

Prezzo Euro 24,50  
ISBN: 88 386 0816-4  
giugno 1999  
420 pagine

Il linguaggio C, creato nel 1972 presso i Bell Laboratories, è attualmente il linguaggio di programmazione più utilizzato nei corsi universitari di introduzione all'informatica, grazie alla sua diffusione in ambito professionale, allo stretto legame con il mondo UNIX e alla facilità con cui consente il passaggio ad altri linguaggi quali C++, Java e Perl. In questo libro si illustrano dapprima in termini semplici e gradualità, e con il continuo ausilio di esempi svolti, la sintassi e la semantica del linguaggio, per poi affrontare in profondità argomenti più avanzati, quali ricorsioni, archivi e strutture dati (pile, code, alberi e grafi); il capitolo conclusivo dedicato all'architettura Internet, introduce il programmatore C all'utilizzo del protocollo HTTP per la realizzazione di pagine Web dinamiche.

## Indice

- 1) Avvio
  - 2) Istruzioni decisionali
  - 3) Istruzioni iterative
  - 4) Array
  - 5) Ricerche, ordinamenti e fusioni
  - 6) Stringhe
  - 7) Funzioni
  - 8) Il processore C
  - 9) Puntatori
  - 10) Ricorsioni
  - 11) Tipi
  - 12) Tipi derivati e classi di memoria
  - 13) File
  - 14) Strutture dati
  - 15) Alberi e grafi
  - 16) Programmare un Web Server
- Appendici

I primi problemi da risolvere sono concettualmente semplici; il loro scopo è quello di prendere confidenza con i costrutti sintattici del linguaggio. I capitoli dall'uno al nove costituiscono le basi del C; inizialmente vi vengono introdotti: istruzioni, tipi dati, variabili e costanti, operatori, espressioni, strutture decisionali e iterative, istruzioni composte e annidate, librerie di funzioni. Successivamente vengono presentati gli array, le stringhe e vengono analizzati tutti i principali programmi di ricerca, ordinamento e fusione. Particolare cura viene infine impiegata nella trattazione della programmazione modulare: funzioni, passaggio di parametri, visibilità delle variabili. Ai puntatori, notoriamente ostici, è dedicato un intero ampio capitolo in cui vengono trattati con dovizia di esempi.

Nella seconda metà del testo si propongono allo studente temi e problemi più complessi che richiedono un'analisi preliminare, per valutare soluzioni alternative. Il capitolo dieci, dedicato alla ricorsione, presenta numerosi interessanti esempi di calcolo combinatorio (fattoriale, disposizioni, combinazioni), mentre i due capitoli successivi riprendono e approfondiscono tipi dati, costanti, operatori ed espressioni. In particolare vi vengono trattate le operazioni bit a bit tanto importanti per la programmazione di basso livello per la gestione di dispositivi hardware per cui il C è spesso utilizzato. Nel capitolo tredici, che si occupa di archivi, viene data una nuova soluzione al problema della "gestione di un'anagrafica", già precedentemente affrontato nel capitolo sui tipi dati derivati, questa volta utilizzando i file. Nei capitoli quattordici e quindici vengono trattate le strutture dati come pile, code, alberi e grafi, presentando differenti implementazioni che coinvolgono array, liste lineari, liste multiple e soluzioni miste. Vengono presi in esame problemi come la creazione, l'inserzione, l'eliminazione di elementi. In tal modo si raggiunge un duplice scopo. Da una parte offrire un valido banco di prova per il programmatore C, spingendolo a sfruttare caratteristiche tipiche del linguaggio come puntatori, strutture, funzioni di allocazione di aree memoria e complesse chiamate di funzioni. Dall'altro costruire una introduzione completa alle strutture dati che spesso vengono studiate solo sul piano teorico nei corsi di informatica. L'ultimo capitolo, dedicato all'architettura Internet, introduce il programmatore C all'utilizzo del protocollo HTTP per la realizzazione di pagine Web statiche e dinamiche.

Alcuni temi ed esemplificazioni percorrono l'intero testo, per esempio del problema della "gestione di una sequenza" viene data una prima soluzione nel capitolo sette con l'uso di funzioni, una seconda nel capitolo nove arricchita dalla disponibilità dei puntatori, infine una terza nel capitolo quattordici utilizzando una lista lineare. Ogni capitolo è concluso dalla presentazione di numerosi esercizi per la verifica del corretto apprendimento degli argomenti. Le soluzioni degli esercizi sono accompagnate da riflessioni e dal confronto di più alternative. La versione del linguaggio a

tipiche del linguaggio macchina: si può, ad esempio, indirizzare la memoria in modo assoluto, funzionalità fondamentale per lo sviluppo di applicazioni di basso livello. E' un linguaggio apparentemente povero: non possiede istruzioni di entrata/uscita, né istruzioni per operazioni matematiche. Ogni funzione diversa dai costrutti di controllo o dalle operazioni elementari sui tipi dati è affidata ad un insieme di librerie esterne. In questo modo, Ritchie riuscì a raggiungere due obiettivi: da una parte, mantenere compatto il linguaggio, dall'altra, poter estenderne le funzionalità semplicemente aggiungendo nuove librerie o ampliando quelle esistenti. E' stato talvolta definito come "il linguaggio di più basso livello tra i linguaggi di alto livello". Infatti, come abbiamo detto, nasce per lo sviluppo di sistemi operativi, quindi per software di basso livello, ma preservando la semplicità d'uso dei linguaggi della terza generazione.

Sono molti i fattori che hanno determinato la sua capillare diffusione. Il trampolino di lancio è stato il sistema operativo Unix. Il C ne ha seguito le sorti fin dall'inizio divenendo ben presto il linguaggio di programmazione preferito dalle università e dagli istituti di ricerca. Unix è stata la dimostrazione pratica della bontà e forza del linguaggio C. Il mondo dell'industria informatica lo ha notato ed oggi praticamente non esiste prodotto commerciale di larga diffusione - database, wordprocessor, foglio elettronico, browser etc. - che non sia scritto in C. Un'altro fattore che ha contribuito al successo del C è stato il personal computer. Quelle che erano funzioni di programmazione di sistema fino a qualche anno fa riservate a pochi specialisti oggi sono accessibili a tutti. Ad esempio, oggi è molto facile, anche per un programmatore C dilettante, pilotare direttamente l'hardware. In pratica il C sta sostituendo l'assembler. Esistono, infatti, dei compilatori talmente evoluti, che il codice assembler equivalente al C prodotto dal compilatore è talmente efficiente e compatto da risultare migliore di quello scritto anche da un buon programmatore assembler. I nuovi linguaggi che si sono presentati sulla scena dell'informatica, quali Java e Perl, devono molto al C, la cui conoscenza costituisce un ottimo punto di partenza per il loro apprendimento. L'enorme diffusione raggiunta e la sua efficienza, anche nel pilotare direttamente l'hardware, continuano a fare del C una scelta largamente condivisa anche per realizzare applicazioni Internet o parti di esse. Infine esistono anche motivi estetici. Per l'eleganza della

```
printf(" tetti");  
printf(" aguzzi");  
}
```

### Listato 1.1 Un programma in linguaggio C

L'istruzione `printf` permette la stampa su video di ciò che è racchiuso tra parentesi tonde e doppi apici. Per esempio

```
printf("Tre");
```

visualizza:

Tre

Per poter utilizzare `printf`, così come le altre funzioni di input/output, si deve inserire all'inizio del testo la linea

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

che avverte il compilatore di includere i riferimenti alla libreria standard di input/output (`stdio` sta appunto per *standard input/output*). ■ Le istruzioni vengono eseguite una dopo l'altra nell'ordine in cui si presentano. Il programma del Listato 1.1 è composto da tutte istruzioni `printf` e la sua esecuzione visualizzerà la frase

Tre casettine dai tetti aguzzi

Le istruzioni `printf` successive alla prima iniziano a scrivere a partire dalla posizione del video che segue quella occupata dall'ultimo carattere visualizzato dalla `printf` immediatamente precedente. Abbiamo perciò inserito all'interno degli apici uno spazio bianco iniziale; se non lo avessimo fatto avremmo ottenuto:

Trecasettinedaitettiaguzzi

Il C distingue tra lettere maiuscole e minuscole; dunque si deve fare attenzione: se per esempio si scrive `MAIN()` o `Main()`, non si sta facendo riferimento a `main()`.

```
base = 3;
altezza = 7;
area = base*altezza;

printf("%d\n", area);
}
```

### Listato 1.3 Uso di variabili

Per rendere evidente la funzione espletata dal programma abbiamo inserito un commento:

```
/* Calcolo area rettangolo */
```

I commenti possono estendersi su più linee e apparire in qualsiasi parte del programma; devono essere preceduti da `/*` e seguiti da `*/`: tutto ciò che appare nelle zone così racchiuse non viene preso in considerazione dal compilatore e non ha nessuna influenza sul funzionamento del programma. Un altro modo per inserire un commento è farlo precedere da `//`, ma in questo caso deve terminare a fine linea:

```
// Calcolo area rettangolo
```

Dopo il `main()` e la parentesi graffa aperta sono presenti le *dichiarazioni* delle variabili (interi) necessarie:

```
int base;
int altezza;
int area;
```

La parola chiave `int` specifica che l'identificatore che lo segue si riferisce a una variabile numerica di tipo intero;

```

{
    dichiarazioni di variabili
    istruzione 1
    istruzione 2
    istruzione 3
    ...
    istruzione N
}

```

Si tenga presente che nella sintassi il punto e virgola fa parte dell'istruzione stessa. ■

Le dichiarazioni delle variabili dello stesso tipo possono essere scritte in sequenza separate da una virgola; per esempio, nel Listato 1.3 avremmo potuto scrivere:

```
int    base, altezza, area;
```

Dopo la dichiarazione di tipo sono specificati gli *identificatori* di variabile, che possono essere in numero qualsiasi, separati da virgola e chiusi da un punto e virgola. In generale, quindi, la dichiarazione di variabili ha la forma:

```
tipo    lista di identificatori;
```

Esistono inoltre regole da rispettare nella costruzione degli identificatori, che devono iniziare con una lettera o con un carattere di sottolineatura `_` e possono contenere lettere, cifre e `_`. La lunghezza può essere qualsiasi ma caratteri significativi sono spesso i primi 255 (247 secondo lo standard), anche se nelle versioni del C meno recenti questo limite scende a 32 o anche a 8 caratteri. Le lettere maiuscole sono considerate diverse dalle corrispondenti minuscole. Esempi di identificatori validi sono: `nome1`, `cognome2`, `cognome_nome`, `alberoBinario`, `volume`, `VOLUME`, `a`, `b`, `c`, `x`, `y`; al contrario non sono corretti: `12nome`, `cognome-nome`, `vero?` e `padre&figli`. Teniamo a ribadire che `volume` e `VOLUME` sono differenti. Oltre a rispettare le regole precedentemente enunciate, un identificatore non può essere una parola chiave del linguaggio (vedi Appendice B per l'elenco delle parole chiave), né può essere uguale a un nome di funzione.

Allo scopo di rendere più chiaro il risultato dell'esempio precedente, si possono visualizzare i valori delle variabili

Altezza: 7  
Area: 21

Mentre `int` è una parola chiave del C e fa parte integrante del linguaggio, `base`, `altezza` e `area` sono identificatori di variabili scelti a nostra discrezione. Lo stesso effetto avremmo ottenuto utilizzando al loro posto altri nomi generici, quali `x`, `y` e `z`.

La forma grafica data al programma è del tutto opzionale; una volta rispettata la sequenzialità e la sintassi, la scrittura del codice è libera. In particolare, più istruzioni possono essere scritte sulla stessa linea, come nell'esempio seguente:

```
#include <stdio.h>
main() {int x,y,z; x = 3; y = 7;
z= x*y; printf("Base: %d\n", x); printf("Altezza: %d\n", y);
printf("Area: %d\n", z);}
```

Questo programma, però, è notevolmente meno leggibile del precedente.

### ✓ NOTA

Lo stile facilita il riconoscimento delle varie unità di programma e riduce il tempo per modificare, ampliare e correggere gli errori. Se ciò è vero in generale, lo è particolarmente per questo linguaggio poiché, come si avrà modo di vedere, il C spinge il programmatore alla sintesi, all'utilizzo di costrutti estremamente asciutti, essenziali. Non importa quale stile si decida di utilizzare, importante è seguirlo con coerenza.

In generale è bene dare alle variabili nomi significativi, in modo che si possa facilmente ricostruire l'uso che si è fatto di una certa variabile, qualora si debba intervenire a distanza di tempo sullo stesso programma.



un'espressione a patto che su di essa non venga mai effettuato un assegnamento. Il programma del paragrafo precedente potrebbe quindi essere trasformato in quello del Listato 1.4.

```
/* Calcolo area rettangolo, prova utilizzo costanti */  
  
#include <stdio.h>  
  
#define BASE 3  
#define ALTEZZA 7  
  
main()  
{  
    int area;  
  
    area = BASE * ALTEZZA;  
    printf("Base: %d\n", BASE);  
    printf("Altezza: %d\n", ALTEZZA);  
    printf("Area: %d\n", area);  
}
```

#### Listato 1.4 Uso di costanti

Un nome di costante può essere un qualsiasi identificatore valido in C. Abbiamo scelto di utilizzare esclusivamente caratteri maiuscoli per le costanti e caratteri minuscoli per le variabili per distinguere chiaramente le une dalle altre. Le costanti `BASE` e `ALTEZZA` vengono considerate di tipo intero in quanto il loro valore è costituito da numeri senza componente frazionaria.

Invece di utilizzare direttamente i valori, è consigliabile fare uso degli identificatori di costante, che sono descrittivi

Valore della base:

In questo istante l'istruzione `scanf` attende l'immissione di un valore; se l'utente digita 15 seguito da Invio:

Valore della base: **15**<Invio>

questo dato verrà assegnato alla variabile `base` ■. Analogamente, possiamo modificare il programma per l'immissione dell'altezza e magari aggiungere un'intestazione che spieghi all'utente cosa fa il programma, come nel Listato 1.5.

```
/* Calcolo area rettangolo */  
  
#include <stdio.h>  
  
int base, altezza, area;  
  
main()  
{  
    printf("AREA RETTANGOLO\n\n");  
  
    printf("Valore base: ");  
    scanf("%d", &base);  
    printf("Valore altezza: ");  
    scanf("%d", &altezza);  
  
    area = base*altezza;  
  
    printf("Base: %d\n", base);  
    printf("Altezza: %d\n", altezza);  
    printf("Area: %d\n", area);  
}
```

Si possono stampare più variabili con una sola `printf`, indicando prima tra doppi apici i formati in cui si desiderano le visualizzazioni e successivamente i nomi delle rispettive variabili. L'istruzione

```
printf("%d %d %d", base, altezza, area);
```

inserita alla fine del programma precedente stamperebbe, se i dati immessi dall'utente fossero ancora 10 e 13:

```
10 13 130
```

Nell'istruzione il primo `%d` specifica il formato della variabile `base`, il secondo `%d` quello di `altezza` e il terzo quello di `area`. Per raffinare ulteriormente l'uscita di `printf`, si possono naturalmente inserire degli a-capo a piacere:

```
printf("%d\n%d\n%d", base, altezza, area);
```

che hanno come effetto

```
10
13
130
```

Se, per esempio, si desidera una linea vuota tra il valore della variabile `base` e quello di `altezza` e due linee vuote prima del valore della variabile `area`, è sufficiente inserire i `\n` nella descrizione dei formati, esattamente dove si vuole che avvenga il salto a riga nuova:

```
printf("%d\n\n%d\n\n%d", base, altezza, area);
```

All'interno dei doppi apici si possono scrivere i commenti che devono essere stampati. Per esempio, se la visualizzazione della terza variabile deve essere preceduta da `Area:`, l'istruzione diventa la seguente:

```
printf("%d\n%d\nArea: %d", base, altezza, area);
```

che darà in uscita

E possiamo inserire nella printf, al posto delle variabili, delle espressioni, di tipo specificato dal formato:

```
printf("Area: %d", 10*13);
```

Il %d ci indica che il risultato dell'espressione è di tipo intero; l'istruzione stamperà 130. Un'espressione può naturalmente contenere delle variabili:

```
printf("Area: %d", base*altezza);
```

Si può definire all'interno di una istruzione printf anche il numero di caratteri riservati per la visualizzazione di un valore, nel seguente modo:

```
printf("%5d%5d%5d", base, altezza, area);
```

Il %5d indica che verrà riservato un campo di cinque caratteri per la visualizzazione del corrispondente valore, che sarà sistemato a cominciare dall'estrema destra di ogni campo:



Se vengono inseriti degli spazi o altri caratteri nel formato, oltre alle descrizioni %5d, essi appariranno nelle posizioni corrispondenti. Inserendo poi un carattere - dopo il simbolo di percentuale e prima della lunghezza del campo il valore viene sistemato a cominciare dall'estrema sinistra della maschera. L'istruzione ■

```
printf("%-5d%-5d%5d", base, altezza, area);
```

assegna a `w` il valore assoluto di `j`. All'interno delle parentesi tonde può essere inserito direttamente un valore, come nel caso

```
w = abs(3);
```

che assegna a `w` il valore 3, o come nel caso

```
w = abs(-186);
```

che assegna a `w` il valore 186. In questo contesto, ribadiamo, la nostra attenzione non è rivolta al modo in cui viene svolto un certo compito ma a cosa immettere come argomento della funzione predefinita per ottenere un certo risultato. Naturalmente in qualche luogo è (pre)definito l'insieme di istruzioni che la compongono; nel caso della funzione `abs()` tale luogo è la libreria standard `math.h`. Perciò, per poter utilizzare tale funzione si deve dichiarare esplicitamente nel programma, prima del `main`, l'inclusione del riferimento a tale libreria.

