `c1` di tipo `struct contatto`.

ESEMPIO DI CODICE C eC.31 DICHIARAZIONE DI STRUTTURA

```
struct contatto {
  char nome[30];
  int tel;
  float altezza;  // in metri
};

struct contatto c1;
strcpy(c1.nome, "Ben Imbrogliabit");
c1.tel = 7226993;
c1.altezza = 1.82;
```

Come per i tipi predefiniti del C, si possono creare array di strutture e puntatori a strutture. L'Esempio di codice C eC.32 crea un array di contatti.

ESEMPIO DI CODICE C eC.32 ARRAY DI STRUTTURE

```
struct contatto elencostudenti[200];
elencostudenti[0].tel = 9642025;
```

È pratica comune usare puntatori a strutture. Il C fornisce l'**operatore di accesso a componente** -> per dereferenziare un puntatore a struttura e accedere a un componente della struttura. L'Esempio di codice C eC.33 mostra come dichiarare un puntatore a una `struct contatto`, come farlo puntare al 42° elemento dell'`elencostudenti` dichiarato nell'Esempio di codice C eC.32 e come usare l'operatore di accesso a componente per assegnare un valore a tale elemento.

ESEMPIO DI CODICE C eC.33 ACCESSO A COMPONENTI DI STRUTTURE CON PUNTATORE E OPERATORE

```
struct contatto *cpunt;
cpunt = &elencostudenti[41];
cpunt->altezza = 1.9  // equivale a (*cpunt).altezza = 1.9;
```

Le strutture possono essere passate come ingressi o uscite di funzioni per valore o per riferimento. Il passaggio per valore richiede al compilatore di copiare l'intera struttura in memoria per consentire alla funzione di usarla, cosa che può richiedere molta memoria se la struttura è grande. Il passaggio per riferimento richiede solo il passaggio di un puntatore, quindi risulta molto più efficiente. Inoltre in questo modo la funzione può modificare la struttura senza dover restituire una seconda struttura. L'Esempio di codice C eC.34 mostra due versioni della funzione `stira` che rende un contatto 2 cm più alto: `stiraPerRiferimento` evita di copiare due volte la grossa struttura.

ESEMPIO DI CODICE C eC.34 PASSAGGIO DI STRUTTURE PER VALORE O PER NOME

```
struct contatto stiraPerValore(struct contatto c)
{
  c.altezza += 0.02;
  return c;
}

void stiraPerRiferimento(struct contatto *cpunt)
{
  cpunt->altezza += 0.02;
}

int main(void)
{
  struct contatto Giorgio;

  Giorgio.altezza = 1.4;              // poveretto, è chino sui libri
  Giorgio = stiraPerValore(Giorgio);  // stirato una prima volta
  stiraPerRiferimento(&Giorgio);      // e stirato una seconda volta
}
```

C.8.6 `typedef`

Il C consente di definire dei nomi per i propri tipi di dati usando l'istruzione `typedef`. Per esempio, è noioso dover scrivere ogni volta `struct contatto` se si deve citare spesso la struttura: si può definire il tipo `cont` e usarlo come mostrato nell'Esempio di codice C eC.35.

ESEMPIO DI CODICE C eC.35 CREAZIONE DI UN PROPRIO TIPO CON `typedef`

```
typedef struct contatto {
  char nome[30];
  int tel;
  float altezza;  // in metri
} cont;           // definisce "cont" come abbreviazione di "struct contatto"

cont c1;          // si può ora definire una variabile di tipo "cont"
```

`typedef` può anche essere usata per creare un nuovo tipo che occupa esattamente lo stesso spazio di un tipo predefinito. L'Esempio di codice C eC.36 definisce i due tipi a 8 bit `byte` e `booleano`. Il tipo `byte` serve a chiarire che la variabile `pos` è un numero a 8 bit invece di un carattere. Il tipo `booleano` serve a chiarire che il numero a 8 bit rappresenta uno dei due valori VERO o FALSO. I due tipi rendono il programma molto più leggibile di quanto non sarebbe usando ovunque il solo tipo `char`.

ESEMPIO DI CODICE C eC.36 `typedef byte E booleano`

```
typedef unsigned char byte;
typedef char booleano;
#define VERO 1
#define FALSO 0

byte pos = 0x45
booleano promosso = VERO;
```

L'Esempio di codice C eC.37 mostra come definire il tipo `vettore` di 3 elementi e il tipo `matrice` di 3 × 3 elementi.

ESEMPIO DI CODICE C eC.37 `typedef vettore E matrice`

```
typedef double vettore[3];
typedef double matrice[3][3];

vettore a = {4.5, 2.3, 7.0};
matrice b = {{3.3, 4.7, 9.2}, {2.5, 4, 9}, {3.1, 99.2, 88}};
```

C.8.7 Allocazione dinamica della memoria*

In tutti gli esempi fatti finora, le variabili sono state allocate **staticamente**: ciò significa che la loro dimensione è nota al momento della compilazione. La cosa può essere problematica per gli array e le stringhe di dimensione variabile, perché devono essere dichiarati grandi abbastanza per contenere la massima dimensione che il programma potrebbe incontrare. L'alternativa è usare un'allocazione **dinamica** della memoria al momento dell'esecuzione, quando si conosce la dimensione effettiva necessaria.