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

Osserviamo nel Listato 1.6 un programma completo che utilizza la funzione `abs()`. Esso permette di risolvere il problema del calcolo della lunghezza di un segmento, i cui estremi vengono immessi dall'utente. Se consideriamo la retta dei numeri interi, ognuno dei due estremi può essere sia positivo sia negativo, per cui la lunghezza del segmento è pari al valore assoluto della differenza tra i due valori.

```
/* Esempio utilizzo di abs() */

#include <stdio.h>
#include <math.h>

main()
{
    int a, b, segmento, lunghezza;
```

un mezzo del programma precedente, così come degli altri, abbiamo incluso il riferimento a tale libreria. Dal punto di vista del programmatore, quello che interessa per ottenere un certo risultato è sapere:

1. che esiste la funzione corrispondente;
2. di quali informazioni essa ha bisogno;
3. in quale libreria è contenuta.

Le funzioni standard sono catalogate rispetto all'applicazione cui sono dedicate; per esempio:

<code>stdio.h</code>	funzioni di input/output
<code>math.h</code>	funzioni matematiche
<code>string.h</code>	funzioni che operano su stringhe

Esistono molte librerie, ognuna delle quali contiene un certo numero di funzioni. Il programmatore può creare delle proprie funzioni ed eventualmente inserirle in file che diventano le sue librerie personali; quando lo desidera può includere nel programma tali librerie così come fa con quelle standard. ■

## 1.6 Fasi di programmazione

Qualsiasi versione del C si abbia a disposizione, e qualsiasi sistema operativo si impieghi, le fasi del lavoro del programmatore sono costituite da:

- editing del programma;
- precompilazione;

```
Valore base: 10
Valore altezza: 13
Base: 10
Altezza: 13
Area: 130
$
```

Se successivamente viene eseguita la compilazione di un altro programma, il nuovo codice oggetto rimpiazzerà il primo in `a.out`, per cui è bene ogni volta effettuare una copia di `a.out` su un diverso eseguibile. Il comando `cc` ha moltissime opzioni; una di esse, `-o`, permette di specificare direttamente il nome del file oggetto:

```
$ cc rettang.c -o rettang.exe
```

È poi possibile effettuare il link separatamente, il che consente, come vedremo in seguito, una notevole flessibilità nella programmazione.

Se si è in un ambiente dedicato, per sviluppare le varie fasi naturalmente basterà scegliere le opzioni relative dai menu messi a disposizione. Per esempio, con “Microsoft Visual C++”, una volta editato il programma possiamo scegliere in sequenza le opzioni `Compile`, `Build` ed `Execute` dal menu `Build`.

## 1.7 Esercizi

1. Predisporre un programma che, utilizzando una sola istruzione `printf`, visualizzi:

```
c = a*b;  
b = 3;
```

8. Indicare tutti gli errori commessi nel seguente listato.

```
#include <stdio.h>  
  
/* Tutto Sbagliato!!! */  
  
#define BASE 3  
#define ALTEZZA  
  
main()  
    area int;  
  
    area = BASE x ALTEZZA;  
    printf("Base: d\n", BASE);  
    printf("Altezza: %d\n", ALTEZZA)  
    printf("Area: %d\n"; area);  
}
```

## 2.1 L'istruzione

Quando si desidera eseguire un'istruzione al verificarsi di una certa condizione, si utilizza l'istruzione `if`. Per esempio, se si vuole visualizzare il messaggio minore di 100 solamente nel caso in cui il valore della variabile intera `i` è minore di 100, si scrive



aritmetiche e logiche e vedremo come si integrano le une nelle altre.

La sintassi completa dell'istruzione `if` è:

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

dove la valutazione di *espressione* controlla l'esecuzione di *istruzione1* e quella di *istruzione2*: se *espressione* è vera viene eseguita *istruzione1*, se è falsa viene eseguita *istruzione2* ■.

Nell'esempio precedente è stato omesso il ramo `else`; il fatto è del tutto legittimo poiché tale ramo è opzionale, come evidenziato dalle parentesi quadre della forma sintattica completa e come vedremo anche nel prossimo paragrafo. Modifichiamo ora l'esempio in modo da visualizzare un messaggio anche quando la variabile `i` non è minore di 100 (Listato 2.1).

```
/* Utilizzo if-else */

#include <stdio.h>

main()
{
    int i;

    printf("Dammi un intero: ");
    scanf("%d", &i);

    if(i<100)
        printf("minore di 100\n");
    else
        printf("maggiore o uguale a 100\n");
}
```

```
    printf("minore di 100 ");  
else  
    printf("maggiore o uguale a 100 ");  
printf("istruzione successiva alla clausola else");
```

l'ultima `printf` viene eseguita indipendentemente dal fatto che `i` risulti minore oppure maggiore o uguale a 100.

Supponiamo ora di voler aggiungere nel Listato 2.1 un'ulteriore istruzione, oltre alla `printf`, ai due rami del costruito `if-else`. Inizializziamo due variabili intere, `mag_100` e `min_100`, a zero. Nel caso in cui `i` risulti essere minore di 100, assegniamo il valore 1 a `min_100`, altrimenti lo assegniamo a `mag_100`. In termini tecnici diciamo che alziamo il *flag* `mag_100` o il *flag* `min_100` in base al risultato del controllo effettuato dall'istruzione `if` (vedi Listato 2.2).

```
/* Esempio istruzioni composte */  
  
#include <stdio.h>  
  
int i;  
int mag_100;  
int min_100;  
  
main()  
{  
    mag_100 = 0;  
    min_100 = 0;  
  
    printf("Dammi un intero: ");  
    scanf("%d", &i);  
  
    if(i<100) {  
        min_100 = 1;  
    }  
    else if(i>=100) {  
        mag_100 = 1;  
    }  
}
```

*Istruzione semplice*

```
printf("minore di 100\n");
```

```
printf("maggiore o uguale  
a 100\n");
```

*Istruzione composta*

```
{  
    printf("minore di 100\n");  
    min_100 = 1;  
}
```

```
{  
    printf("maggiore o uguale a 100\n");  
    mag_100 = 1;  
}
```

Se non si fossero utilizzate le parentesi graffe il significato del programma sarebbe stato ben diverso:

```
if(i<100)  
    printf("minore di 100\n");  
    min_100 = 1;  
else  
    printf("maggiore o uguale a 100\n");  
    mag_100 = 1;
```

Ciò che può fuorviare è l'aspetto grafico del codice, ma dato che l'indentazione non viene considerata la compilazione rivelerà un errore trovando un'istruzione `else` spuria, cioè non ricollegabile a un'istruzione `if`. Non essendo stata aperta la parentesi graffa, l'`if` regge la sola istruzione `printf("è minore di 100\n")`, dopo di che, se non trova la clausola `else`, si chiude.

Un'istruzione composta può essere immessa nel programma dovunque possa comparire un'istruzione semplice e quindi indipendentemente dal costrutto `if-else`; al suo interno, dopo la parentesi graffa aperta e prima delle altre istruzioni, possono essere inserite delle dichiarazioni di *variabili locali* a quel blocco, variabili – cioè – che possono essere utilizzate fino alla chiusura del blocco stesso:

else

```
printf("minore di 100 ma non maggiore di zero");
```

Come fa capire il messaggio della seconda `printf`, l'`else` che abbiamo aggiunto si riferisce al secondo `if`, cioè a quello più interno, il quale insieme alle due `printf` e all'`else` costituiscono ancora un'unica istruzione. Per fare in modo che l'`else` si riferisca al primo `if` bisogna usare le parentesi graffe:

```
if(i<100) {  
    if(i>0)  
        printf("minore di 100 e maggiore di zero");  
}  
else  
    printf("maggiore o uguale a 100");
```

Se invece avessimo anche l'`else` dell'`if` più interno le parentesi graffe sarebbero superflue, e si potrebbe scrivere:

```
if(i<100)  
    if(i>0)  
        printf("minore di 100 e maggiore di zero");  
    else  
        printf("minore di 100 ma non maggiore di zero");  
else  
    printf("maggiore o uguale a 100");
```

Ciò è reso possibile dal fatto che il tutto viene considerato come un'unica istruzione. Modifichiamo ora il nostro esempio in modo che vengano visualizzati due messaggi diversi a seconda che la variabile `i` sia maggiore oppure uguale a 100, e non soltanto un messaggio generico come sopra:

```
if(i<100)  
    if(i>0)  
        printf("minore di 100 e maggiore di zero");
```

- le righe 2..8 sono un'unica istruzione if-else la quale ha per *istruzione1* la riga 3 e per *istruzione2* le righe 5..8;
- le righe 5..8 sono un'unica istruzione if-else la quale ha per *istruzione1* la riga 6 e per *istruzione2* la riga 8;
- le righe 10..13 sono un'unica istruzione if-else la quale ha per *istruzione1* la riga 11 e per *istruzione2* la riga 13.

### ✓ NOTA

Quanto sopra può essere scritto in modo più compatto:

```
if(i<100)
    if(i>0)
        printf("minore di 100 e maggiore di zero");
    else if(i==0)
        printf("uguale a zero");
    else
        printf("minore di zero");
else if(i==100)
    printf("uguale a 100");
else
    printf("maggiore di 100");
```

Questo schema è frequente nei programmi C ed è molto comodo per simulare l'istruzione *elseif*, tipica di altri linguaggi.

Figura 2.1 Gerarchia degli operatori aritmetici e di assegnamento

Quello di negazione è l'unico operatore unario, cioè che si applica a un solo operando. Se  $x$  ha valore 5, l'espressione

$-x$ ;

restituisce  $-5$ , mentre

$2 * -(x-6)$ ;

restituisce 2. Si noti che mentre il  $-$  anteposto alla parentesi tonda aperta corrisponde all'operatore unario di negazione, l'altro rappresenta l'operatore binario di sottrazione.

L'operatore di modulo,  $\%$ , consente di ottenere il resto della divisione intera tra l'operando che lo precede e quello che lo segue. Quindi, sempre nell'ipotesi che  $x$  valga 5,

$34 \% x$ ;

ha valore 4, perché  $(34 : 5 = 6 \text{ con resto } 4)$ .

All'interno delle espressioni aritmetiche, la priorità degli operatori segue le regole dell'algebra. La valutazione di una espressione contenente operazioni matematiche avviene esaminandola da sinistra a destra più volte, dunque gli operatori sono associativi da sinistra verso destra. Tutte le operazioni di negazione sono eseguite per prime, quindi l'espressione è esaminata nuovamente per eseguire tutte le moltiplicazioni, divisioni e le operazioni modulo. Infine l'espressione viene sottoposta a scansione ancora una volta per eseguire le addizioni e le sottrazioni. La priorità degli operatori può essere alterata mediante le parentesi tonde: vengono valutate per prime le operazioni all'interno delle parentesi tonde più interne.

Osserviamo le seguenti espressioni, nell'ipotesi che le variabili intere  $a$ ,  $b$  e  $c$  abbiano rispettivamente valore: 7, 3 e 5. Il risultato di

dove il valore restituito da  $z^2/x$  viene assegnato a  $x$ ,  $y$  e  $t$ .

## 2.4.2 Espressioni logiche

Un'espressione logica è un'espressione che genera come risultato un valore vero o falso (abbiamo visto che in C non esiste il tipo booleano, presente in altri linguaggi), e viene utilizzata dalle istruzioni di controllo. Le espressioni logiche, per la precisione, producono come risultato 1 per vero e 0 per falso (qualsiasi valore numerico diverso da zero viene comunque considerato vero). Un semplice esempio di espressione logica è una variabile il cui contenuto può essere interpretato in due modi: vero se diverso da zero, falso se uguale a zero.

Le espressioni logiche possono contenere gli *operatori relazionali*, usati per confrontare fra loro dei valori, riportati in Figura 2.2.

> (maggiore di)	>= (maggiore uguale)
< (minore di)	<= (minore uguale)
== (uguaglianza) != (disuguaglianza)	

Figura 2.2 Gerarchia degli operatori relazionali

Si noti come l'operatore di uguaglianza `==` sia diverso anche nella notazione da quello di assegnamento `=`, fatto a nostro avviso positivo e non comune negli altri linguaggi di programmazione.

La priorità di `>`, `>=`, `<`, e `<=` è la stessa ed è maggiore di `==` e `!=`. Dunque in

`x > y == z > t`

viene valutato prima `x > y` e `z > t` e successivamente verificata l'uguaglianza tra i due risultati come se l'espressione fosse: `(x > y) == (z > t)`.

restituisce falso solamente se  $b$  non è minore di  $c$  e contemporaneamente  $t$  è uguale a  $r$ .

L'espressione

$!y$

restituisce vero se  $y$  è falso e viceversa. L'operatore  $!$  è di tipo unario, gli altri sono binari.

L'ordine di priorità complessiva degli operatori logici e relazionali è mostrato in Figura 2.4. Naturalmente, si possono utilizzare le parentesi tonde per alterare la priorità; per esempio, in

```
(nome1!=nome2 || cognome1>cognome2) && presenti < 524  
    viene valutato prima || di &&.
```

Il risultato di un'espressione logica può essere assegnato a una variabile, come abbiamo già visto tra gli esempi del primo paragrafo:

```
a = i<100
```

Dalla tabella gerarchica degli operatori deduciamo che l'assegnamento è quello con minor priorità; è per questa ragione che viene valutato innanzitutto  $i<100$ ; se  $i$  risulta minore di 100  $a$  assume valore 1, altrimenti assume valore 0. L'aggiunta di un punto e virgola trasforma l'espressione in una istruzione:

```
a = i<100;
```

Esaminiamo ora un altro esempio:

```
y = a==b;
```

dove  $y$  assume valore 1 se  $a$  è uguale a  $b$ , 0 viceversa. Dunque è lecito anche questo assegnamento, la cui interpretazione lasciamo al lettore:

```
x = (x==y) && (z!=t || m>=n);
```



Un'espressione può contenere una combinazione di operatori aritmetici, logici e relazionali. L'espressione

```
x + sereno + (i<100)
```

restituisce la somma di `x`, di `sereno` e del risultato di `i<100`, che è 1 se `i` è minore di 100, zero altrimenti. Per esempio, se `x` vale 5, `sereno` 6 e `i` 98, l'espressione restituisce 12.

È un'espressione anche la seguente:

```
y = x + sereno + (i<100)
```

dove l'assegnamento a `y` viene effettuato alla fine, perché `=` ha la priorità più bassa. Essendo l'operatore di assegnamento trattato alla stregua degli altri, sarà lecita anche la seguente espressione:

```
i>n && (x=y)
```

che mette in AND le espressioni `i>n` e `x=y`. La prima è vera se il valore di `i` è maggiore di `n`. La seconda corrisponde all'assegnamento del valore di `x` alla variabile `y`; tale assegnamento viene effettuato all'interno dell'espressione, dopo di che se il valore di `x` è diverso da zero l'espressione risulta vera, altrimenti falsa.

### ✓ NOTA

Queste caratteristiche rendono il C un linguaggio flessibile che consente la scrittura di codice sintetico ed efficiente, ma anche difficilmente interpretabile. Prendiamo ad esempio il seguente frammento di codice:

```
if((x=y) && i>n)
    printf("vero");
else
    printf("falso");
```

### 2.4.3 Espressioni condizionali

Una *espressione condizionale* si ottiene con l'operatore ternario `?:` che ha la seguente sintassi:

*espr1* ? *espr2* : *espr3*

Se la valutazione di *espr1* restituisce vero, il risultato è uguale a *espr2*, altrimenti è uguale a *espr3*. Per esempio,

`x==y ? a : b`

significa: “se *x* è uguale a *y*, allora *a*, altrimenti *b*”.

Si può utilizzare l'operatore `?:` per assegnare un valore a una variabile, come nel caso che segue:

`v=x==y ? a*c+5 : b-d;`

Se *x* è uguale a *y*, a *v* viene assegnato il valore di *a\*c+5*, altrimenti gli viene assegnato il valore di *b-d*. Le espressioni *espr1*, *espr2* ed *espr3* vengono valutate prima di `?:` e l'assegnamento viene effettuato dopo, data la posizione dell'operatore condizionale nella tavola gerarchica.

L'espressione condizionale può essere sempre sostituita da un `if` corrispondente: nel caso precedente, per esempio, avremmo potuto scrivere:

```
if (x==y)
    v = a*c+5;
else
    v = b-d;
```

Essendo quella condizionale un'espressione come tutte le altre, può essere inserita in qualsiasi posizione sia appunto lecito scrivere un'espressione. Potremmo cioè scrivere:

```
printf("nero");  
else  
    printf("bianco");
```

In generale, quando più operatori con la stessa priorità devono essere valutati in un'espressione, lo standard C non garantisce l'ordine di valutazione. Nell'espressione

`x > y <= z`

non possiamo sapere se verrà valutata prima la condizione `x > y` e successivamente se il risultato (zero o uno) è minore o uguale a `z`, o viceversa. In ogni caso è buona norma non scrivere codice dipendente dall'ordine di valutazione per non rischiare di produrre programmi non portabili tra versioni C differenti.

Esiste comunque un'eccezione. Per gli operatori `&&`, `||`, e `?:` il linguaggio garantisce la valutazione delle espressioni da sinistra verso destra. Per esempio, in

`x==y && a>b`

l'espressione `a>b` sarà valutata soltanto se il risultato dell'espressione `x==y` è vero. Analogamente in

`b<c || t!=r`

l'espressione `t!=r` sarà valutata soltanto se il risultato dell'espressione `b<c` è falso.

Nell'espressione

`x>y ? a>b ? a : b : y`

prima viene valutata `a>b`, quindi restituito `a` oppure `b`; successivamente è valutata `x>y`, dato che `?:` è associativo da destra verso sinistra.

```
y = ' , ' ;  
z = ' & ' ;
```

Come si può facilmente dedurre dal codice ASCII riportato in Appendice D ■, si hanno le seguenti corrispondenze.

Carattere	Decimale	Esadecimale	Binario
A	65	41	01000001
;	59	3B	00111011
&	38	26	00100110

A ogni carattere presente nel codice corrisponde una rappresentazione numerica univoca, per cui è possibile confrontare due simboli non solamente con uguaglianze e disuguaglianze, ma anche per mezzo di tutti gli altri operatori relazionali.

“A” (65) è maggiore di “;” (59) che a sua volta è maggiore di “&” (38). Osservando il codice ASCII possiamo vedere che le lettere alfabetiche maiuscole sono in ordine crescente da A (65) a Z (90), le minuscole vanno da a (98) a z (122) e le cifre decimali da 0 (48) a 9 (57) ■. Dunque ha perfettamente senso l’istruzione condizionale

```
if(x=='A')  
    printf("Si tratta di una A");
```

ma anche

```
if(x>='A' && x<='Z')  
    printf("Si tratta di una lettera maiuscola");
```

Per poter visualizzare dei char con una printf si deve come al solito indicarne il formato; per esempio:

```
printf("%c %c %c", x, y, z);
```

```
x = getchar();
```

bloccherà il programma in attesa di un carattere introdotto da tastiera. Si noti che la presenza delle parentesi tonde è necessaria anche se dentro non vi è racchiuso alcun argomento.

Per visualizzare un carattere abbiamo invece la funzione

```
putchar(x);
```

Vediamo una semplice applicazione di queste due funzioni. I due programmi del Listato 2.3 hanno funzionamenti identici: i caratteri dovranno essere digitati uno dietro l'altro e successivamente dovrà essere battuto un Invio.

### ✓ NOTA

Se il programma dovesse prevedere l'immissione di più valori in tempi diversi, l'inserimento di un carattere potrebbe costituire un problema, dato che la digitazione del tasto di Invio da tastiera corrisponde a un carattere accettato da `scanf("%c", ...)`. In tali casi verrà utilizzata un'opportuna ulteriore lettura di un carattere in una variabile ausiliaria tramite un'istruzione del tipo `scanf("%c", pausa) o pausa=getchar()`.

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

```
main()
```

```
{
```

```
char x, y, z;
```

```
printf("digita tre carat.: ");
```

```
scanf("%c%c%c", &x, &y, &z);
```

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

```
main()
```

```
{
```

```
char x, y, z;
```

```
printf("digita tre carat.: ");
```

```
x = getchar();
```

```
y = getchar();
```

```
z = getchar();
```

```

switch(espressione) {
    case costante1:
        istruzione
        ...
    case costante2:
        istruzione
        ...
    case costante3:
        istruzione
        ...
    [default:
        istruzione
        ...      ]
}

```

```

switch(espressione) {
    case costante1:
        istruzione
        ...
        break;
    case costante2:
        istruzione
        ...
        break;
    case costante3:
        istruzione
        ...
        break;
    ...
    [default:
        istruzione
        ...      ]
}

```

**Figura 2.5** A sinistra sintassi del costrutto switch-case; a destra forma spesso utilizzata del costrutto switch-case

La parola `switch` è seguita da una *espressione*, racchiusa tra parentesi tonde, il cui risultato deve essere di tipo `int` o `char`. Il resto del costrutto è formato da un'istruzione composta, costituita da un numero qualsiasi di sottoparti, ciascuna delle quali inizia con la parola chiave `case`, seguita da un'espressione costante intera o carattere. Questa è seguita da un'istruzione da eseguire se la condizione è vera. Il costrutto termina con una parola chiave `break` o una parentesi chiudente `}`.

```
case 1:
    printf("uno\n");
    break;
case 2:
    printf("due\n");
    break;
case 3:
    printf("tre\n");
    break;
case 4:
    printf("quattro\n");
    break;
case 5:
    printf("cinque\n");
    break;
default:
    printf("non compreso\n");
    break;
}
}
```

Listato 2.4 Esempio di diramazione multipla del flusso di esecuzione

```

case 5:
    printf("dispari\n");
    break;
default:
    printf("altro\n");
}
}

```

Listato 2.5 Più valori costanti corrispondono allo stesso gruppo di istruzioni

## Esercizi

1. Scrivere un programma che richieda in ingresso tre valori interi distinti e ne determini il maggiore.
- \* 2. Ripetere l'Esercizio 1 ma con quattro valori in ingresso.
3. Ripetere l'Esercizio 2 nell'ipotesi che i quattro valori possano anche essere tutti uguali, caso nel quale il messaggio da visualizzare dev'essere `Valori identici`.
- \* 4. Ripetere l'Esercizio 1 ma individuando anche il minore dei tre numeri in input.
- \* 5. Se le variabili intere `a`, `b` e `c` hanno rispettivamente valore 5, 35 e 7, quali valore viene assegnato alla variabile `ris` dalle seguenti espressioni?

- 1) `ris = a+b*c`
- 2) `ris = (a>b)`
- 3) `ris = (a+b) * (a<b)`
- 4) `ris = (a+b) && (a<b)`
- 5) `ris = (a+b) || (a>b)`



- • il volume del parallelepipedo di lati  $a$ ,  $b$  e  $c$  se il valore di  $a$  al quadrato sommato a  $b$  è diverso da  $c$ ;
- • la somma di  $a$ ,  $b$  e  $c$ , altrimenti.

\* 9. Scrivere un programma che visualizzi il seguente menu:

MENU DI PROVA

- a) Per immettere dati
- b) Per determinare il maggiore
- c) Per determinare il minore
- d) Per ordinare
- e) Per visualizzare

Scelta: \_

quindi attenda l'immissione di un carattere da parte dell'utente e visualizzi una scritta corrispondente alla scelta effettuata, del tipo: "In esecuzione l'opzione a". Se la scelta non è tra quelle proposte (a, b, c, d, e) deve essere visualizzata la scritta: "Opzione inesistente". Si utilizzi il costrutto switch-case e la funzione getchar.

\* 10. Ripetere l'Esercizio 1 ma utilizzando l'espressione condizionale con l'operatore "?:".

11. Scrivere un programma che, richiesto il numero MM rappresentante il valore numerico di un mese, visualizzi, se  $1 \leq MM \leq 12$ , il nome del mese per esteso, altrimenti la frase "Valore numerico non valido".

12. Scrivere un programma che, richiesto il numero AA rappresentante un anno, verifichi se questo è bisestile.  
[Suggerimento: un anno è bisestile se è divisibile per 4 ma non per 100 (cioè si escludono gli anni-secolo).]

13. Scrivere un programma che, richiedi i numeri GG, MM, AA di una data, verifichi se questa è valida.

Successivamente l'esecuzione prosegue così:

- |    |    |  |
|----|----|--|
| 1. | 1. | se $i \leq 3$ allora vai al passo 2 altrimenti termina |
| 2. | 2. | $somma = somma + 7$                                    |
| 3. | 3. | $i = i + 1$ , vai al passo 1                           |

Inizialmente la condizione risulta vera, in quanto 1 è minore o uguale a 3, e quindi viene eseguito il corpo del ciclo, che in questo caso è composto dalla sola istruzione  $somma = somma + 7$ . La variabile  $somma$  assume il valore 7. Viene incrementato di 1 il valore di  $i$ , che quindi assume il valore 2.

Alla fine del terzo ciclo la variabile  $somma$  ha il valore 21 e  $i$  vale 4. Nuovamente viene verificato se  $i$  è minore o uguale a 3. La condizione risulta falsa e l'iterazione ha termine; il controllo passa all'istruzione successiva del programma.

Il formato del costrutto `for` è il seguente:

```
for(esp1; esp2; esp3)  
  istruzione
```

Si faccia attenzione ai punti e virgola all'interno delle parentesi. Il ciclo inizia con l'esecuzione di *esp1*, la quale non verrà mai più eseguita. Quindi viene esaminata *esp2*. Se *esp2* risulta vera, viene eseguita *istruzione*, altrimenti il ciclo non viene percorso neppure una volta.

Successivamente viene eseguita *esp3* e di nuovo valutata *esp2* che se risulta essere vera dà luogo a una nuova esecuzione di *istruzione*. Il processo si ripete finché *esp3* non risulta essere falsa. Nell'esempio precedente,

```
for( $i=1$ ;  $i \leq 3$ ;  $i=i+1$ )  
   $somma = somma + 7$ ;
```

*esp1* era  $i=1$ , *esp2*  $i \leq 3$ , *esp3*  $i=i+1$  e *istruzione* era  $somma = somma + 7$ .

Nella sintassi del `for`, *istruzione*, così come nel costrutto `if`, può essere un blocco, nel qual caso deve iniziare con una parentesi graffa aperta e terminare con parentesi graffa chiusa. Supponiamo di voler ottenere la somma di tre numeri interi immessi dall'utente: si può scrivere:

```

{
printf("SOMMA 5 NUMERI\n");
somma = 0;

for(i=1; i<=5; i=i+1) {
    printf("Inser. intero: ");
    scanf("%d", &numero);
    somma = somma + numero;
}

printf("Somma: %d\n",somma);
}

```

### Listato 3.1 Iterazione con l'istruzione `for`

Durante l'esecuzione del programma l'utente sarà chiamato a introdurre cinque valori interi:

```

SOMMA 5 NUMERI
Inser. intero: 32
Inser. intero: 111
Inser. intero: 2
Inser. intero: 77
Inser. intero: 13
Somma: 235

```

In questo caso l'utente ha inserito 32, 111, 2, 77 e 13. Naturalmente il numero dei valori richiesti può variare: se si vogliono accettare 100 valori si deve modificare soltanto la condizione di fine ciclo *esp2* nel `for`:

```

for(i=1; i<=100; i=i+1)

```

infinita, come nell'esempio seguente:

```
for (i=5; i>=5; i=i+1)
```

Il valore di *i* viene inizializzato a 5; è dunque verificata la condizione *i*>=5. Successivamente *i* viene incrementato di una unità e assume di volta in volta i valori 6, 7, 8 ecc. che risulteranno essere sempre maggiori di 5: il ciclo è infinito. Il compilatore non segnalerà nessun errore ma l'esecuzione del programma probabilmente non farà ciò che si desidera.

### ✓ NOTA

Situazioni di questo genere si presentano di frequente perché non è sempre banale riconoscere un'iterazione infinita; perciò si utilizzino pure le libertà linguistiche del C, ma si abbia cura di mantenere sempre uno stile di programmazione strutturato e lineare, in modo da accorgersi rapidamente degli eventuali errori commessi.

Ognuna delle *esp1*, *esp2* ed *esp3* può essere l'espressione nulla, nel qual caso comunque si deve riportare il punto e virgola corrispondente. Vedremo nei prossimi paragrafi alcuni esempi significativi. Anche l'*istruzione* del *for* può essere nulla, corrispondere cioè a un punto e virgola, come nell'esempio:

```
for (i=1; i<1000; i=i+100)
    ;
```

che incrementa il valore *i* di 100 finché *i* risulta minore di 1000. Si osservi che al termine dell'esecuzione dell'istruzione *i* avrà valore 1001: è chiaro perché?

## 3.2 Incrementi e decrementi

L'incremento unitario del valore di una variabile è una delle operazioni più frequenti. Si ottiene con l'istruzione

```
}
```

Si noti come il codice C si faccia via via più compatto.

Gli operatori ++ e -- possono precedere una variabile in un'espressione:

```
int a, b, c;  
a = 5;  
b = 7;  
c = ++a + b;  
printf("%d \n", a);  
printf("%d \n", b);  
printf("%d \n", c);
```

Nell'espressione ++a+b, la variabile `a` viene incrementata di una unità (++a) e sommata alla variabile `b`. Successivamente il risultato viene assegnato a `c`. Le tre istruzioni `printf` visualizzeranno rispettivamente 6, 7 e 13.

Gli operatori ++ e -- hanno priorità maggiore degli operatori binari aritmetici, relazionali e logici, per cui vengono considerati prima degli altri (Figura 3.1) ■.

!	-	++	--
*	/	%	
	+	-	
>	>=	<	<=
	==	!=	
	&&		

non produce la stessa visualizzazione della sequenza precedente. La variabile *a* viene sommata a *b* e il risultato viene assegnato a *c*, successivamente *a* viene incrementata di una unità. Le istruzioni `printf` visualizzeranno rispettivamente 6, 7 e 12. Si osservi l'identità dei due cicli `for`:

```
for(i=1; i<=3; ++i)          for(i=1; i<=3; i++)
```

poiché `esp3` è da considerarsi un'istruzione a sé stante. Viceversa

```
for(i=1; ++i<=3;)           for(i=1; i++<=3;)
```

sono diversi in quanto nel caso di sinistra *i* viene incrementata prima della valutazione di `espr2`, per cui nel primo ciclo *i* acquista valore 2, nel secondo 3 e il terzo ciclo non verrà mai eseguito dato che *i* ha già valore 4. Nel caso di destra *i* assume valore 4 solamente dopo il confronto operato nel terzo ciclo, che quindi verrà portato a termine; per verificarlo si provino le successive due sequenze.

```
j=0;
for(i=1; ++i<=3;)
    printf("Ciclo: %d\n", ++j);
printf("Cicli:%d i:%d\n", j, i);
```

```
j=0;
for(i=1; i++<=3;)
    printf("Ciclo: %d\n", ++j);
printf("Cicli:%d i:%d\n", j, i);
```

Le visualizzazioni prodotte saranno rispettivamente

Ciclo:1	Ciclo:1
Ciclo:2	Ciclo:2
Cicli:2 i:4	Ciclo:3
	Cicli:3 i:5

2. 2. viene moltiplicato il valore ottenuto per il valore della variabile posta a sinistra dell'assegnamento:  
 $a * (4+b);$
3. 3. viene assegnato il risultato ottenuto alla variabile posta a sinistra dell'assegnamento:  $a = a * (4+b) .$

Questo funzionamento è coerente con la bassa priorità degli operatori  $+=$ ,  $-=$ ,  $*=$ ,  $/=$  e  $\%=$  che hanno lo stesso livello dell'assegnamento semplice  $=$  (Figura 3.1). Per esempio, dopo la sequenza di istruzioni

$a = 3;$

$b = 11;$

$c = 4;$

$c -= a*2+b;$

La variabile  $c$  ha valore  $-13$ .

```
scanf("%d", &n);

fat = n;
for(m=n; m>2; m--)
    fat = fat*(m-1);

printf("Il fattoriale di: %d ha valore: %d\n", n, fat);
}
```

### Listato 3.2 Calcolo del fattoriale di $n$

Nell'ipotesi di non considerare il caso  $n = 0$ , un algoritmo possibile è quello del Listato 3.2. Se viene passato in ingresso il valore 4, `fat` assume tale valore:

```
fat = n;
```

Il ciclo `for` inizializza 4 a `m` e controlla che sia maggiore di 2. Viene eseguito una prima volta il ciclo

```
fat = fat*(m-1);
```

e `fat` assume il valore 12. Di nuovo il controllo dell'esecuzione passa al `for` che decrementa il valore di `m` e verifica se  $m > 2$ , cioè se  $3 > 2$ . Viene eseguito il corpo del ciclo

```
fat = fat*(m-1);
```

e `fat` assume il valore 24. Il `for` decrementa `m` e verifica se  $m > 2$ , cioè se  $2 > 2$ . Questa volta l'esito è negativo e le iterazioni hanno termine. Utilizzando l'operatore `*=`, al posto di `fat=fat*(m-1)` avremmo potuto scrivere

```
fat *= m-1;
```



```
main()
{
    int n, fat, aux;

    printf("CALCOLO DI N!\n\n");
    printf("Inser. n: ");
    scanf("%d", &n);

    fat = 1;
    for(aux=2; aux<=n; aux++) fat = fat*aux;

    printf("Il fattoriale di: %d ha valore: %d\n", n, fat);
}
```

Listato 3.3 Un'altra possibilità per il calcolo del fattoriale

## 3.4 Istruzione `while`

Anche l'istruzione `while`, come l'istruzione `for`, permette di ottenere la ripetizione ciclica di una istruzione:

```
while(esp)
    istruzione
```

Viene verificato che *esp* sia vera, nel qual caso viene eseguita *istruzione*. Il ciclo si ripete fintantoché *esp* risulta essere vera. Naturalmente, ancora una volta, *istruzione* può essere un blocco. Riprendiamo il programma

caso contrario il numero di iterazioni sarebbe uguale a nove. Quando si deve ripetere il volte un ciclo la migliore soluzione è ancora un'altra:

```
i = n;
while(i-->0)
    corpo_del_ciclo
```

Come abbiamo visto nel capitolo precedente, la condizione logica diviene falsa quando `i` assume valore zero. Nell'esempio precedente si ha:

```
i = 5;
while(i-->0) {
    printf("Inser. intero: ");
    scanf("%d", &numero);
    somma = somma+numero;
}
```

Osserviamo, ancora una volta, come il codice si faccia sempre più compatto.