La funzione `malloc` della libreria `stdlib.h` alloca un blocco di memoria di dimensione specificata e restituisce il puntatore a tale blocco. Se la memoria disponibile non è sufficiente, viene restituito il valore NULL. Per esempio, il codice seguente alloca 10 variabili di tipo `short` (10 × 2 = 20 byte). L'operatore `sizeof` restituisce la dimensione del tipo di variabile in byte.

```
// alloca dinamicamente 20 byte di memoria
short *dati = malloc(10*sizeof(short));
```

L'Esempio di codice C eC.38 mostra l'allocazione e la de-allocazione dinamiche. Il programma accetta un numero variabile di valori inseriti, li memorizza in un array allocato dinamicamente e ne calcola la media. La quantità di memoria necessaria dipende dal numero di valori nell'array e dalla dimensione di ogni elemento. Per esempio, se un `int` è una variabile di 4 byte e servono 10 elementi, verranno allocati 40 byte. La funzione `free` de-alloca la memoria in modo che possa essere usata successivamente per altri scopi. La mancata de-allocazione viene definita **perdita di memoria** (*memory leak*) e va evitata.

ESEMPIO DI CODICE C eC.38 **ALLOCAZIONE E DE-ALLOCAZIONE DINAMICA DI MEMORIA**

```
// Alloca e de-alloca dinamicamente un array usando malloc e free
#include <stdlib.h>

// Inserire la funzione calcolaMedia dell'Esempio di codice C eC.24.

int main(void) {
  int lun, i;
  int *numeri;

  printf("Quanti valori vuoi inserire? ");
  scanf("%d", &lun);
 numeri = malloc(lun*sizeof(int));
```

```
  if (numeri == NULL) printf("ERRORE: memoria esaurita.\n");
  else {
    for (i=0; i<lun; i++) {
      printf("Inserisci un numero: ");
      scanf("%d", &numeri[i]);
    }
    printf("La media e' %f\n", calcolaMedia(numeri, lun));
  }
  free(numeri);
}
```

C.8.8 Liste collegate*

Una **lista collegata** è una struttura dati comune, usata per memorizzare un numero variabile di elementi. Ogni elemento della lista è una struttura contenente uno o più dati e un collegamento al prossimo elemento. Il primo elemento è definito **testa** della lista. Le liste collegate illustrano molti dei concetti relativi alle strutture, ai puntatori e all'allocazione dinamica della memoria.

L'Esempio di codice C eC.39 mostra una lista collegata usata per memorizzare gli account degli utenti di un calcolatore il cui numero è variabile. Ogni utente è caratterizzato da username, password, numero identificativo univoco (UID, *User Identification Number*) e un campo che indica se l'utente abbia o meno i privilegi di amministratore. Ogni elemento della lista è di tipo `utenteL`, e contiene tutte le suddette informazioni più il collegamento al prossimo elemento della lista. Il puntatore alla testa della lista è memorizzato nella variabile globale `utenti`, inizializzata a NULL per indicare che non ci sono ancora utenti.

Il programma definisce le funzioni per inserire, eliminare e trovare un utente, e per contare il numero di utenti. La funzione `inserisciU` alloca spazio per un nuovo elemento della lista e lo aggiunge alla testa. La funzione `cancellaU` percorre la lista fino a trovare l'utente specificato dal proprio UID, elimina tale utente sistemando il collegamento dall'elemento precedente in modo che salti l'elemento da eliminare, e de-alloca la memoria occupata dall'elemento eliminato. La funzione `trovaU` percorre la lista fino a trovare l'utente specificato dal proprio UID e restituisce un puntatore a tale elemento, o il valore NULL se non trova l'utente. La funzione `numeroU` conta il numero di elementi della lista.

ESEMPIO DI CODICE C eC.39 LISTA COLLEGATA

```
#include <stdlib.h>
#include <string.h>

typedef struct utenteL {
  char uname[80];        // username
  char passwd[80];       // password
  int uid;               // user identification number
  int admin;             // 1 indica privilege di amministratore
  struct utenteL *pross;
} utenteL;

utenteL *utenti = NULL;

void inserisciU(char *uname, char *passwd, int uid, int admin) {
  utenteL *nuovoU;

  nuovoU = malloc(sizeof(utenteL)); // crea spazio per il nuovo utente
  strcpy(nuovoU->uname, uname);    // copia i valori nei campi dell'utente
  strcpy(nuovoU->passwd, passwd);
  nuovoU->uid = uid;
  nuovoU->admin = admin;
```

```
  nuovoU->pross = utenti;          // inserisce all'inizio della lista
  utenti = nuovoU;
}
void cancellaU(int uid) {          // cancella l'utente corrispondente all'uid
  utenteL *corr = utenti;
  utenteL *prec = NULL;

  while (corr != NULL) {
    if (corr->uid == uid) {        // trovato utente da eliminare
      if (prec == NULL) utenti = corr->pross;
      else prec->pross = corr->pross;
      free(cur);
      return; // done
    }
    prec = corr;                   // altrimenti procede a esaminare la lista
    corr = corr->pross;
  }
}
utenteL *trovaU(int uid) {
  utenteL *corr = utenti;

  while (corr != NULL) {
    if (corr->uid == uid) return corr;
    else corr = corr->pross;
  }
  return NULL;
}

int numeroU(void) {
  utenteL *corr = utenti;
  int cont = 0;

  while (corr != NULL) {
    cont++;
    corr = corr->pross;
  }
  return cont;
}
```