Trasformiamo adesso il programma in modo che la lunghezza della serie dei numeri in ingresso non sia determinata a priori ma termini quando viene inserito il valore zero. Non è possibile evidentemente risolvere il problema con una ripetizione sequenziale d'istruzioni in quanto il numero di valori non è noto, ma viene deciso a tempo d'esecuzione (Listato 3.4).

```
/* Calcola la somma dei valori interi passati dall'utente
   termina quando viene immesso il valore 0 (zero) */

#include <stdio.h>

main()
{
```

Ogni istruzione `for` può essere sostituita da un'istruzione `while` se si ha cura di aggiungere le opportune inizializzazioni prima del ciclo e gli opportuni incrementi all'interno dello stesso. In C è vero anche l'inverso. Ogni istruzione `while` ha un suo corrispondente `for`, anche quando il numero d'iterazione non è noto a priori. Per esempio, la parte centrale del programma precedente può essere realizzata con un ciclo `for`:

```
numero = 1;                numero = 1;
    somma = 0;              somma = 0;
    while(numero!=0) {      for(; numero!=0;) {
        printf("Inser. intero: ");    printf("Inser. intero: ");
        scanf("%d", &numero);        scanf("%d", &numero);
        somma = somma+numero;        somma = somma+numero;
    }                               }
```

Infatti, come si è già evidenziato, nel costrutto `for`

```
for(esp1; esp2; esp3)
```

è possibile sostituire *esp1*, *esp2* ed *esp3* con qualsiasi espressione, nella fattispecie *esp2* corrisponde al controllo `n!=0` (`n` diverso da 0) mentre *esp1* ed *esp3* corrispondono a espressioni vuote. La presenza dei punti e virgola è naturalmente obbligatoria ■.

### ✓ NOTA

L'istruzione `for`, con la sua chiarezza d'intenti, l'enorme potenza e compattezza, è largamente utilizzata dai programmatori C.

Supponiamo che oltre alla somma si desideri determinare il valore massimo della sequenza in ingresso, con la limitazione che i valori debbano essere tutti positivi. Una volta inizializzata la variabile intera `max` a zero il ciclo

```

somma = 0;
max = 0;

i = 1;
while(numero!=0 && i<=10)
{
    printf("Valore int.: ");
    scanf("%d", &numero);
    if(numero>max)
        max = numero;
    somma = somma+numero;
    i++;
}
printf("Somma: %d\n", somma);
printf("Maggiore: %d\n", max);
}

```

### Listato 3.5 Diramazione if all'interno di una iterazione while

L'incremento della variabile che conta il numero di valori immessi può essere inserito direttamente nella parte *espressione di while*:

```

while(numero!=0 && i++<=10) {
    printf("Inser. intero positivo: ");
    scanf("%d", &numero);
    if(numero>max) max=numero;
    somma+=numero;
}

```

## 3.5 Istruzione do-while

Quando l'istruzione compresa nel ciclo deve essere comunque eseguita almeno una volta, risulta più comodo utilizzare il costrutto

```
do  
    istruzione  
while (cond);
```

```

}
while (numero != 0 && i <= 10);

printf("Somma: %d\n", somma);
printf("Maggiore: %d\n", max);
}

```

### Listato 3.6 Esempio di utilizzo del costrutto do-while

Il ciclo viene ripetuto fino a quando la condizione del `while` risulta essere vera, o in altre parole si esce dal ciclo quando la condizione del `while` risulta essere falsa. Per trasformare un `while` in `do-while` si deve semplicemente porre il `do` all'inizio del ciclo e il `while` (*esp*) alla fine dello stesso. Il ciclo poteva essere più sinteticamente espresso come segue:

```

do {
    printf("Valore int.: ");
    scanf("%d", &numero);
    if (numero > max)
        max = numero;
    somma = somma + numero;
}
while (numero != 0 && ++i <= 10);

```

In cui l'operatore `++` deve obbligatoriamente precedere il nome della variabile in quanto l'incremento deve avvenire prima del controllo `i <= 10`.

Questo modo di operare porta a istruzioni lunghissime, difficilmente leggibili, consigliamo pertanto di usare l'operatore virgola essenzialmente là dove ci siano da inizializzare o incrementare più variabili che controllano il ciclo.

## 3.7 Cicli annidati

In un blocco `for` o `while`, così come nei blocchi `if-else`, può essere presente un numero qualsiasi di istruzioni di ogni tipo. Si sono visti esempi di cicli all'interno di costrutti `if` e viceversa, ora vediamo un esempio di cicli innestati uno nell'altro.

Per ripetere una determinata istruzione  $n*m$  volte possiamo scrivere

```
for(i=1; i<=n; i++)
    for(j=1; j<=m; j++)
        printf("i: %d j: %d \n", i, j);
```

Se prima dell'inizio del ciclo  $n$  ha valore 2 e  $m$  ha valore 3 l'esecuzione provocherà la seguente visualizzazione:

```
i: 1      j: 1
i: 1      j: 2
i: 1      j: 3
i: 2      j: 1
i: 2      j: 2
i: 2      j: 3
```

Alla variabile  $i$  viene assegnato il valore 1 e si esegue il ciclo interno in cui la variabile  $j$  assume via via i valori 1, 2 e 3; a questo punto l'istruzione di controllo del ciclo interno appura che  $j$  non sia minore o uguale a  $m$  ( $4 \leq 3$ ) e il controllo ripassa al ciclo esterno;  $i$  assume il valore 2 e si ripete l'esecuzione del ciclo interno.

Se desideriamo produrre sul video una serie di  $n$  linee e  $m$  colonne costituite dal carattere +, possiamo scrivere il programma del Listato 3.7

## 3.8 Salti condizionati e incondizionati

L'istruzione `break` consente di interrompere l'esecuzione del `case`, provocando un salto del flusso di esecuzione alla prima istruzione successiva. Una seconda possibilità di uso di `break` è quella di forzare la terminazione di un'iterazione `for`, `while` o `do-while`, provocando un salto alla prima istruzione successiva al ciclo.

Riprendiamo il programma per il calcolo della somma e la ricerca del massimo dei numeri immessi dall'utente. L'istruzione di controllo del ciclo ne bloccava l'esecuzione se veniva immesso in ingresso il valore zero o quando fossero già stati inseriti dieci valori. Scorporiamo quest'ultima verifica affidandola a un'altra istruzione:

```
for(i=1; numero!=0; i++) {  
    printf("Inser. intero positivo: \t");  
    scanf("%d", &numero);  
    if(numero>max)    max=numero;  
    somma+=numero;  
    if(i==10) break;  
}
```

Naturalmente era migliore la soluzione precedente:

```
for(i=1; numero!=0 && i<=10; i++)
```

L'istruzione `break` provoca l'uscita solo dal ciclo più interno. Per esempio la sequenza di istruzioni

```
for(j=1; j<=3; j++) {  
    somma = 0;  
    max = 0;  
    numero = 1;  
    for(i=1; numero!=0; i++) {  
        printf("Inser. intero positivo: \t");
```



Generalmente `exit` viene inserito dopo la verifica negativa di una condizione indispensabile per il proseguimento dell'esecuzione. Per ipotesi, un programma relativo a un gioco potrebbe richiedere la presenza di una scheda grafica nel sistema per essere eseguito:

```
if(!scheda_grafica()) exit;  
gioco();
```

`scheda_grafica` è una funzione definita dall'utente che ha valore vero se almeno una delle schede grafiche richieste è presente e falso in caso contrario. In quest'ultimo caso viene eseguito `exit` e il programma ha termine ■.

L'istruzione `goto` richiede un'*etichetta* – un identificatore C valido – seguito da un carattere due punti. Tale identificatore deve essere presente nell'istruzione `goto`:

```
i=1;  
ciclo:  
    i++;  
    printf("Inser. intero positivo: \t");  
    scanf("%d", &numero);  
    if(numero>max)    max=numero;  
    somma+=numero;  
if(numero!=0 && i<=10) goto ciclo;
```

Le ragioni della programmazione strutturata, tra cui pulizia ed eleganza del codice, sconsigliano l'uso generalizzato di `break`, `continue` ed `exit`, e “proibiscono” quello del `goto`.



```
scanf("%f", &x);
```

memorizza il valore passato dall'utente nella variabile `float x`. Al posto di `%f` si può utilizzare indifferentemente `%e`. Analogamente per le variabili `double` si usa `%f` o `%lf` ■.

Ogni definizione di costante che includa un punto decimale fa sì che venga creata una costante di tipo `double`:

```
#define PI 3.14159
```

definisce la costante `PI` che può essere utilizzata all'interno del programma al posto del valore `3.14159`; naturalmente il valore di `PI` non può essere modificato ■.

## 3.10 Operazioni in virgola mobile

Le operazioni aritmetiche permesse sulle variabili `float` e `double` sono le stesse che per gli `int`, e si possono scrivere espressioni con variabili di tipo misto. In ogni espressione dove compaia una variabile `float` (`double`) il calcolo viene eseguito considerando le parti frazionarie in precisione semplice (doppia). Naturalmente quando si va ad assegnare il valore ottenuto a una variabile, se essa è di precisione inferiore al risultato può succedere che ciò che si ottiene non sia significativo. Per esempio, date le seguenti dichiarazioni

```
int i;  
float x;  
double y;
```

se alle richieste

```
printf("\n\n Digitare un valore reale: ");  
scanf("%f", &x);  
printf("\n\n Digitare un valore intero: ");
```

e altre di uso generale come

<code>log(x)</code>	logaritmo in base <code>e</code> di <code>x</code>
<code>log10(x)</code>	logaritmo in base 10 di <code>x</code>
<code>sqrt(x)</code>	radice quadrata

L'uso di variabili di alta precisione dovrebbe comunque essere limitato ai casi di effettiva utilità in quanto utilizzare un `float` al posto di un `int` o un `double` al posto di uno degli altri due tipi, oltre a portare a una maggiore occupazione di memoria, determina una maggior lavoro di elaborazione delle operazioni e quindi diminuisce i tempi di risposta. Inoltre, a nostro avviso, usare una variabile `float` invece di un `int` dove non sia necessario porta a una peggiore leggibilità dei programmi.

## 3.11 Zero di una funzione

Per esercitarci con le strutture iterative e i numeri reali prendiamo in considerazione il problema del calcolo dello zero di una funzione continua  $f(x)$  con il cosiddetto metodo *dicotomico*. Ricordiamo che si dice *zero* di  $f$  un numero  $x_0$  tale che  $f(x_0)=0$ .

Sia  $f(x)$  una funzione continua che negli estremi dell'intervallo  $[a,b]$  assume valori di segno discorde, ovvero uno negativo e uno positivo e quindi tale che  $f(a)*f(b)<0$ , come mostrato in Figura 3.2. Sia  $m=(a+b)/2$  il punto medio dell'intervallo; se  $f(m)=0$  abbiamo ottenuto lo zero cercato, altrimenti si considera:

- l'intervallo  $[a,m]$  se  $f(a)$  e  $f(m)$  hanno segno discorde;
- l'intervallo  $[m,b]$  se  $f(a)$  e  $f(m)$  hanno segno concorde.

Si itera quindi lo stesso procedimento nell'intervallo appena determinato e si prosegue fino a trovare uno zero di  $f$ .

È importante osservare come per la corretta soluzione del problema sia indispensabile che  $f(a)$  e  $f(b)$  abbiano segno

```

    fb = 2*b*b*b-4*b+1; /* Calcolo della funzione per x=b */
}
while (fa*fb>0);

/* calcolo zero f */
do {
    m = (a+b)/2;
    fm = 2*m*m*m-4*m+1; /* Calcolo della funzione per x=m */
    if (fm!=0) {
        fa = 2*a*a*a-4*a+1; /* Calcolo della funzione per x=a */
        fb = 2*b*b*b-4*b+1; /* Calcolo della funzione per x=b */
        if (fa*fm<0) b=m; else a=m;
        fm = 2*m*m*m-4*m+1; /* Calcolo della funzione per x=m */
    }
}
while (fabs(fm) > ERR);

printf("Zero di f in %7.2f\n", m);
}

```

Listato 3.8 Programma per il calcolo dello zero di una funzione



\* 2. Predisporre un programma, che determini il maggiore, il minore e la media degli  $n$  valori immessi dall'utente.

\* 3. Predisporre un programma che stampi un rettangolo la cui cornice sia costituita da caratteri asterisco, e la parte interna da caratteri Q. Il numero di linee e di colonne del rettangolo viene deciso a tempo di esecuzione dall'utente; per esempio se il numero delle linee è uguale a 5 e il numero di colonne a 21, sul video deve apparire:

```
*****
*QQQQQQQQQQQQQQQQQQQ*
*QQQQQQQQQQQQQQQQQQQ*
*QQQQQQQQQQQQQQQQQQQ*
*****
```

\* 4. Ripetere l'esercizio 3 ma permettendo all'utente di decidere anche i caratteri che devono comporre la cornice e la parte interna del rettangolo e quante volte debba essere ripetuta la visualizzazione del rettangolo.

5. Realizzare un programma che richieda all'utente  $n$  interi, e visualizzi il numero di volte in cui sono stati eventualmente immessi i valori 10, 100 e 1000.

6. Predisporre un programma che visualizzi la tavola pitagorica del sistema di numerazione decimale.

7. Scrivere un programma che visualizzi tutte le coppie di numeri presenti sulla superficie dei pezzi del domino.

8. Supponiamo che  $x$ ,  $y$ ,  $z$  e  $t$  siano variabili di tipo `float` e che  $a$ ,  $b$  e  $c$  siano di tipo `int`. Determinare il valore di  $a$  e  $x$  dopo l'esecuzione delle seguenti istruzioni.

```
y = 2.4;
z = 7.0;
b = 3;
c = 7;
t = 0.1E2;
```

## 4.1 Array

Quando si ha la necessità di trattare un insieme omogeneo di dati esiste una soluzione diversa da quella di utilizzare tante variabili dello stesso tipo: definire un *array*, ovvero una variabile strutturata dove è possibile memorizzare più valori tutti dello stesso tipo. Intuitivamente, un array monodimensionale o *vettore* può essere immaginato come un contenitore suddiviso in tanti scomparti quanti sono i dati che vi si vogliono memorizzare. Ognuno di questi scomparti, detti *elementi* del vettore, contiene un unico dato ed è individuato da un numero progressivo, detto *indice*, che specifica

In definitiva, un vettore è una struttura di dati composta da un numero determinato di elementi tutti dello stesso tipo, ognuno dei quali è individuato da un indice specifico. È ora chiaro perché i vettori si dicano *variabili strutturate* mentre all'opposto tutte le variabili semplici siano anche dette *non strutturate*. Il tipo dei dati contenuti nel vettore viene detto *tipo del vettore*, ovvero si dice che il vettore è di quel particolare tipo.

Dunque per il vettore, come per qualsiasi altra variabile, devono essere definiti il nome e il tipo; inoltre si deve esplicitarne la lunghezza, cioè il numero di elementi che lo compongono. Una scrittura possibile è perciò la seguente:

```
int a[6];
```

Come sempre in C, prima deve essere dichiarato il tipo (nell'esempio `int`), poi il nome della variabile (`a`), successivamente – tra parentesi quadre – il numero degli elementi (`6`) che dev'essere un intero positivo. Questa dichiarazione permette di riservare in memoria centrale uno spazio strutturato come in Figura 4.2.

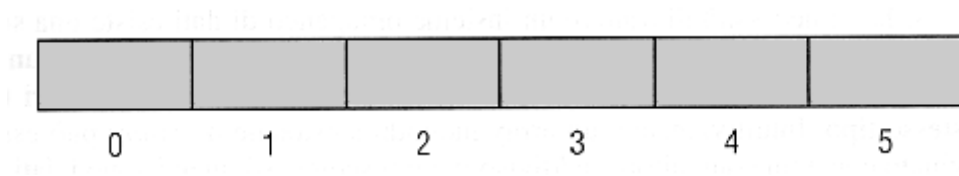


Figura 4.2 Struttura dell'array `a[6]`

Per accedere a un singolo elemento di `a` si deve specificare il nome del vettore seguito dall'indice dell'elemento posto tra parentesi quadre. L'array `a` è composto da sei elementi e l'indice può quindi assumere i valori: 0, 1, 2, 3, 4, e 5. Le istruzioni

```
a[0] = 71;  
a[1] = 4;
```

assegnano al primo elemento del vettore `a` il valore 71 e al secondo 4. Se `b` è una variabile intera (cioè dello stesso



Anche per ricercare all'interno dell'array valori che soddisfano certe condizioni si utilizzano abitualmente i cicli:

```
/* Ricerca del maggiore */
    max = a[0];
    for(i=1; i<=5; i++)
        if(a[i]>max) max = a[i];
```

L'esempio permette di determinare il maggiore degli elementi dell'array `a`: la variabile `max` viene inizializzata al valore del primo elemento del vettore, quello con indice zero. Successivamente ogni ulteriore elemento viene confrontato con `max`: se risulta essere maggiore, il suo valore viene assegnato a `max` ■. Il ciclo deve comprendere dunque tutti gli elementi del vettore meno il primo, perciò l'indice `i` assume valori che vanno da 1 a 5.

Nel Listato 4.1 è riportato un programma che richiede all'utente l'immissione del punteggio raggiunto da sei studenti, li memorizza nel vettore `voti` e ne determina il maggiore, il minore e la media.

```
/* Memorizza in un array di interi il punteggio raggiunto da sei
   studenti e ne determina il maggiore, il minore e la media */

#include <stdio.h>

main()
{
    int voti[6];
    int i, max, min;
    float media;

    printf("VOTI STUDENTI\n\n");
    /* Immissione voti */
```

Si noti che la richiesta dei voti all'utente viene fatta evidenziando il numero d'ordine che corrisponde al valore dell'indice aumentato di una unità.

```
printf("Voto %d° studente: ", i+1);
```

Alla prima iterazione appare sullo schermo:

Voto 1° studente:

Considerazioni analoghe a quelle fatte per il calcolo del maggiore valgono per il minimo e la media.