RIASSUNTO

- **Puntatori:** un puntatore contiene l'indirizzo di una variabile.
- **Array:** un array è una sequenza di elementi simili, dichiarata usando le parentesi quadre `[]`.
- **Caratteri:** il tipo `char` può memorizzare numeri interi piccoli oppure codici speciali per rappresentare testo o simboli.
- **Stringhe:** una stringa è un array di caratteri terminata dal terminatore NULL 0x00.
- **Strutture:** una struttura memorizza un insieme di variabili imparentate.
- **Allocazione dinamica della memoria:** la funzione `malloc` è una funzione predefinita per allocare memoria durante l'esecuzione del programma. La funzione `free` de-alloca la memoria dopo l'uso.
- **Liste collegate:** una lista collegata è una struttura dati comune per memorizzare un numero variabile di elementi.

C.9 ■ LIBRERIE STANDARD

I programmatori usano normalmente una varietà di funzioni standard, per esempio per visualizzare informazioni o per effettuare calcoli trigonometrici. Per evitare che ogni programmatore debba scriversi queste funzioni da zero, il C fornisce **librerie** di funzioni di uso frequente. Ogni libreria è un file header associato a un file oggetto, ovvero un file C parzialmente compilato. Il file header contiene le dichiarazioni di variabili, le definizioni di tipi e i prototipi delle funzioni. Il file oggetto contiene le funzioni e viene collegato durante la compilazione per costruire l'eseguibile. Dal momento che le funzioni sono già compilate nel file oggetto, il tempo di compilazione risulta ridotto. La **Tabella eC.5** elenca alcune delle librerie C di uso più frequente, descritte brevemente nel seguito.

C.9.1 `stdio`

La libreria standard di input/output `stdio.h` contiene funzioni per visualizzare sulla console, leggere l'ingresso da tastiera, e leggere e scrivere file. Per poter usare queste funzioni, la libreria deve essere inclusa all'inizio del file:

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

`printf`

La funzione `printf` (*print formatted*) visualizza del testo sulla console. Il suo argomento obbligatorio è una stringa racchiusa fra doppi apici `"  "` che contiene il testo e dei comandi opzionali per visualizzare variabili. Le variabili da visualizzare sono elencate dopo la stringa e sono visualizzate usando i codici di formato elencati nella **Tabella eC.6**. L'Esempio di codice C eC.40 mostra un semplice caso di uso di `printf`.

ESEMPIO DI CODICE C eC.40 VISUALIZZAZIONE SULLA CONSOLE MEDIANTE `printf`

```
// Semplice visualizzazione
#include <stdio.h>

int numero = 42;

int main(void) {
  printf("La risposta e' %d.\n", numero);
}
```

Visualizzazione sulla console

```
La risposta è 42.
```

I formati per i numeri in virgola mobile (`float` e `double`) per default visualizzano sei cifre dopo la virgola. Per modificare la precisione si deve sostituire `%f` con `%w.df`, dove `w` indica la dimensione minima del numero, e `d` il numero di cifre decimali da visualizzare. Si noti che il punto decimale è incluso nella dimensione minima. Nell'Esempio di codice C eC.41, `pi` viene visualizzato in quattro caratteri, due dei quali dopo il punto decimale: 3.14. `e` viene visualizzato in otto caratteri, tre dei quali dopo il punto decimale; dal momento che ha una sola cifra prima del punto decimale, viene riempito di spazi per raggiungere la dimensione richiesta. `c` dovrebbe venire visualizzato in cinque caratteri, tre dei quali dopo il punto decimale, ma siccome è troppo grande la dimensione richiesta viene ignorata anche se si mantengono le tre cifre dopo il punto decimale.

Tabella eC.5 Librerie C di uso frequente.

Header file della libreria C	Descrizione
`stdio.h`	**Libreria standard di input/output.** Include le funzioni per scrivere/leggere su terminale o su file (`printf`, `fprintf` e `scanf`, `fscanf`) e per aprire e chiudere file (`fopen` e `fclose`).
`stdlib.h`	**Liberia standard.** Include funzioni per la generazione di numeri casuali (`rand` e `srand`), per allocare e de-allocare dinamicamente memoria (`malloc` e `free`), per terminare anticipatamente un programma (`exit`) e per effettuare conversioni da stringhe a numeri (`atoi`, `atol` e `atof`).
`math.h`	**Libreria matematica.** Include funzioni matematiche standard come `sin`, `cos`, `asin`, `acos`, `sqrt`, `log`, `log10`, `exp`, `floor` e `ceil`.
`string.h`	**Libreria per le stringhe.** Include funzioni per confrontare, copiare e concatenare stringhe e per determinarne la lunghezza.

Tabella eC.6 Codici di formato di `printf` per visualizzare variabili.

Codice	Formato
`%d`	Decimale
`%u`	Decimale senza segno
`%x`	Esadecimale
`%o`	Ottale
`%f`	Numero in virgola mobile (`float` o `double`)
`%e`	Numero in virgola mobile (`float` o `double`) in notazione scientifica (per es. 1.56e7)
`%c`	Carattere (`char`)
`%s`	Stringa (array di caratteri terminato dal terminatore NULL)

ESEMPIO DI CODICE C eC.41 FORMATI PER LA VISUALIZZAZIONE DI NUMERI IN VIRGOLA MOBILE

```
// visualizzazione di numeri in virgola mobile in diversi formati
float pi = 3.14159, e = 2.7182, c = 2.998e8;
printf("pi = %4.2f\ne  = %8.3f\nc  = %5.3f\n", pi, e, c);
```

Visualizzazione sulla console

```
pi = 3.14
e  =    2.718
c  = 299800000.000
```

Dal momento che `%` e `\` sono utilizzati per indicare le formattazioni, per visualizzare tali caratteri si devono usare le sequenze speciali mostrate nell'Esempio di codice C eC.42.

ESEMPIO DI CODICE C eC.42 VISUALIZZAZIONE DI `%` E `\` CON `printf`

```
// Come visualizzare % e \ sulla console
printf("Ecco alcuni caratteri speciali: %% \\ \n");
```

Visualizzazione sulla console

```
Ecco alcuni caratteri speciali: % \
```

scanf

La funzione `scanf` legge il testo digitato sulla tastiera; usa gli stessi codici di formato di `printf`. L'Esempio di codice C eC.43 mostra come usare `scanf`. Quando incontra `scanf`, il programma attende che l'utente inserisca un valore prima di continuare l'esecuzione. Gli argomenti di `scanf` sono la stringa che indica uno o più codici di formato e i puntatori alle variabili nelle quali memorizzare i risultati.

ESEMPIO DI CODICE C eC.43 LETTURA DI IMMISSIONI DELL'UTENTE DA TASTIERA CON `scanf`

```
// Legge variabili dalla riga di comando
#include <stdio.h>

int main(void)
{
  int a;
  char str[80];
  float f;

  printf("Inserire un intero.\n");
  scanf("%d", &a);
  printf("Inserire un numero in virgola mobile.\n");
  scanf("%f", &f);
  printf("Inserire una stringa.\n");
  scanf("%s", str); // si noti che non serve &: str è già un puntatore
}
```

Gestione di file

Molti programmi devono leggere e scrivere file, per operare su dati memorizzati in un file o per salvare grandi quantità di informazioni. In C un file deve per prima cosa essere aperto con `fopen`. Poi può essere letto o scritto con `fscanf` e `fprintf` in modo analogo a quanto avviene con la console. Infine il file deve essere chiuso con `fclose`.

La funzione `fopen` riceve come argomenti il nome del file e la **modalità di apertura**, e restituisce un **puntatore a file** di tipo `FILE*`. Se non è in grado di aprire il file, restituisce NULL: questo può succedere per esempio se si cerca di leggere un file inesistente o di scrivere un file già aperto da un altro programma. Le modalità di apertura sono:

`"w"`: Scrivere su file. Se il file esiste, viene sovrascritto.
`"r"`: Leggere da file.
`"w"`: Appendere alla fine di un file esistente. Se il file non esiste, viene creato.

L'Esempio di codice C eC.44 mostra come aprire un file, scrivere nel file e chiuderlo. È buona norma controllare sempre se il file è stato aperto correttamente, e dare un messaggio di errore in caso contrario. La funzione `exit` è discussa nel paragrafo C.9.2. La funzione `fprintf` è simile alla `printf` ma richiede anche un puntatore a file come argomento di ingresso per sapere su quale file scrivere. La funzione `fclose` chiude il file, garantendo che tutte le informazioni siano state effettivamente salvate su disco e liberando le risorse del file system.

È tipico aprire un file e verificare se il puntatore restituito vale NULL con una singola riga di codice, come mostrato nell'Esempio di codice C eC.44. Si può comunque fare la stessa cosa su due righe di codice nel modo seguente:

```
fptr = fopen("risultato.txt", "w");
if (fptr == NULL)
...
```

ESEMPIO DI CODICE C eC.44 SCRITTURA SU FILE CON `fprintf`

```
// Scrive "Prova di scrittura su file." su risultato.txt
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  FILE *fptr;

  if ((fptr = fopen("risultato.txt", "w")) == NULL) {
    printf("Impossibile aprire risultato.txt in scrittura.\n");
    exit(1); // termina il programma indicando errore di esecuzione
  }
  fprintf(fptr, "Prova di scrittura su file.\n");
  fclose(fptr);
}
```

L'Esempio di codice C eC.45 mostra come leggere dei numeri dal file di nome `dati.txt` mediante `fscanf`. Il file deve per prima cosa essere aperto in lettura. Il programma usa poi la funzione `feof` per sapere se è arrivato alla fine del file. Finché non è arrivato alla fine, legge il prossimo numero e lo visualizza sullo schermo. Come prima, il programma chiude il file alla fine per liberare risorse.