Nella pratica la memorizzazione in un vettore ha senso quando i valori debbono essere utilizzati più volte, come nel caso precedente. Nell'esempio, comunque, i calcoli potevano essere effettuati all'interno della stessa iterazione con un notevole risparmio di tempo di esecuzione: ■

```
/* Ricerca maggiore, minore e media */
max = voti[0];
min = voti[0];
media = voti[0];
for(i = 0; i <= 5; i++) {
    if(voti[i] > max)
        max = voti[i];
    if(voti[i] < min)
        min = voti[i];
    media = media+voti[i];
}
media = media / 6;
```

```

#include <stdio.h>

#define MAX_CONC 1000    /* massimo numero di concorrenti */
#define MIN_PUN  1      /* punteggio minimo per ogni prova */
#define MAX_PUN  10      /* punteggio massimo per ogni prova */

main()
{
    float proval[MAX_CONC], prova2[MAX_CONC], totale[MAX_CONC];
    int i, n;

    do {
        printf("\nNumero concorrenti: ");
        scanf("%d", &n);
    }
    while(n<1 || n>MAX_CONC);

    /* Per ogni concorrente, richiesta punteggio nelle due prove */
    for(i=0; i<n; i++) {
        printf("\nConcorrente n.%d \n", i+1);

        do {
            printf("Prima prova: ");
            scanf("%f", &proval[i]);
        }
        while(proval[i]<MIN_PUN || proval[i]>MAX_PUN);

        do {
            printf("Seconda prova: ");

```

memorizzato in `MIN_PUN` e `MAX_PUN` i limiti inferiore e superiore del punteggio assegnabile, in maniera che, se questi venissero modificati, basterebbe intervenire sulle loro definizioni perché il programma continui a funzionare correttamente. Infine calcoliamo il punteggio totale e lo visualizziamo ■. Un esempio di esecuzione è mostrato in Figura 4.4.

### ✓ NOTA

Si noti che soltanto  $n$  elementi di ogni array vengono utilizzati veramente; quindi se, per esempio, i concorrenti sono dieci, si ha un utilizzo di memoria pari a solo il 10 per mille (valore attuale di `MAX_CONC`); in questo modo però il programma è più flessibile.

Esistono altre soluzioni più complesse che permettono di gestire la memoria dinamicamente (cioè di adattarla alle effettive esigenze del programma), anziché staticamente (cioè riservando a priori uno spazio): le vedremo in seguito ■.

```
Numero concorrenti: 3
```

```
Concorrente n.1
```

```
Prima prova: 8    Seconda prova: 7
```

```
Concorrente n.2
```

```
Prima prova: 5    Seconda prova: 9
```

```
Concorrente n.3
```

```
Prima prova: 8    Seconda prova: 8
```

Le variabili `extern` sono quelle che vengono definite prima di `main`. L'inizializzazione si ottiene inserendo i valori tra parentesi graffe, separati da una virgola:

```
int voti[6] = {11, 18, 7, 15, 21, 9};
```

Il compilatore fa la scansione dei valori presenti tra parentesi graffe da sinistra verso destra e genera altrettanti assegnamenti consecutivi agli elementi del vettore, rispettando la loro posizione; dunque `voti[0]` assume il valore 11, `voti[1]` 18, `voti[2]` 7 ecc. Quando tutti gli elementi dell'array vengono inizializzati è possibile omettere l'indicazione del numero di elementi, e scrivere

```
int voti[] = {11, 18, 7, 15, 21, 9};
```

È infatti il compilatore stesso che conta i valori e di conseguenza determina la dimensione del vettore.

Gli array di caratteri, comunemente detti *stringhe*, possono essere inizializzati anche inserendo il loro contenuto tra doppi apici:

```
char frase[] = "Analisi, requisiti";
```

## 4.4 Matrici

Nei paragrafi precedenti abbiamo trattato i vettori, detti anche matrici monodimensionali. Per la memorizzazione abbiamo usato una variabile di tipo array dichiarandone il numero di componenti, per esempio:

```
int vet[3];
```

Per accedere direttamente a ciascuno degli elementi del vettore si è utilizzato un indice che varia da zero a  $n-1$ . Nell'esempio  $n$  è uguale a 3.

In una matrice bidimensionale i dati sono organizzati per righe e per colonne, come se fossero inseriti in una tabella. Per la memorizzazione si utilizza una variabile di tipo array specificando il numero di componenti per ciascuna delle

```

main()
{
int i, j;

printf("\n \n CARICAMENTO DELLA MATRICE \n \n");
for(i=0; i<4; i++)
    for(j=0; j<3; j++) {
        printf("Inserisci linea %d colonna %d val: ", i, j);
        scanf("%d", &mat[i][j]);
    };

/* Visualizzazione */
for(i=0; i<4; i++) {
    printf("\n");
    for(j=0; j<3; j++)
        printf("%5d", mat[i][j]);
}
}

```

#### Listato 4.3 Esempio di utilizzo di un array bidimensionale

Per effettuare il caricamento dei dati nella matrice utilizziamo due cicli, uno più esterno che mediante la variabile *i* fa la scansione delle righe da zero a 3 (4-1) e un altro che percorre, per mezzo della variabile *j*, le colonne da zero a 2 (3-1):

```

for(i=0; i<4; i++)
    for(j=0; j<3; j++) {
        printf("Inserisci linea %d colonna %d val:", i, j);

```

```
int n, m;
int i, j;

/* Richiesta delle dimensioni */
do {
    printf("\nNumero di linee: ");
    scanf("%d", &n);
}
while((n>=MAXLINEE) || (n<1));

do {
    printf("Numero di colonne: ");
    scanf("%d", &m);
}
while((m>=MAXCOLONNE) || (m<1));

printf("\n \n CARICAMENTO DELLA MATRICE \n \n");
for(i=0; i<n; i++)
    for(j=0; j<m; j++) {
        printf("Inserisci linea %d colonna %d val:", i, j);
        scanf("%d", &mat[i][j]);
    };

/* Visualizzazione */
for(i=0; i<n; i++) {
    printf("\n");
    for(j=0; j<m; j++)
        printf("%5d", mat[i][j]);
}
```

per  $i=1..N$ ,  $j=1..M$

Il prodotto così definito si può ottenere soltanto se il numero di colonne della prima matrice ( $P$ ) è uguale al numero di righe della seconda. La matrice  $pmat$  è dunque costituita da  $N$  righe e  $M$  colonne.

Consideriamo le matrici di Figura 4.5: l'elemento  $[2][4]$  della matrice prodotto è dato da

$$\begin{aligned} pmat[2][4] = & mat1[2][0]*mat2[0][4] + \\ & mat1[2][1]*mat2[1][4] + \\ & mat1[2][2]*mat2[2][4] \end{aligned}$$

ossia

$$pmat[2][4] = 5*3 + 2*4 + 0*5 = 23$$

Venendo dunque al programma richiesto (Listato 4.5), in primo luogo si devono caricare i dati delle due matrici.

```
/* Calcolo del prodotto di due matrici */

#include <stdio.h>

#define N 4
#define P 3
#define M 5

int mat1[N][P];          /* prima matrice */
int mat2[P][M];          /* seconda matrice */
int pmat[N][M];          /* matrice prodotto */
```



```

    for(j=0; j<M; j++)
        printf("%5d", mat1[i][j]);
}

printf("\n \n SECONDA MATRICE \n ");
for(i=0; i<P; i++) {
    printf("\n");
    for(j=0; j<M; j++)
        printf("%5d", mat2[i][j]);
}

printf("\n \n MATRICE PRODOTTO \n ");
for(i=0; i<N; i++) {
    printf("\n");
    for(j=0; j<M; j++)
        printf("%5d", pmat[i][j]);
}
}

```

#### Listato 4.5 Calcolo del prodotto tra matrici

Per ottenere il valore dell'elemento  $i, j$  della matrice prodotto lo si inizializza a zero:

```
pmat[i][j] = 0;
```

Successivamente, con un ciclo che fa la scansione della riga  $i$  di `mat1` e della colonna  $j$  di `mat2`, si accumula in `pmat[i][j]` la sommatoria dei prodotti dei corrispondenti elementi di `mat1` e `mat2`:

```
for(k=0; k<P; k++)
```

## CARICAMENTO DELLA SECONDA MATRICE

Inserisci linea 0 colonna 0 val:**2**  
Inserisci linea 0 colonna 1 val:**0**  
Inserisci linea 0 colonna 2 val:**4**  
Inserisci linea 0 colonna 3 val:**0**  
Inserisci linea 0 colonna 4 val:**3**  
Inserisci linea 1 colonna 0 val:**0**  
Inserisci linea 1 colonna 1 val:**1**  
Inserisci linea 1 colonna 2 val:**5**  
Inserisci linea 1 colonna 3 val:**1**  
Inserisci linea 1 colonna 4 val:**4**  
Inserisci linea 2 colonna 0 val:**21**  
Inserisci linea 2 colonna 1 val:**1**  
Inserisci linea 2 colonna 2 val:**2**  
Inserisci linea 2 colonna 3 val:**2**  
Inserisci linea 2 colonna 4 val:**5**

## PRIMA MATRICE

1	0	0
22	-6	3
5	2	0
11	4	7

## SECONDA MATRICE

2	0	4	0	3
1	1	5	1	1

lunghezza 10 ottenuto gli elementi di  $v1$  e  $v2$ . Visualizzare  $v1$ ,  $v2$  e  $w$ .  
Per esempio: se  $v1$  e  $v2$  sono i vettori di caratteri

v1	<table><tr><td>B</td><td>N</td><td>S</td><td>I</td><td>O</td></tr></table>	B	N	S	I	O						
B	N	S	I	O								
v2	<table><tr><td>E</td><td>I</td><td>S</td><td>M</td><td>!</td></tr></table>	E	I	S	M	!	si deve ottenere il vettore					
E	I	S	M	!								
w	<table><tr><td>B</td><td>E</td><td>N</td><td>I</td><td>S</td><td>S</td><td>I</td><td>M</td><td>O</td><td>!</td></tr></table>	B	E	N	I	S	S	I	M	O	!	
B	E	N	I	S	S	I	M	O	!			

6. Scrivere un programma che, letti gli elementi di due vettori  $v1$  e  $v2$  di lunghezza  $n$ , inizializzi un terzo vettore  $w$  di lunghezza  $n$  con i valori

```
w(i) = 1      se v1(i) > v2(i);  
w(i) = 0      se v1(i) = v2(i);  
w(i) = -1     se v1(i) < v2(i).
```

Visualizzare quindi  $v1$ ,  $v2$  e  $w$ .

7. Scrivere un programma che, inizializzato un vettore di `char` con una stringa di lettere dell'alfabeto e punteggiatura, visualizzi il numero complessivo delle vocali e delle consonanti del vettore.

8. Scrivere un programma di inizializzazione che richiedo un elemento controlli, prima di inserirlo nel vettore, se è già presente, nel qual caso chieda che l'elemento sia digitato di nuovo.

9. Scrivere un programma che inizializzi e quindi visualizzi una matrice di `int` in cui ciascun elemento è dato dalla somma dei propri indici.

10. [*Matrici simmetriche*] Una matrice quadrata  $n \times n$  di un tipo qualsiasi si dice simmetrica se gli elementi simmetrici rispetto alla diagonale principale (dal vertice alto sinistro al vertice basso destro) sono due a due uguali. Scrivere un programma che, letta una matrice quadrata di interi, controlli se è simmetrica.

crescente o decrescente, unire (fondere) due o più insiemi in un unico insieme evitando possibili duplicazioni. Queste tre attività, che in informatica vengono indicate rispettivamente con i termini di *ricerca*, *ordinamento* e *fusione* oppure con i loro equivalenti inglesi *search*, *sort* e *merge*, sono estremamente frequenti e svolgono un ruolo della massima importanza in tutti i possibili impieghi degli elaboratori. È stato per esempio stimato che l'esecuzione dei soli programmi di ordinamento rappresenti oltre il 30% del lavoro svolto dai computer. È quindi ovvio come sia della massima importanza disporre di programmi che svolgano questi compiti nel minor tempo possibile.

## 5.2 Ricerca completa

Un primo algoritmo per determinare se un valore è presente all'interno di un array, applicabile anche a sequenze non ordinate, è quello comunemente detto di *ricerca completa*, che opera una scansione sequenziale degli elementi del vettore confrontandoli con il valore ricercato. Nel momento in cui tale verifica dà esito positivo la scansione ha termine e viene restituito l'indice dell'elemento all'interno dell'array stesso.

Per determinare che il valore non è presente, il procedimento (Listato 5.1) deve controllare uno a uno tutti gli elementi fino all'ultimo, prima di poter sentenziare il fallimento della ricerca. L'array che conterrà la sequenza è `vet` formato da `MAX_ELE` elementi.

```
/* Ricerca sequenziale di un valore nel vettore */

#include <stdio.h>
#define MAX_ELE 1000    /* massimo numero di elementi */

main()
{
    char vet[MAX_ELE];
    int i, n;
    char c;
```

valori che la compongono. Successivamente l'utente inserisce il carattere da ricercare, che viene memorizzato nella variabile `c`. La ricerca parte dal primo elemento dell'array (quello con indice zero) e prosegue fintantoché il confronto fra `c` e `vet[i]` dà esito negativo e contemporaneamente `i` è minore di `n-1`:

```
while(c!=vet[i] && i<n-1) ++i;
```

Il corpo del ciclo è costituito dal semplice incremento di `i`. L'iterazione termina in tre casi:

1. `c` è uguale a `vet[i]` e `i` è minore di `n-1`;
2. `c` è diverso da `vet[i]` e `i` è uguale a `n-1`;
3. `c` è uguale a `vet[i]` e `i` è uguale a `n-1`.

In ogni caso `i` ha un valore minore o uguale a `n-1`, è dunque all'interno dei limiti di esistenza dell'array. L'`if` successivo determinerà se è terminato perché `c` è risultato essere uguale a `vet[i]`:

```
if(c==vet[i])
```

Esistono molte altre soluzioni al problema. Per esempio si potrebbe adottare un costrutto `while` ancora più sintetico, come il seguente:

```
while(c!=vet[i] && i++<n-1)  
    ;
```

dove il corpo del ciclo non è esplicitato in quanto l'incremento di `i` avviene all'interno dell'espressione di controllo. Si noti però che, in questo caso, al termine delle iterazioni `i` ha un valore maggiorato di uno rispetto alla condizione che ha bloccato il ciclo, e di questo bisognerà tener conto nel prosieguo del programma ■.

## 5.3 Ordinamenti

Un altro, fondamentale problema dell'informatica spesso collegato con la ricerca è quello che consiste nell'ordinare un insieme di dati secondo un certo criterio. Esistono molti algoritmi per risolvere questo problema, che in questa sezione non verranno trattati.



}

Si osserva dal confronto

```
vet[0] = 9
vet[1] = 18
vet[2] = 7
vet[3] = 15
vet[4] = 21
vet[5] = 11
```

	9 9 9 9 9 18 7 7 7 7 7 18 15 15 15 15 15 18 18 18 21 21 21 21 11 11 11 11 11 21	7 7 7 7 9 9 9 9 15 15 15 15 18 18 18 11 11 11 11 18 21 21 21 21	7 7 7 9 9 9 15 15 11 11 11 15 18 18 18 21 21 21	7 7 9 9 11 11 15 15 18 18 21 21
i	1 2 3 4 5	1 2 3 4	1 2 3	1 2
j	0	1	2	3

Figura 5.2 Esempio di ordinamento con l'algoritmo di *bubblesort* ottimizzato

```
p = n;  
do {  
    k = 0;  
    for(i=0; i<n-1; i++) {  
        if(vet[i]>vet[i+1]) {  
            aux = vet[i]; vet[i] = vet[i+1]; vet[i+1] = aux;  
            k = 1; p = i+1;  
        }  
    }  
}   
n = p;  
}  
while(k==1);
```

```
printf("\nElemento da ricercare: ");  
scanf("%ls", &ele);
```

```
/* ricerca binaria */
```

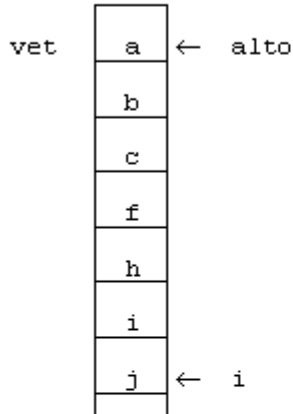
```
n = 6;  
alto = 0; basso = n-1; pos = -1;  
do {  
    i = (alto+basso)/2;  
    if(vet[i]==ele) pos = i;  
    else  
        if(vet[i]<ele)  
            alto = i+1;  
        else  
            basso = i-1;  
}  
while(alto<=basso && pos==-1);
```



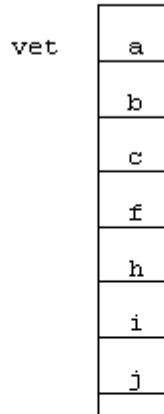
1. 1. `vet[i]` è uguale a `ele`, la ricerca è finita positivamente, si memorizza l'indice dell'elemento in `pos` e il ciclo di ricerca ha termine;
2. 2. `vet[i]` è minore di `ele`, la ricerca continua tra i valori maggiori di `vet[i]` che sono memorizzati negli elementi con indice compreso tra `i+1` e `basso`, per cui si assegna ad `alto` il valore `i+1`. Se non si sono già esaminati tutti gli elementi del vettore (`alto` non è minore o uguale a `basso`) la ricerca continua assegnando ancora una volta a `i` il valore  $(basso+alto)/2$ ;
3. 3. `vet[i]` è maggiore di `ele`, la ricerca continua tra i valori minori di `vet[i]` che sono memorizzati negli elementi con indice compreso tra `alto` e `i-1`, per cui si assegna a `basso` il valore `i-1`. Se non si sono già esaminati tutti gli elementi del vettore (`alto` non è minore di `basso`) la ricerca continua assegnando ancora una volta a `i` il valore  $(basso+alto)/2$ .

Valore cercato `o(ele='o')`

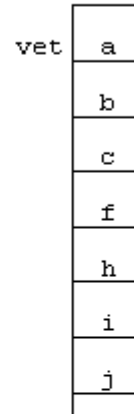
#### I ciclo



#### II ciclo



#### III ciclo



## 5.5 Fusione

Un altro algoritmo interessante è quello che partendo da due array monodimensionali ordinati ne ricava un terzo, anch'esso ordinato. I due array possono essere di lunghezza qualsiasi e in generale non uguale. Il programma del Listato 5.3 richiede all'utente l'immissione della lunghezza di ognuna delle due sequenze e gli elementi che le compongono. Successivamente ordina le sequenze ed effettua la fusione (*merge*) di una nell'altra, memorizzando il risultato in un array a parte.

```
/* Fusione di due sequenze ordinate */

#include <stdio.h>
#define MAX_ELE 1000

main()
{
    char vet1[MAX_ELE];      /* prima sequenza */
    char vet2[MAX_ELE];      /* seconda sequenza */
    char vet3[MAX_ELE*2];    /* merge */

    int n;                   /* lunghezza prima sequenza */
    int m;                   /* lunghezza seconda sequenza */

    char aux;                /* variabile di appoggio per lo scambio */

    int i, j, k, p, n1, m1;

    do {
        printf("Lunghezza prima sequenza: ");
```

```

        }
    }
    n1 = p;
}
while(k==1);

/* ordinamento seconda sequenza */
p = m; m1 = m;
do {
    k = 0;
    for(i=0; i<m1 - 1; i++) {
        if(vet2[i]>vet2[i+1]) {
            aux = vet2[i]; vet2[i] = vet2[i+1]; vet2[i+1] = aux;
            k = 1; p = i+1;
        }
    }
}
m1 = p;
}
while(k==1);

/* fusione delle due sequenze (merge) */
i = 0; j = 0; k = 0;
do {
    if(vet1[i]<=vet2[j])
        vet3[k++] = vet1[i++];
    else
        vet3[k++] = vet2[j++];
}

```

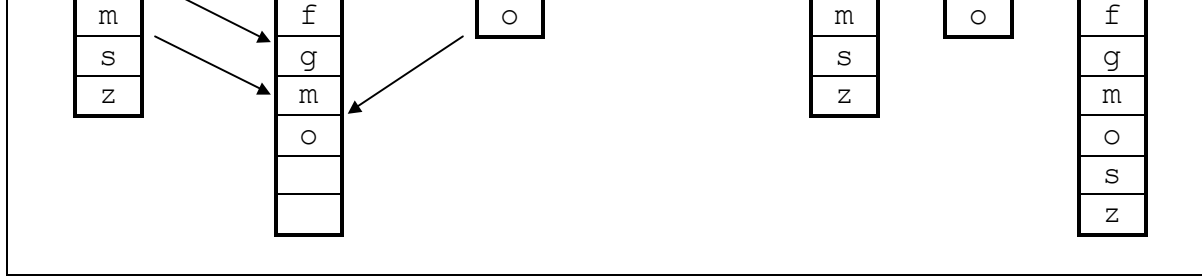


Figura 5.4 Risultato parziale e finale della fusione tra due vettori

In Figura 5.4 osserviamo il merge tra gli array ordinati *vet1* e *vet2* ordinati. L'operazione viene effettuata in due parti. La prima è data da:

```
i = 0; j = 0; k = 0;
do {
    if(vet1[i]<=vet2[j])
        vet3[k++] = vet1[i++];
    else
        vet3[k++] = vet2[j++];
}
while(i<n && j<m);
```

Si controlla se l'*i*-esimo elemento di *vet1* è minore o uguale al *j*-esimo elemento di *vet2*, nel qual caso si aggiunge *vet1*[*i*] a *vet3* e si incrementa *i*. Nel caso contrario si aggiunge a *vet3* l'array *vet2*[*j*] e si incrementa *j*. In ogni caso si incrementa *k*, la variabile che indicizza *vet3*, perché si è aggiunto un elemento a *vet3*. Dal ciclo si esce quando *i* ha valore *n*-1 o *j* ha valore *m*-1.

Si devono ancora aggiungere a *vet3* gli elementi di *vet1* (*i*=*n*-1) o di *vet2* (*j*=*n*-1) che non sono stati

vet	3	31	1	23	41	5	0	66	2	8	88	9	91	19	99
-----	---	----	---	----	----	---	---	----	---	---	----	---	----	----	----

- Verificare il comportamento della versione ottimizzata di bubblesort applicata al vettore del precedente esercizio. Quanti cicli interni si sono risparmiati rispetto alla prima versione?
- Calcolare il numero di confronti effettuati dall'algoritmo di ordinamento ingenuo applicato al vettore dell'Esercizio 4 e confrontarlo con quello di bubblesort.
- Scrivere un programma che, richiedi i valori di un vettore ordinato in modo crescente, li inverta ottenendo un vettore decrescente. Si chiede di risolvere il problema utilizzando un solo ciclo.
- Verificare il comportamento del programma di fusione applicato ai seguenti vettori:

vet1	3	31	41	43	44	45	80
------	---	----	----	----	----	----	----

vet2	5	8	21	23	46	51	60	66
------	---	---	----	----	----	----	----	----

- Modificare l'algoritmo di ricerca binaria nel caso il vettore sia ordinato in modo decrescente invece che crescente.
- Se il vettore è ordinato la ricerca completa può essere migliorata in modo da diminuire in media il numero di confronti da effettuare: come? Modificare in questo senso il programma esaminato nel presente capitolo.
- Scrivere un programma che, richiedi all'utente i primi  $n-1$  elementi già ordinati di un vettore di dimensione  $n$  e un ulteriore elemento finale, inserisca quest'ultimo nella posizione corretta facendo *scivolare* verso il basso tutti gli elementi più grandi.
- [*Insertion-sort*] Utilizzare l'algoritmo del precedente esercizio per scrivere un programma che ordini il vettore contemporaneamente all'inserimento dei dati da parte dell'utente.

È importante osservare la differenza tra le due inizializzazioni:

```
char d = 'r';  
char b[] = "r";
```

La prima assegna alla variabile `d` di tipo `char` il valore `r`, la seconda assegna all'array `b[]` la sequenza di caratteri `r` e `\0`; in quest'ultimo caso si tratta effettivamente di una stringa. Naturalmente, quando si desidera far riferimento a un carattere si deve inserirlo tra apici singoli: per esempio

```
b[2] = 't';
```

assegna al terzo elemento dell'array `b` il carattere `t`.

Il carattere terminatore `\0` ci permette di trattare le stringhe senza conoscere a priori la dimensione.

Il programma del Listato 6.1 consente di verificare la corrispondenza tra ogni carattere presente in una stringa e il suo equivalente valore all'interno del codice ASCII, espresso nel sistema decimale e ottale.

```
/* Visualizzazione caratteri di una stringa */  
  
#include <stdio.h>  
  
char frase[] = "Analisi, requisiti ";  
  
main()  
{  
    int i=0;  
    while(frase[i]!='\0') {  
        printf("%c = %d = %o \n", frase[i], frase[i], frase[i]);  
        i++;  
    }  
}
```

```
s = 115 = 105  
i = 105 = 151  
t = 116 = 164  
i = 105 = 151  
= 32 = 40
```

Comunque, se si desidera la visualizzazione dell'intera stringa, è possibile usare l'istruzione `printf` tramite la specifica del formato `%s`:

```
printf("%s", frase);
```

Tale istruzione, se inserita nel Listato 6.1 , restituirebbe:

```
Analisi, requisiti
```

In questo caso è l'istruzione `printf` stessa che provvede a stampare carattere per carattere la stringa e a bloccarsi nel momento in cui identifica il carattere `\0`.

## 6.2 Esempi di uso delle stringhe

Il programma del Listato 6.2 copia il contenuto di una stringa in un'altra.

```
/* Copia di una stringa su un'altra */  
  
#include <stdio.h>  
  
char frase[] = "Analisi, requisiti ";  
  
main()
```

L'istruzione `printf` stampa la stringa di partenza e quella di arrivo, per cui l'esecuzione del programma visualizzerà:

```
originale: Analisi, requisiti
        copia:      Analisi, requisiti
```

### ✓ NOTA

Abbiamo evidenziato il fatto che il costrutto `for` non contiene alcuna istruzione esplicita all'interno del ciclo ponendo il punto e virgola nella riga successiva:

```
for(i=0; (discorso[i]=frase[i])!='\0'; i++)
    ;
```

Questa una notazione è utilizzata spesso in C. In realtà, come abbiamo visto esaminando il programma, qualcosa di concreto accade a ogni iterazione: un elemento di `frase` viene assegnato a `discorso`; l'operazione relativa è però contenuta (nascosta) all'interno di un'espressione della parte controllo del `for`. È chiaro che si poteva ottenere lo stesso risultato scrivendo:

```
for(i=0; frase[i]!='\0'; i++)
    discorso[i]=frase[i];
discorso[i]='\0';
```

Anche se questa notazione è più familiare, si esorta a utilizzare la precedente per sfruttare al meglio le caratteristiche del C.

Se si desidera copiare soltanto alcuni caratteri della prima stringa sulla seconda si deve modificare l'istruzione `for` del Listato 6.2. Scrivendo



In questo caso indichiamo esplicitamente il numero di elementi (160) che compongono la variabile `frase`, poiché desideriamo definire un array che possa contenere più caratteri di quelli presenti nella stringa assegnatagli all'inizio (20):

```
char frase[160] = "Analisi, requisiti ";
```

In questo modo `frase` potrà contenere i caratteri che gli verranno concatenati. La prima istruzione `printf` richiede all'utente una stringa e `scanf` la inserisce nell'array di caratteri `dimmi`. In generale non si conosce il numero di caratteri che attualmente costituiscono la stringa di partenza, per cui si deve scorrerla fino a posizionare l'indice sul carattere terminatore:

```
for(i=0; (frase[i])!='\0'; i++)
    ;
```

Alla fine del ciclo `i` conterrà l'indice dell'elemento del vettore dov'è presente il carattere `\0`. Nel caso specifico, avendo assegnato a `frase` la stringa "Analisi, requisiti ", `i` avrà valore 20.

Adesso dobbiamo assegnare agli elementi successivi di `frase` il contenuto di `dimmi`:

```
for(j=0; (frase[i]=dimmi[j])!='\0'; i++,j++)
    ;
```

L'indice `j` scorre `dimmi` a partire dalla prima posizione, mentre `i` scorre `frase` a partire dal suo carattere terminatore; alla prima iterazione il carattere `\0` di `frase` viene sostituito dal primo carattere di `dimmi`. A ogni ciclo successivo viene assegnato a `frase[i]` il valore di `dimmi[j]`. All'ultima iterazione il carattere `\0` di `dimmi` viene posto in `frase`, così da chiuderla correttamente.

L'istruzione `printf` visualizza il nuovo contenuto di `frase`.

```
printf("frase: %s \n", frase);
```

Osserviamo di seguito una possibile esecuzione del programma.

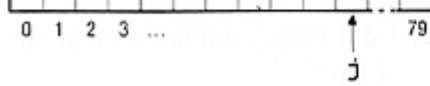


Figura 6.2 Stato degli array prima e dopo la concatenazione

Un altro modo per memorizzare una stringa è l'uso, all'interno di un ciclo, della funzione `getchar`, che cattura il carattere passato in ingresso (Listato 6.4).

```
/* Concatenazione di due stringhe
   introduzione della seconda stringa con getchar */

#include <stdio.h>

char frase[160] = "Analisi, requisiti ";

main()
{
    char dimmi[80];
    int i, j;

    printf("Inserisci una parola: ");
    for(i=0; (dimmi[i]=getchar())!='\n'; i++)
        ;
    dimmi[i]='\0';
    for(i=0; frase[i]!='\0'; i++)
        ;
    for(j=0; (frase[j]=dimmi[j]); j++)
        ;
    frase[j]='\0';
}
```

preoccupare di verificare che gli assegnamenti vengano effettuati su elementi definiti dell'array, per cui un più corretto ciclo d'inserimento del programma sarebbe:

```
for(i=0; ((dimmi[i]=getchar())!='\n') && (i<80)) ;i++)  
    ;
```

Il ciclo prosegue finché il carattere catturato è diverso da `\n` e contemporaneamente `i` è minore di 80.

Scriviamo adesso un programma che confronta due stringhe rivelando se la prima è uguale, maggiore o minore della seconda (Listato 6.5). L'ordinamento seguito è quello definito dal codice di rappresentazione dei caratteri, che nella maggior parte delle macchine è il codice ASCII.

```
#include <stdio.h>  
  
/* Confronto fra due stringhe */  
  
char prima[160] = "mareggiata";  
  
main()  
{  
    char seconda[80];  
    int i;  
  
    printf("Inserisci una parola: ");  
    for(i=0; ((seconda[i]=getchar()) != '\n') && (i<80) ;i++)  
        ;  
    seconda[i]='\0';  
    for(i=0; (prima[i] == seconda[i]) && (prima[i] != '\0') &&  
        (seconda[i] != '\0'); i++)  
        ;  
}
```

```
strcpy(stringa1, stringa2);
```

La funzione `strncpy` permette invece di copiare i primi *n* caratteri di `stringa2` in `stringa1`:

```
strncpy(stringa1, stringa2, n);
```

mentre la funzione `strcat` consente di concatenare `stringa2` a `stringa1`:

```
strcat(stringa1, stringa2);
```

La funzione `strcmp` serve a confrontare `stringa2` con `stringa1` (Listato 6.6):

```
strcmp(stringa1, stringa2);
```

Se risultano essere uguali viene restituito zero, se `stringa1` è maggiore di `stringa2` viene restituito un valore positivo, altrimenti un valore negativo ■.

```
/* Confronto tra due stringhe con strcmp */
```

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

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

```
char prima[160] = "mareggiata";
```

```
main()
```

```
{
```

```
char seconda[80];
```

```
int i, x;
```

```
printf("Inserisci una parola: ");
```

5. Data la seguente assegnazione alla stringa esercizio

```
esercizio='1234567890abcdefghijklmopqrstvuzABCDEFGHIJLMNOPQRSTUVWXYZ';
```

spostare i caratteri numerici dopo le lettere minuscole e prima delle lettere maiuscole, in modo che la stringa assuma il valore

```
abcdefghijklmopqrstvuz1234567890ABCDEFGHIJLMNOPQRSTUVWXYZ
```

Effettuare le operazioni necessarie senza utilizzare costanti che identifichino la posizione dei caratteri, ma reperire dinamicamente tali posizioni, in modo che il programma abbia una valenza più generale.

6. Scrivere un programma che, richieste all'utente le stringhe frase, parola1 e parola2, controlli se in frase è contenuta parola1, e in tal caso sostituisca tutte le sue occorrenze con parola2.

7. Scrivere un programma che controlli se una stringa richiesta all'utente è palindroma. (Una stringa si dice palindroma se si legge nello stesso modo da sinistra verso destra e da destra verso sinistra. Sono esempi di stringhe palindrome: ANNA, radar, anilina.)

8. Scrivere un programma che richieda all'utente una stringa controlli se vi compaiono almeno tre caratteri uguali consecutivi.

9. Scrivere un programma che richieda all'utente un carattere e una stringa e calcoli quindi il numero di occorrenze del carattere nella stringa.

10. Scrivere un programma che, letta una stringa composta da sole lettere dell'alfabeto, visualizzi il numero delle vocali, quello delle consonanti e la lettera più frequente.

11. Scrivere un programma che, letta una stringa composta da sole cifre (0..9), visualizzi accanto a ogni cifra il numero di volte che questa compare nella stringa. (Attenzione: si scriva un programma che utilizzi un solo ciclo.)

13. Scrivere un programma che richieda all'utente una stringa e ne visualizzi una seconda, ottenuta dalla prima

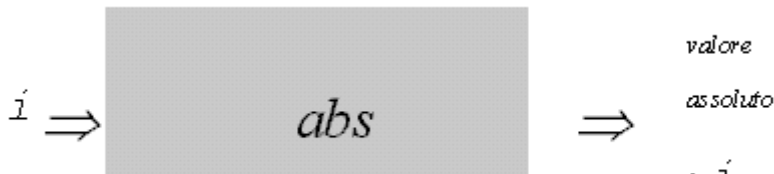
## 7.2 Sottoprogrammi C

In C i sottoprogrammi sono detti *funzioni*: a partire da uno o più valori presi in ingresso, esse *ritornano* (o *restituiscono*) un valore al programma chiamante. Come indicato in Figura 7.1, una funzione può essere pensata come una *scatola nera* che a determinati valori in ingresso fa corrispondere un determinato valore in uscita.