ESEMPIO DI CODICE C eC.45 LETTURA DA FILE CON `fscanf`

```
#include <stdio.h>

int main(void)
{
  FILE *fptr;
  int dato;

  // legge dato dal file di ingresso
  if ((fptr = fopen("dati.txt", "r")) == NULL) {
    printf("Impossibile leggere il file dati.txt\n");
    exit(1);
  }
  while (!feof(fptr)) { // verifica se non è arrivato a fine file
    fscanf(fptr, "%d", &dato);
    printf("Dato letto: %d\n", dato);
    }
  fclose(fptr);
}
```

dati.txt

```
25 32 14 89
```

Visualizzazione sulla console

```
Dato letto: 25
Dato letto: 32
Dato letto: 14
Dato letto: 89
```

Altre comode funzioni di stdio

La funzione `sprintf` scrive caratteri in una stringa e la funzione `sscanf` legge variabili da una stringa. La funzione `fgetc` legge un singolo carattere da un file e la funzione `fgets` legge una riga completa da un file in una stringa.

`fscanf` è alquanto limitata come possibilità di leggere e analizzare file complessi, quindi è spesso più semplice leggere una riga alla volta con `fgets` e assimilare la riga con `sscanf`, o scrivere un ciclo che analizza un carattere alla volta con `fgetc`.

C.9.2 `stdlib`

La libreria standard `stdlib.h` fornisce funzioni di tipo generale che includono la generazione di numeri casuali (`rand` e `srand`), l'allocazione e de-allocazione dinamica della memoria (`malloc` e `free`, già discusse nel paragrafo C.8.7), la terminazione anticipata di un programma (`exit`) e le conversioni di formato dei numeri. Per poter usare queste funzioni, si deve aggiungere la riga seguente all'inizio del file C:

```
#include <stdlib.h>
```

`rand` e `srand`

La funzione `rand` restituisce un numero intero pseudo-casuale. I numeri pseudo-casuali hanno le caratteristiche statistiche dei numeri casuali ma seguono uno schema deterministico a partire da un valore iniziale chiamato **seme** (*seed*). Per far rientrare il numero in un particolare intervallo, si deve usare l'operatore resto o modulo (`%`) come mostrato nell'Esempio di codice C eC.46 per l'intervallo da 0 a 9. I valori `x` e `y` sono casuali, ma saranno sempre gli stessi a ogni esecuzione del programma. La visualizzazione su console è riportata alla fine del codice.

ESEMPIO DI CODICE C eC.46 GENERAZIONE DI NUMERI CASUALI CON `rand`

```
#include <stdlib.h>
int x, y;

x = rand();      // x = numero intero casuale
y = rand() % 10; // y = numero intero casuale compreso tra 0 e 9
printf("x = %d, y = %d\n", x, y);
```

Visualizzazione sulla console

```
x = 1481765933, y = 3
```

Il programmatore può generare diverse sequenze di numeri casuali a ogni esecuzione del programma modificando il seme. Questo si fa con la funzione `srand`, che riceve il seme come argomento di ingresso. Come mostrato nell'Esempio di codice C eC.47, il seme deve essere a sua volta un valore casuale, quindi il tipico programma C lo definisce chiamando la funzione `time` che restituisce l'ora corrente in secondi.

ESEMPIO DI CODICE C eC.47 ASSEGNAMENTO DEL SEME DEL GENERATORE DI NUMERI CASUALI CON `srand`

```
// Produce un numero casuale differente a ogni esecuzione
#include <stdlib.h>
#include <time.h>    // necessaria per poter chiamare time()

int main(void)
  {
  int x;
  srand(time(NULL)); // assegna il seme al generatore di numeri casuali
  x = rand() % 10;   // numero casuale compreso tra 0 e 9
  printf("x = %d\n", x);
}
```

Per ragionini storiche, la funzione `time` restituisce di solito l'ora corrente in secondi relativa al 1 gennaio 1970 00:00 UTC. UTC sta per Coordinated Universal `Time`, che corrisponde all'ora di Greenwich (GMT: Greenwich Mean `Time`). La data è di poco successiva alla creazione di UNIX da parte del gruppo di lavoro ai Laboratori Bell nel 1969, che comprendeva Dennis Ritchie e Brian Kernighan. Come per i veglioni di capodanno, alcuni fan di UNIX organizzano feste in occasione di particolari valori restituiti da `time`. Per esempio, alle 23:31:30 UTC del 1 febbraio 2009, il valore restituito da `time` è stato 1 234 567 890. Nel 2038 gli orologi UNIX nelle versioni a 32 bit avranno un traboccamento e torneranno al 1901.

`exit`

La funzione `exit` termina anticipatamente l'esecuzione del programma. Riceve un solo argomento che viene restituito al sistema operativo per indicare la ragione della terminazione. Il valore 0 indica terminazione normale, mentre un valore diverso da 0 indica una condizione di errore.

Conversioni di formato: `atoi`, `atol`, `atof`

La libreria standard fornisce funzioni per convertire stringhe di caratteri ASCII in numeri interi, numeri interi lunghi e numeri in virgola mobile in doppia precisione rispettivamente con le funzioni `atoi`, `atol` e `atof`, come mostrato nell'Esempio di codice C eC.48. Sono particolarmente utili quando si leggono dati misti (cioè una miscela di stringhe e numeri) da un file o quando si devono elaborare dati numerici dalla riga di comando, come descritto nel paragrafo C.10.3.

ESEMPIO DI CODICE C eC.48 CONVERSIONI DI FORMATO

```
// Converte stringhe ASCII in interi, interi lunghi e numeri in virgola
mobile
#include <stdlib.h>

int main(void)
{
  int x;
  long int y;
  double z;

  x = atoi("42");
  y = atol("833");
  z = atof("3.822");
  printf("x = %d\ty = %d\tz = %f\n", x, y, z);
}
```

Visualizzazione sulla console

```
x = 42 y = 833 z = 3.822000
```

C.9.3 `math`

La libreria matematica `math.h` fornisce funzioni matematiche di uso frequente come funzioni trigonometriche, radice quadrata, logaritmi. L'Esempio di codice C eC.49 mostra come usare alcune di queste funzioni. Per poter usare le funzioni matematiche, si deve aggiungere la riga seguente all'inizio del file C:

```
#include <math.h>
```

ESEMPIO DI CODICE C eC.49 FUNZIONI MATEMATICHE

```
// Esempi di funzioni matematiche
#include <stdio.h>
#include <math.h>

int main(void) {
  float a, b, c, d, e, f, g, h;

  a = cos(0);         // 1: l'argomento è espresso in radianti
  b = 2 * acos(0);    // pi: acos significa arcoseno
  c = sqrt(144);      // 12
  d = exp(2);         // e^2 = 7.389056,
  e = log(7.389056);  // 2 (logaritmo naturale, in base e)
  f = log10(1000);    // 3 (logaritmo in base 10)
  g = floor(178.567); // 178, arrotonda all'intero per difetto
  h = pow(2, 10);     // calcola 2 elevato alla 10ma potenza
  printf("a = %.0f, b = %f, c = %.0f, d = %.0f, e = %.2f,
          f = %.0f, g = %.2f, h =%.2f\n",
          a, b, c, d, e, f, g, h);
}
```

Visualizzazione sulla console

```
a = 1, b = 3.141593, c = 12, d = 7, e = 2.00, f = 3, g = 178.00, h = 1024.00
```

C.9.4 `string`

La libreria `string.h` fornisce funzioni di uso comune per la manipolazione di stringhe. Le funzioni più importanti sono:

```
// Copia sorg in dest e restituisce dest
char *strcpy(char *dest, char *sorg);

// Concatena (appende) sorg alla fine di dest e restituisce dest
char *strcat(char *dest, char *sorg);

// Confronta due stringhe. Restituisce 0 se uguali, non zero se diverse
int strcmp(char *s1, char *s2);

// Restituisce la lunghezza di stringa, senza includere il terminatore null
int strlen(char *stringa);
```

C.10 ■ OPZIONI DEL COMPILATORE E ARGOMENTI NELLA RIGA DI COMANDO

I programmi visti finora sono molto semplici, ma i programmi veri sono costituiti da decine fino anche a migliaia di file C per consentire modularità, leggibilità e cooperazione tra molti programmatori. Questo paragrafo descrive come compilare un programma distribuito su più file e mostra come usare le opzioni del compilatore e gli argomenti nella riga di comando.

C.10.1 Compilare più file sorgente C

Vari file C vengono compilati in un unico eseguibile elencando i nomi di tutti i file sulla riga di comando del compilatore, come mostrato sotto. Si ricordi che il gruppo di file C deve contenere una sola funzione `main`, convenzionalmente posta nel file `main.c`.

```
gcc main.c file2.c file3.c
```

C.10.2 Opzioni del compilatore

Le opzioni del compilatore consentono all'utente di specificare aspetti come i nomi e i formati dei file di uscita, i livelli di ottimizzazione ecc. Le opzioni del compilatore non sono standardizzate, ma la **Tabella eC.7** mostra quelle usate più comunemente. Ogni opzione è tipicamente preceduta da un trattino (`-`) nella riga di comando: per esempio, l'opzione `"-o"` consente al programmatore di indicare un nome di file di uscita diverso da `"a,out"` di default. Le opzioni sono numerosissime: si possono vedere con il comando `gcc --help`.

C.10.3 Argomenti nella riga di comando

Come le altre funzioni, anche `main` può ricevere variabili in ingresso come argomenti. A differenza delle altre funzioni però, tali argomenti sono specificati nella riga di comando. Come mostrato nell'Esempio di codice C eC.50, `argc` sta per *argument count* (conteggio argomenti) e indica il numero di argomenti nella riga di comando; `argv` sta per *argument vector* (vettore degli argomenti) ed è un array delle stringhe incontrate nella riga di comando. Si supponga per esempio che il programma dell'Esempio di codice C eC.50 sia compilato nel file eseguibile denominato `verifArg`. Quando vengono inserite le righe sottostanti nella riga di comando, `argc` assume il valore 4 e l'array `argv` contiene i valori `{"./verifArg", "arg1", "25", "ultimoArg!"}`. La visualizzazione su console dopo aver digitato questi comandi è riportata alla fine dell'Esempio di codice C eC.50.

```
gcc -o verifArg verifArg.c
./verifArg arg1 25 ultimoArg!
```

Tabella eC.7 Opzioni del compilatore.