Figura 7.1 La funzione come scatola nera

Un esempio di funzione C è *abs(i)*, già utilizzata più volte. Considerando la funzione *abs* come una scatola nera, tutto quello che dobbiamo sapere – e in effetti già sappiamo – è che inserendo come argomento *i* di tale funzione un numero intero essa ne ritorna il valore assoluto (Figura 7.2).



```
{  
    return (c*c*c);  
}
```

## Listato 7.1 Dichiarazione, definizione e invocazione di una funzione

Per poter usare un identificatore occorre innanzitutto dichiararlo. La dichiarazione:

```
double cubo(float);
```

che precede main introduce l'identificatore cubo. Per mezzo di questa dichiarazione si specifica che cubo è il nome di una funzione che restituisce al programma chiamante un valore di tipo double. Inoltre si dichiara che la funzione cubo accetta in ingresso un solo valore come argomento, il cui tipo è float.

Si noti come con questa dichiarazione non si sia ancora definita la funzione cubo, cioè ancora non vengano specificate le istruzioni che caratterizzano la funzione; semplicemente abbiamo dichiarato un nuovo *nome*, cubo, e abbiamo detto a quale categoria appartiene questo nome. La definizione della funzione cubo avviene più tardi, dopo la fine del blocco di istruzioni di main:

```
double cubo(float c)  
{  
    return (c*c*c);  
}
```

Oltre al nome della funzione viene definito il numero, il tipo e il nome dei suoi parametri, cioè le variabili su cui essa agisce. Nel nostro esempio è presente un solo parametro, il cui tipo è float e il cui nome è c. Il compito svolto da cubo è molto semplice: il valore passato nel parametro c è moltiplicato per se stesso tre volte (c\*c\*c) e il risultato di questa espressione è convertito in double e restituito (con return) al programma chiamante.

Il programma chiamante non ha da fare altro che passare alla funzione cubo un valore. Nell'esempio lo fa passando a cubo il valore contenuto nella variabile a: cubo(a). Successivamente il valore calcolato da cubo viene assegnato a una

Nella dichiarazione di una funzione si potrebbero specificare anche i nomi dei parametri formali. Per esempio:

```
double cubo(float c);
```

è una dichiarazione valida. Il nome del parametro formale, però, è assolutamente superfluo. Ciò che conta in una dichiarazione è il tipo, o meglio la lista dei tipi dei parametri formali. Se in una dichiarazione di una funzione si specificano anche i nomi dei parametri formali il compilatore semplicemente li ignora.

## 7.4 Definizione di una funzione

In termini generali una funzione viene definita con la sintassi *prototyping* nel seguente modo:

```
tipo_ritorno nome_funz (tipo_par1 par1, ..., tipo_parN parN)
{
    ...
}
```

La definizione stabilisce il nome della funzione, i valori in ingresso su cui agisce – detti *parametri formali* –, il blocco di istruzioni che ne costituiscono il contenuto, e l'eventuale valore di ritorno. Per i nomi delle funzioni valgono le consuete regole in uso per gli identificatori. Nelle parentesi tonde che seguono il nome della funzione sono definiti i parametri formali specificandone il tipo e il nome.

Per ogni funzione introdotta nel programma occorre una definizione, ma si ricordi che in C non è ammesso che più funzioni abbiano lo stesso nome. Per esempio, le due definizioni, poste in uno stesso programma:

<pre>double cubo(float c) {     return (c*c*c); }</pre>	<pre>float cubo(int c) {     return (c*c*c); }</pre>
---	--



```
double quad(float c)
{
    return(c*c);
}
```

```
double cubo(float c)
{
    return(c*c*c);
}
```

```
double quar(float c)
{
    return(c*c*c*c);
}
```

```
double quin(float c)
{
    return(c*c*c*c*c);
}
```

```
double pote(float b, int e)
{
    switch (e) {
        case 0: return (1);
        case 1: return (b);
        case 2: return (quad( b ));
        case 3: return (cubo( b ));
```

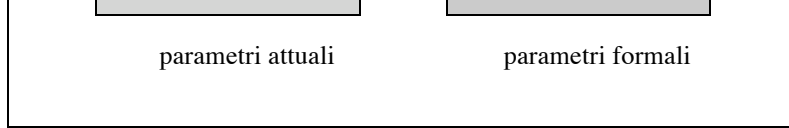
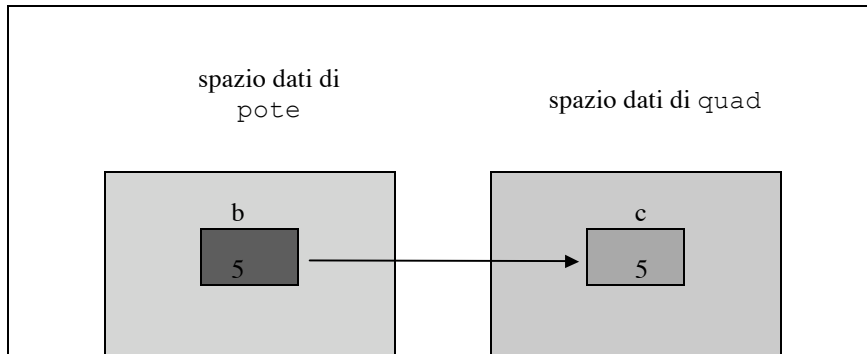


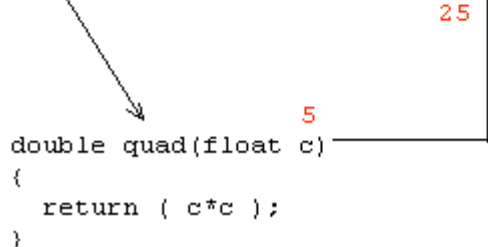
Figura 7.3 Passaggio di parametri tra main e pote

La funzione `pote`, che riceve in ingresso i valori nei parametri `b` ed `e` corrispondenti rispettivamente a `base` ed `esponente`, valuta il valore dell'esponente; dopo di ciò effettua una delle seguenti azioni: restituisce il valore 1 per esponente 0, la base stessa `b` per esponente 1, invoca la funzione `quad` per esponente 2, `cubo` per esponente 3, `quar` per esponente 4, `quin` per esponente 5 oppure restituisce -1 per segnalare la non disponibilità della potenza richiesta. Se l'esponente è 2, viene dunque invocata la funzione `quad`, cui la funzione `pote` trasmette il parametro attuale `b`:

```
case 2: return quad( b );
```

`quad` lo riceve in ingresso nel parametro formale `c` (Figura 7.2).





```
double quad(float c)
{
    return ( c*c );
}
```

Figura 7.5 Esempio di chiamata e ritorno delle funzioni

#### ✓ NOTA

In C, `main` non è una parola chiave del linguaggio ma il nome della funzione principale, cioè la funzione che, fra tutte quelle definite nel programma, viene eseguita per prima. La funzione `main` non è sintatticamente diversa dalle altre funzioni. La sua struttura:

```
main()
{
    ...
}
```

rispetta la sintassi generale di definizione delle funzioni. Alla funzione `main` possono essere anche passati dei parametri attraverso la linea di comando. Ritorneremo sull'argomento dopo aver trattato i puntatori ■.

## 7.5 Visibilità

Prima di passare all'elemento sintattico “return” dobbiamo osservare quanto segue

corrispondente parentesi graffa chiusa `}`. Quindi la definizione di `y` e `z` precede all'interno del blocco la definizione delle altre variabili locali `k` e `l` aventi anch'esse una visibilità che va dal punto di definizione alla fine del blocco ■. Per questo motivo la funzione:

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

è errata: in essa si tenta di definire due volte la variabile locale `x` nello stesso blocco.

Una dichiarazione di un nome in un blocco può nascondere, o come si dice in gergo, *mascherare*, la dichiarazione dello stesso nome in un blocco più esterno o la dichiarazione dello stesso nome globale. Un nome ridefinito all'interno di un blocco nasconde il significato precedente di quel nome, significato che verrà ripristinato all'uscita del blocco di appartenenza (Listato 7.3) ■.

```
int x;          /* nome globale */

f()
{
    int x;      /* x locale che nasconde x globale */
    x = 1;      /* assegna 1 a x locale */
    {
        int x;  /* nasconde il primo x locale */
        x = 2;  /* assegna 2 al secondo x locale */
    }
    x = 3;      /* assegna 3 al primo x locale */
}

scanf ("%d", &x); /* inserisce un dato in x globale */
```

```
double cubo( float c)
{
    return( c*c*c );
}
```

restituisce il controllo al programma chiamante e ritorna il cubo di `c` per mezzo dell'istruzione

```
return (c*c*c);
```

All'interno del blocco istruzioni di una funzione si possono avere più istruzioni `return`. Nella funzione `pote`:

```
double pote( b, e)
float b;
int e;
{
    switch (e) {
        case 0: return 1;
        case 1: return b;
        case 2: return quad(b);
        case 3: return cubo(b);
        case 4: return quar(b);
        case 5: return quin(b);
        default : return -1;
    }
}
```

a ogni scelta del costrutto `switch-case` corrisponde un'uscita e la restituzione di un diverso valore. In questo caso abbiamo usato la forma sintattica del `return` non inserendo l'espressione di ritorno tra parentesi tonde.

```

scanf("%c", &p);

printf("\nInserire base: ");
scanf("%f", &b);
printf("\nInserire altezza : ");
scanf("%f", &h);

a = area( b, h, p);

printf("Il poligono (b = %f, h = %f) ha area %f\n", b, h, a);
}

double area(float base, float altezza, char poligono)
{
    switch (poligono) {
        case 'T':    return (base * altezza/2.0);
        case 'R':    return (base * altezza);
        default :    return -1;
    }
}

```

#### Listato 7.4 Chiamata di funzione

Il contenuto delle variabili di tipo float `b`, `h` e di tipo char `p` vengono passati alla funzione `area` per mezzo dell'istruzione

```
a = area( b, h, p);
```

Le variabili `b`, `h` e `p` sono dette *parametri attuali* poiché contengono i valori di ingresso in quella specifica chiamata di funzione con i quali si può calcolare l'area del poligono. Al posto di una variabile si può comunque passare una costante

```

{
    float b, h;
    double tri, ret;

    printf("Inserire base: ");
    scanf("%f", &b);
    printf("Inserire altezza: ");
    scanf("%f", &h);

    tri = area(b, h, 'T');

    ret = area(b, h, 'R');

    printf("Il triangolo (b = %f, h = %f) ha area %f\n", b, h, tri);
    printf("Il rettangolo (b = %f, h = %f) ha area %f\n", b, h, ret);
}

double area(float base, float altezza, char poligono)
{
    switch (poligono) {
        case 'T':    return (base * altezza/2.0);
        case 'R':    return (base * altezza);
        default :    return -1;
    }
}

```

Listato 7.5 Le funzioni come strumento di riutilizzo del codice

Poiché con il passaggio dei parametri i valori dei parametri attuali sono travasati nelle locazioni di memoria corrispondenti ai parametri formali, si ha che la semantica del passaggio dei parametri è quella delle inizializzazioni di variabile: come per le inizializzazioni sono previste delle conversioni implicite di tipo. Più in dettaglio, si ha che nel passaggio dei parametri possono avvenire le conversioni seguenti.

<code>float</code>	I parametri attuali <code>float</code> sono convertiti in <code>double</code> prima di essere passati alla funzione. Di conseguenza tutti i parametri formali <code>float</code> sono automaticamente trasformati in <code>double</code> .
<code>char</code>	Tutti i parametri attuali <code>char</code> e <code>short int</code> , che esamineremo nei capitoli successivi, sono convertiti in <code>int</code> . Di conseguenza tutti i parametri formali <code>char</code> sono trasformati in <code>int</code> .

Occorre poi osservare che non è consentito il passaggio di parametri di tipo array, proprio perché in C il passaggio dei parametri avviene esclusivamente per valore. Infatti, se il compilatore si trovasse nella necessità di passare un array di tipo `int a[1000]`, occorrerebbe una quantità di tempo proporzionale per effettuare il travaso di valori tra due array di `1000 int`.

Oltre al passaggio *esplicito* di parametri, è possibile anche il passaggio *implicito*. Infatti basta definire una variabile globale sia alla funzione chiamante sia a quella chiamata per ottenere la condivisione della variabile stessa. Si consideri l'esempio del Listato 7.6, in cui la variabile globale

```
char str[] = "Lupus in fabula";
```

è visibile sia dalla funzione `main` sia dalla funzione `lung_string`: quest'ultima fa riferimento a `str` per calcolarne il numero di caratteri, mentre la funzione `main` vi fa riferimento per visualizzarne il contenuto.

```
#include <stdio.h>

char str[] = "Lupus in fabula";

int lung_string(void):
```



Abbiamo trattato il passaggio di parametri e la restituzione di un valore da parte di una funzione. Prendiamo ora in esame funzioni che non restituiscono alcun valore, e funzioni che non hanno parametri. In entrambi i casi il C mette a disposizione un “tipo” speciale detto `void`.

Tipico esempio di funzioni che non restituiscono alcun valore è quello delle funzioni il cui scopo è la visualizzazione di un messaggio o, più in generale, la produzione di un’uscita su uno dei dispositivi periferici. Queste funzioni sono talvolta conosciute con il curioso nome di funzioni “lavandino” (*sink*, in inglese) poiché prendono dati che riversano in una qualche uscita, senza ritornare niente al chiamante. Un esempio di funzione “lavandino” è la funzione `stampa_bin` (Listato 7.7).

```
#include <stdio.h>
#define DIM_INT 16

void stampa_bin ( int );

main()
{
    char resp[2];
    int  num;

    resp[0] = 's';

    while( resp[0] == 's' ) {
        printf("\nInserisci un intero positivo: ");
        scanf("%d", &num);
        printf("La sua rappresentazione binaria è: ");

        stampa_bin( num );

        printf("\nVuoi continuare? (s/n): ");
```

Vuoi continuare? (s/n): **n**

La funzione `stampa_bin` divide ripetutamente per 2 il decimale `v` e memorizza i resti delle divisioni intere nel vettore `a[]`, che poi legge a ritroso per visualizzare l'equivalente binario del decimale `v`. Come il lettore avrà osservato, sia nella dichiarazione

```
void stampa_bin( int );
```

sia nella definizione

```
void stampa_bin( int v )
{
    ...
}
```

si usa lo specificatore di tipo `void` per indicare l'assenza di un valore di ritorno. Viceversa, quando per una funzione non è specificato il tipo `void` per il valore di ritorno, nel blocco istruzioni della funzione è logico che sia presente per lo meno una istruzione `return`.

Il tipo `void` è usato anche per le funzioni che non assumono alcun parametro. Un esempio di funzione che non ha parametri e che non restituisce alcun valore è rappresentato da `mess_err` (Listato 7.8).

```
#include <stdio.h>

void mess_err( void );

main()
{
    int a, b, c;
```

e

```
mess_err();
```

sono dichiarazioni valide. Nel secondo caso, però, `mess_err` viene considerata una funzione il cui eventuale valore di ritorno è di tipo `int`, anche se in realtà non ha nessun valore di ritorno. Non è forse questo il caso anche della funzione `main`? Abbiamo continuato a definirla con:

```
main()  
{  
...  
}
```

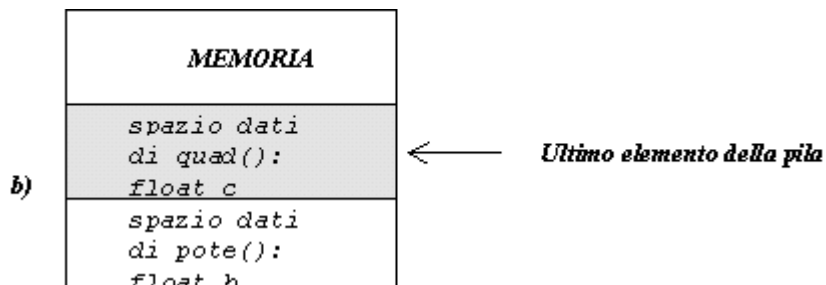
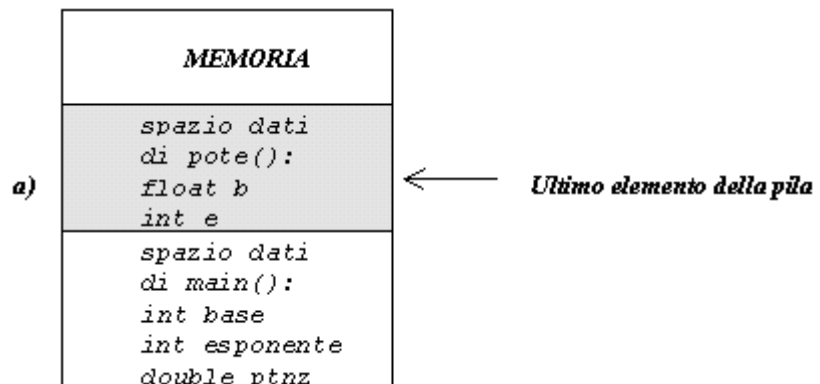
a indicare il fatto che non ritorna nessun valore – è quindi di tipo `void` – e che non assume parametri. Una equivalente (e forse anche più corretta) definizione di `main` potrebbe essere:

```
void main( void )  
{  
...  
}
```

Il fatto è che prima dello standard ANSI il C non prevedeva la parola chiave `void`, e oggi, per motivi di compatibilità, sono ammesse le due notazioni, con e senza `void`. Attualmente lo standard stabilisce che `main` sia implicitamente definita come funzione `void`, mentre in passato veniva comunemente definita di tipo `int`. Per evidenti motivi di leggibilità si consiglia caldamente di far uso di `void` tutte le volte che è necessario, soprattutto al fine di indicare che la funzione non ritorna nessun valore ■.

## 7.10 La scomposizione funzionale

funzione quad per determinare il quadrato del numero immesso e il sistema alloca spazio per i suoi dati (Figura 7.4b). Quando quad restituisce il controllo a pote la situazione ritorna a essere ancora quella di Figura 7.4a.