Opzione del compilatore	Descrizione	Esempio
`-o` file di uscita	Specifica il nome del file di uscita	`gcc -o hello hello.c`
`-S`	Crea un file di uscita in linguaggio assembly (non eseguibile)	`gcc -S hello.c` produce `hello.s`
`-v`	Modo verboso: visualizza i passi e i risultati del compilatore durante la compilazione	`gcc -v hello.c`
`-Olevel`	Specifica il livello di ottimizzazione (tipicamente da 0 a 3) producendo codice più rapido e/o più corto a scapito di un tempo di compilazione maggiore	`gcc -O3 hello.c`
`--version`	Mostra la versione del compilatore	`gcc --version`
`--help`	Elenca tutte le opzioni della riga di comando	`gcc --help`
`-Wall`	Visualizza tutti gli avvertimenti (*warning*)	`gcc -Wall hello.c`

I programmi che richiedono argomenti numerici possono convertire gli argomenti di tipo stringa in numeri con le funzioni di `stdlib.h`.

ESEMPIO DI CODICE C eC.50 ARGOMENTI NELLA RIGA DI COMANDO

```
// Visualizza gli argomenti nella riga di comando
#include <stdio.h>

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

  for (i=0; i<argc; i++)
  printf("argv[%d] = %s\n", i, argv[i]);
}
```

Visualizzazione sulla console

```
argv[0] = ./verifArg
argv[1] = arg1
argv[2] = 25
argv[3] = ultimoArg!
```

C.11 ■ ERRORI FREQUENTI

Come con ogni altro linguaggio di programmazione, si può star certi che si commetteranno errori scrivendo programmi in C che non siano molto banali. Nel seguito si descrivono alcuni errori frequenti, quando si programma in C: alcuni sono particolarmente insidiosi perché consentono una compilazione corretta ma non danno in esecuzione il comportamento che il programmatore si aspetta.

ERRORE NEL CODICE C eC.1 DIMENTICANZA DI & NELLA `scanf`

Codice errato

```
int a;
printf("Inserire un intero:\t");
scanf("%d", a); // dimenticato & prima di a
```

Codice corretto

```
int a;
printf("Inserire un intero:\t");
scanf("%d", &a);
```

ERRORE NEL CODICE C eC.2 USO DI = INVECE DI == NEI CONFRONTI

Codice errato

```
if (x = 1) // dà sempre risultato TRUE
  printf("Trovato!\n");
```

Codice corretto

```
if (x == 1)
  printf("Trovato!\n");
```

ERRORE NEL CODICE C eC.3 ACCESSO A UN ELEMENTO DI UN ARRAY DOPO L'ULTIMO

Codice errato

```
int array[10];
array[10] = 42; // l'indice varia da 0 a 9
```

Codice corretto

```
int array[10];
array[9] = 42;
```

ERRORE NEL CODICE C eC.4 USO DI = IN UNA DIRETTIVA `#define`

Codice errato

```
// sostituisce NUM con "= 4" nel codice
#define NUM = 4
```

Codice corretto

```
#define NUM 4
```

ERRORE NEL CODICE C eC.5 USO DI UNA VARIABILE NON INIZIALIZZATA

Codice errato

```
int i;
if (i == 10) // i non è inizializzata
  ...
```

Codice corretto

```
int i = 10;
if (i == 10)
  ...
```

ERRORE NEL CODICE C eC.6 PERCORSO DI UNA LIBRERIA CREATA DALL'UTENTE NON INDICATO

Codice errato

```
#include "miofile.h"
```

Codice corretto

```
#include "percorso\miofile.h"
```

ERRORE NEL CODICE C eC.7 USO DEGLI OPERATORI LOGICI (`!`, `||`, `&&`) INVECE DI QUELLI BIT A BIT (`~`, `|`, `&`)

Codice errato

```
char x=!5;   // NOT logico: x = 0
char y=5||2; // OR  logico: y = 1
char z=5&&2; // AND logico: z = 1
```

Codice corretto

```
char x=~5; // NOT bit a bit: x = 0b11111010
char y=5|2;// OR  bit a bit: y = 0b00000111
char z=5&2;// AND bit a bit: z = 0b00000000
```

ERRORE NEL CODICE C eC.8 DIMENTICANZA DI UN `break` IN UN'ISTRUZIONE `switch/case`

Codice errato

```
char x = 's';
...
switch (x) {
case 'n': direzione = 1;
case 's': direzione = 2;
case 'e': direzione = 3;
case 'o': direzione = 4;
default:  direzione = 0;
}
// direzione = 0
```

Codice corretto

```
char x = 's';
...
switch (x) {
case 'n': direzione = 1; break;
case 's': direzione = 2; break;
case 'e': direzione = 3; break;
case 'o': direzione = 4; break;
default:  direzione = 0;
}
// direzione = 2
```

ERRORE NEL CODICE C eC.9 DIMENTICANZA DELLE PARENTESI GRAFFE { }

Codice errato

```
if (ptr == NULL) // no parentesi graffe
  printf("File non trovato.\n");
  exit(1);       // viene eseguita sempre
```

Codice corretto

```
if (ptr == NULL) {
  printf("File non trovato.\n");
  exit(1);
}
```

La capacità di trovare gli errori nei programmi (*debugging*) si acquisisce con l'esperienza, ma ecco qui alcuni suggerimenti.

- Iniziare a correggere il programma partendo dal primo errore segnalato dal compilatore. Gli errori successivi possono essere conseguenza del primo. Dopo averlo corretto, ricompilare il programma e ripetere il procedimento finché tutti gli errori (almeno quelli che il compilatore è in grado di identificare…) sono stati eliminati.
- Se il compilatore indica un errore in una riga di codice che invece è corretta, verificare la riga precedente (dove potrebbe mancare un punto e virgola o una parentesi).
- Quando necessario, dividere istruzioni complesse su più righe.
- Inserire chiamate a `printf` per visualizzare risultati intermedi.
- Quando un risultato non corrisponde alle aspettative, iniziare a controllare il codice nel *primo* punto in cui si verifica la discrepanza.
- Esaminare tutti gli avvertimenti (*warning*) del compilatore: alcuni possono essere ignorati, ma altri possono far scoprire errori che non impediscono la compilazione ma che causano un comportamento non corretto in esecuzione.

ERRORE NEL CODICE C eC.10 USO DI UNA FUNZIONE PRIMA CHE SIA STATA DICHIARATA

Codice errato

```
int main(void)
{
  test();
}

void test(void)
{...
}
```

Codice corretto

```
void test(void)
{...
}

int main(void)
{
  test();
}
```

ERRORE NEL CODICE C eC.11 STESSO NOME PER UNA VARIABILE GLOBALE E PER UNA LOCALE

Codice errato

```
int x = 5;   // dichiarazione globale
 di x
int test(void)
{
  int x = 3; // dichiarazione locale
 di x
  ...
}
```

Codice corretto

```
int x = 5;   // dichiarazione
 globale di x
int test(void)
{
  int y = 3; // la variabile
 locale e' y
  ...
}
```

ERRORE NEL CODICE C eC.12 TENTATIVO DI INIZIALIZZARE UN ARRAY CON { } DOPO LA DICHIARAZIONE

Codice errato

```
int voti[3];
voti = {25, 23, 30};  //non viene compilato
```

Codice corretto

```
int voti[3] = {25, 23, 30};
```

ERRORE NEL CODICE C eC.13 ASSEGNAMENTO DI UN ARRAY A UN ALTRO CON =

Codice errato

```
int voti[3] = {25, 23, 30};
int voti2[3];

voti2 = voti;
```

Codice corretto

```
int voti[3] = {25, 23, 30};
int voti2[3];

for (i=0; i<3; i++)
  voti2[i] = voti[i];
```

ERRORE NEL CODICE C eC.14 DIMENTICANZA DEL PUNTO E VIRGOLA DOPO UN CICLO `do/while`

Codice errato

```
int num;

do {
  num = leggiNum();
} while (num < 100) // manca il ;
```

Codice corretto

```
int num;

do {
  num = leggiNum();
} while (num < 100);
```

ERRORE NEL CODICE C eC.15 USO DELLE VIRGOLE INVECE DEI PUNTI E VIRGOLA IN UN CICLO `for`

Codice errato

```
for (i=0, i<200, 1++)
  ...
```

Codice corretto

```
for (i=0; i<200; 1++)
  ...
```

ERRORE NEL CODICE C eC.16 **DIVISIONE INTERA INVECE DI DIVISIONE IN VIRGOLA MOBILE**

Codice errato	Codice corretto

`// la divisione intera (troncamento)`
`// si verifica quando entrambi gli`
`// operandi della divisione sono interi`
`float x = 9 / 4;  // x = 2.0`
	`// almeno un operando della divisione`
	`// deve essere un float per effettuare`
	`// la divisione in virgola mobile`
	`float x = 9.0 / 4;  // x = 2.25`

ERRORE NEL CODICE C eC.17 **SCRIVERE CON UN PUNTATORE NON INIZIALIZZATO**

Codice errato	Codice corretto
`int *y = 77;`	`int x, *y=&x;` `*y = 77;`

ERRORE NEL CODICE C eC.18 **GRANDI ATTESE (O NESSUNA ATTESA)**

Un tipico errore dei principianti è quello di scrivere un intero programma (di solito con scarsa modularità) e aspettarsi che funzioni al primo colpo. Per programmi non banali, è essenziale scrivere codice modulare e collaudare ogni singola funzione. Con la complessità, il debugging diventa esponenzialmente difficile e richiede un'enorme quantità di tempo.

Un altro tipico errore è quello di non avere un'attesa: quando succede, il programmatore può solo verificare che il programma produca un risultato, non che il risultato sia corretto. Collaudare un programma con valori di ingresso noti e conoscendo i risultati attesi è fondamentale per verificarne le funzionalità.

Questa appendice si è concentrata sull'uso del C su un sistema come un personal computer. Il Capitolo 9 (disponibile come supplemento Web) descrive come il C viene utilizzato per programmare un calcolatore Raspberry Pi con processore ARM, utilizzabile in sistemi embedded. I microcontrollori sono generalmente programmati in C perché questo linguaggio fornisce praticamente le stesse possibilità di controllo a basso livello dell'hardware fornite dal linguaggio assembly, ma è molto più sintetico e rapido da scrivere.