2. 2. rendere note alla funzione delle locazioni in cui andare a depositare le uscite.

Per saperne di più su questo secondo tipo di soluzione, si rimanda al Capitolo 9 sui puntatori ■.

## 7.11 Gestione di una sequenza

In questo paragrafo consideriamo il problema di far gestire all'utente una o più sequenze di interi mediante il seguente menu:

```
GESTIONE SEQUENZA
```

```
1. Immissione
2. Ordinamento
3. Ricerca completa
4. Ricerca binaria
5. Visualizzazione
0. fine
```

```
Scegliere una opzione:
```

Le opzioni possono essere scelte un numero di volte qualsiasi, finché non si seleziona la numero zero, che fa terminare il programma. Ovviamente, prima di tutto si deve scegliere la prima opzione per immettere la sequenza, ma successivamente questa possibilità può essere sfruttata per lavorare su altre sequenze.

Nel Listato 7.9 proponiamo il programma completo; dato che tutti gli algoritmi relativi sono stati visti nel Capitolo 5, adesso ci soffermiamo soltanto sull'uso delle funzioni e sul passaggio dei parametri.

L'array che conterrà la sequenza viene dichiarato come variabile globale:

La funzione ritorna un valore intero, che corrisponde alla posizione dove è stato reperito l'elemento. Considerazioni analoghe valgono per la funzione di ricerca binaria `ric_bin` ■.

```
#include <stdio.h>

#define MAX_ELE 1000      /* massimo numero di elementi */
int vet[MAX_ELE];         /* array che ospita la sequenza */

void gestione_sequenza( void );
int immissione( void );
void ordinamento( int );
int ricerca( int, int );
int ric_bin( int, int );
void visualizzazione( int );

main()
{
    gestione_sequenza();
}

void gestione_sequenza()
{
    int n;
    int scelta = -1;
    char invio;
    int ele, posizione;

    while(scelta != 0) {
```

```

        break;
    case 4: printf("Elemento da ricercare: ");
            scanf("%d", &ele);
            scanf("%c", &invio);
            posizione = ric_bin( n, ele );
            if(posizione != -1)
                printf("\nElemento %d presente in posizione
                        %d\n", ele, posizione);
            else
                printf("\nElemento non presente!\n");
            printf("\n\n Premere Invio per continuare...");
            scanf("%c", &invio);
            break;
    case 5: visualizzazione( n );
            break;
}
}
}

int immissione()
{
    int i, n;
    char invio;
    do {
        printf("\nNumero elementi: ");
        scanf("%d", &n);
    }
    while (n < 1 || n > MAX_ELE);

```

```

while (ele != vet[i] && i < n-1) i++;
return( i );
}

/* ricerca binaria */
int ric_bin( int n, int ele )
{
int i, alto, basso, pos;
alto = 0; basso = n - 1; pos = -1;
do {
    i = (alto+basso)/2;
    if(vet[i] == ele) pos = i;
    else if(vet[i] < ele) alto = i + 1;
        else basso = i - 1;
}
while(alto <= basso && pos == -1);
return( pos );
}

void visualizzazione( int n )
{
int i;
char invio;
for(i = 0; i < n; i++)
    printf("\n%d", vet[i]);
printf("\n\n Premere Invio per continuare...");
scanf("%c", &invio);
}

```



che dopo aver effettuato l'inizializzazione dell'array esso venga ordinato.

6. Modificare il programma del Listato 3.8, che calcola uno zero della funzione matematica  $f(x) = 2x^3 - 4x + 1$ , in modo che utilizzi una funzione per determinare i valori di  $f$ .

7. Progettare e realizzare una funzione che accetti in ingresso una data e restituisca in uscita il corrispondente giorno della settimana. La funzione deve effettuare anche i controlli di validità della data immessa.

8. Progettare e realizzare una funzione che, data una stringa  $s$ , calcoli il numero di occorrenze del carattere  $c$  all'interno della stringa.

9. Modificare la funzione dell'Esercizio 1 esercizio in modo che calcoli anche le potenze negative.

10. Esaminare i programmi di questo capitolo e scrivere per ognuno di essi la lista delle variabili globali all'intero programma e di quelle locali a ogni sottoprogramma.

11. Scrivere una funzione che calcoli, al variare di  $x$ , il valore dell'espressione:

$$3x^3 - \sqrt{\frac{x^2 + 3}{2}}$$

12. Scrivere una funzione che visualizzi sullo schermo

## 8.1 Direttive

Un compilatore traduce le istruzioni di un programma sorgente in linguaggio macchina. Generalmente il programmatore non è consapevole del lavoro del compilatore: usa delle istruzioni in linguaggio di alto livello per evitare le idiosincrasie del linguaggio macchina. Talvolta però è conveniente prendere coscienza dell'esistenza del compilatore per impartirgli delle direttive.

In C è possibile inserire in un codice sorgente tali direttive, dette più propriamente *direttive del preprocessore*. Il preprocessore è il modulo del compilatore che scandisce per primo il codice sorgente e interpreta le direttive prima della traduzione in codice macchina. Le direttive del preprocessore non fanno realmente parte della grammatica del linguaggio, ma estendono l'ambiente di programmazione del C includendo il compilatore. Tra esse troviamo `define` e `include`, che abbiamo introdotto già dal Capitolo 1. Secondo lo Standard ANSI l'elenco delle direttive del preprocessore C è il seguente:

<code>#define</code>	<code>#error</code>	<code>#include</code>
<code>#elif</code>	<code>#if</code>	<code>#line</code>
<code>#else</code>	<code>#ifdef</code>	<code>#pragma</code>
<code>#endif</code>	<code>#ifndef</code>	<code>#undef</code>

Tutte iniziano con il simbolo `#` e una linea di codice non ne deve contenere più di una. La linea

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

produrrebbe perciò un messaggio di errore da parte del preprocessore.

macro. Per esempio, il frammento di codice che segue definisce i valori UNO, DUE e TRE:

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

La sostituzione di macro è un processo vantaggioso che risparmia al programmatore un lavoro ripetitivo e tedioso. Se per esempio si volesse definire un messaggio di errore standard si potrebbe usare una soluzione del tipo:

```
#define MYERR "standard error on input\n"
...
printf(MYERR);
```

Il preprocessore procederebbe fedelmente a sostituire ogni occorrenza di MYERR con la stringa "standard error on input\n". Il compilatore vedrebbe istruzioni del tipo:

```
printf("standard error on input\n");
```

e il programmatore avrebbe la garanzia di una messaggistica di errore uniforme, precisa e a basso costo.

### ✓ NOTA

Non si ha sostituzione di macro se il nome di macro è usato all'interno di una stringa:

```
#define XYZ Buona Lettura
...
printf("XYZ");
```

L'ultima istruzione non visualizzerà la scritta Buona Lettura ma la stringa XYZ.

```

{
    printf("valore assoluto di -1 e 1: %d %d", ABS(-1), ABS(1));
    return 0;
}

```

All'atto della compilazione l'identificatore `a` introdotto nella definizione della macro viene sostituito con i valori `-1` e `1`. Le parentesi tonde che circondano `a` garantiscono la correttezza della sostituzione. Potrebbero infatti verificarsi dei casi in cui la sostituzione provoca risultati non attesi. Per esempio:

`ABS(10-20)`

senza l'uso delle parentesi tonde verrebbe convertito in

`10-20<0 ? -10-20 : 10 -20`

producendo un risultato falso!

### ✓ **NOTA**

La sostituzione di macro per la costruzione di funzioni presenta vantaggi e svantaggi. Il principale vantaggio è costituito dalle prestazioni. Infatti, producendo una espansione del codice localmente al punto in cui compare una occorrenza del `nome-macro` non si perde tempo nelle procedure di chiamata funzione. D'altra parte questa espansione locale della sequenza di caratteri duplica il codice e quindi aumenta le dimensioni del programma. Infine, occorre tenere presente che anche il programmatore C più esperto rimane talvolta vittima di sostituzioni non desiderate.

## 8.3 `#include`

## 8.4 `#error`

Quando il compilatore incontra la direttiva `#error` visualizza un messaggio di errore. La sintassi di questa direttiva è:

```
#error messaggio-errore
```

Il termine *messaggio-errore* non è racchiuso tra doppi apici. La direttiva `#error` è usata per la correzione degli errori (*debugging*). Oltre al messaggio indicato dalla direttiva, il compilatore potrebbe aggiungere ulteriori informazioni sullo stato della compilazione.

## 8.5 Direttive condizionali di compilazione

Una comoda funzionalità offerta dal preprocessore C è quella delle compilazioni condizionali. Ciò significa che alcune porzioni di codice possono essere selettivamente compilate, per esempio, per includere o meno personalizzazioni dell'applicativo. Questa funzionalità è usata frequentemente quando si devono fornire diverse versioni di uno stesso programma su piattaforme differenti, per esempio Unix, Windows, Macintosh.

Le compilazioni condizionali si ottengono con le direttive `#if`, `#else`, `#elif` ed `#endif`, il cui significato è molto semplice. Se l'espressione costante che segue `#if` è vera, la porzione di codice compresa tra `#if` ed `#endif` viene compilata, altrimenti sarà ignorata dal compilatore. La direttiva `#endif` viene usata per marcare la fine del blocco `#if`. La sua forma sintattica generale è:

```
#if espressione-costante  
    sequenza istruzioni  
#endif
```

Per esempio il semplice programma seguente:

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

```
#endif  
}
```

In questo caso MAX è inferiore a 99, motivo per cui la porzione di codice che segue `#if` non è compilata, mentre lo è il codice che segue `#else`. In altre parole, sarà mostrato il messaggio compilato per array piccoli.

Si osservi come il termine `#else` venga usato sia per marcare la fine del blocco `#if` sia per segnare l'inizio del blocco `#else`. Ciò è necessario perché in una direttiva `#if` può essere presente un solo termine `#endif`.

La direttiva `#elif` significa “else if” ed è usata per stabilire una catena di “if-else-if” che realizza una compilazione condizionale multipla. Se l'espressione è vera, la sequenza di istruzioni è compilata e nessun'altra sequenza `#elif` è valutata; altrimenti si controlla la prossima serie. La forma generale è:

```
#if espressione  
    sequenza istruzioni  
#elif espressione 1  
    sequenza istruzioni  
#elif espressione 2  
    sequenza istruzione  
...  
#elif espressione N  
    sequenza istruzioni  
#endif
```

Per esempio, il seguente frammento usa il valore COUNTRY per definire la moneta corrente:

```
#define US 0  
    #define ENGLAND 1  
    #define ITALY 2  
  
#define COUNTRY ITALY  
  
#if COUNTRY==ITALY
```

```

main()
{
    int i = 100;

    #if defined DEBUG
        printf("la variabile i vale: %d\n", i);
    #endif
}

```

Facendo precedere la parola chiave `define` dal simbolo `!` si ottiene una compilazione condizionale nel caso in cui la macro non sia stata definita.

Altre due direttive per la compilazione condizionale sono `#ifndef`, che significa “if defined”, e `#ifdef`, che significa “if not defined”. La sintassi generale è:

```

#ifndef nome-macro
    sequenza istruzioni
#endif

```

Se il *nome-macro* è stato precedentemente definito da una `#define`, allora la sequenza di istruzioni verrà compilata.

### ✓ NOTA

Usare `#ifndef` è equivalente a usare `#if` insieme all’operatore `defined`.

La sintassi di `#ifdef` è:

```

#ifdef nome-macro

```

`#undef nome-macro`

Vediamo un esempio:

```
#define LEN 100
#define WIDTH 100

char array[LEN][WIDTH];

#undef LEN
#undef WIDTH
/* da questo punto in poi le costanti simboliche
   LEN e WIDTH non sono più definite */
```

La direttiva `#undef` è usata per confinare la definizione di macro in precise porzioni di codice.

## 8.7 `#line`

La direttiva `#line` è usata per modificare il contenuto di due identificatori predefiniti dal compilatore i cui nomi sono `__LINE__` e `__FILE__`. La sintassi di questa direttiva è:

```
#line numero "nomefile"
```

dove `numero` è un qualsiasi intero positivo e `nomefile` è un qualunque identificatore valido di file. Il `numero` diviene il numero della linea di codice attuale del sorgente, e il nome del file diviene il nome del file sorgente. Il nome del file è opzionale.

La direttiva `#line` viene usata prevalentemente per il debug di applicazioni. L'identificatore `__LINE__` è un intero, mentre `__FILE__` è una stringa. Vediamo ora un esempio:



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

```
#define makestring(s) # s
```

```
main()
{
    printf(makestring(Mi piace il C));
}
```

Il preprocessore trasforma l'istruzione

```
printf(makestring(Mi piace il C));
```

in

```
printf("Mi piace il C");
```

L'operatore ## è invece usato per concatenare due sequenze di caratteri in una #define:

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

```
#define concat(a, b) a##b
```

```
main()
{
    int xy = 10;
    printf("%d", concat(x, y));
}
```

Il preprocessore trasforma la curiosa istruzione

La macro `__DATE__` è una stringa costata secondo il formato *mes/giorn/anno* che riporta la data dell'ultima compilazione del codice sorgente.

L'ora dell'ultima compilazione invece è conservata in `__TIME__`, stringa che assume la forma *ora:minuti:secondi*.

Infine, la macro `__STDC__` contiene la costante decimale 1. Ciò significa che l'implementazione del compilatore è conforme allo standard. Se invece riporta un qualsiasi altro numero l'implementazione non è standard.

## 8.11 Esercizi

1. Realizzare la macro `ABS(X)` che calcola il valore assoluto di `X`.
2. Realizzare la macro `CUBO(X)` che verrà espansa al valore di `X` elevato alla terza potenza. Quali valutazioni possiamo fare, sia in termini di efficienza sia più in generale, scegliendo di calcolare il valore assoluto e il cubo di un numero con una funzione o con una macro?

3. Date le macroistruzioni

```
#define DIM 100;  
#define VERO (a>100);
```

quale errore abbiamo probabilmente commesso nello scrivere le macroistruzioni (che comunque verranno accettate ed espande dal precompilatore)? Come verrebbero espande le seguenti istruzioni?

1. `n = DIM;`
2. `float array[DIM];`
3. `while VERO`  
    `Calcola();`

In quale caso si avrà un errore in fase di compilazione? In quale caso un effetto indesiderato? E in quale tutto andrà

L'operatore `&`, introdotto con la funzione `scanf`, restituisce l'indirizzo di memoria di una variabile. Per esempio l'espressione `&a` è un'espressione il cui valore è l'indirizzo della variabile `a`. Un indirizzo può essere assegnato solo a una speciale categoria di variabili dette *puntatori*, le quali sono appunto variabili abilitate a contenere un indirizzo. La sintassi di definizione è

```
tipo_base *var_punt;
```

dove `var_punt` è definita come variabile di tipo "puntatore a `tipo_base`"; in sostanza `var_punt` è creata per poter mantenere l'indirizzo di variabili di tipo `tipo_base`, che è uno dei tipi fondamentali già introdotti: `char`, `int`, `float` e `double`.

Il tipo puntatore è un classico esempio di tipo derivato; infatti, non ha senso parlare di tipo puntatore in generale, ma occorre sempre specificare a quale tipo esso punta. Per esempio, in questo caso:

```
int    a;
char   c;

int    *pi;
char   *pc;

pi = &a;
pc = &c;
```

si ha che `pi` è una variabile di tipo puntatore a `int`, e `pc` è una variabile di tipo puntatore a `char`. Le variabili `pi` e `pc` sono inizializzate rispettivamente con l'indirizzo di `a` e di `c`.

a = 5      c = x

Vediamo un altro esempio:

```
char  c1, c2;
char  *pc;
...
c1 = 'a';
c2 = 'b';
printf(" c1 = %c, c2 = %c \n", c1, c2);

pc = &c1;          /* pc contiene l'indirizzo di c1 */
c2 = *pc;          /* c2 contiene il carattere 'a' */
printf(" c1 = %c, c2 = %c \n", c1, c2);
```

Dopo l'assegnazione `pc=&c1`; i nomi `c1` e `*pc` sono perfettamente equivalenti (*alias*), e si può accedere allo stesso oggetto creato con la definizione `char c1` sia con il nome `c` sia con l'espressione `*pc`. L'effetto ottenuto con l'assegnazione `c2=*pc` si sarebbe ottenuto, equivalentemente, con l'assegnazione

```
c2 = c1;
```

Un ulteriore esempio di uso di puntatori e dell'operatore di indirezione, riferiti a elementi di un array, è il seguente:

```
int  buf[2];
int  *p;
...
p  = &buf[1];
*p = 4;
```

si ha che le due assegnazioni `s=&buf[0]` e `s=buf` sono perfettamente equivalenti. Infatti in C il nome di un array, come nel nostro caso `buf`, è una costante – si noti bene: una costante, non una variabile! – il cui valore è l'indirizzo del primo elemento dell'array. Allora, come gli elementi di un array vengono scanditi per mezzo dell'indice, equivalentemente si può avere accesso agli stessi elementi per mezzo di un puntatore. Per esempio consideriamo il seguente codice:

```
char  buf[100];
char  *s;
...
s = buf;
buf[7] = 'a';
printf("buf[7] = %c\n", buf[7]);

*(s + 7) = 'b';
printf("buf[7] = %c\n", buf[7]);
```

Si ha che `s` e `buf` sono due sinonimi, con la differenza che `s` è una variabile puntatore a carattere, mentre `buf` è una costante. Incrementato di 7 il valore di `s` si ottiene un valore corrispondente all'indirizzo dell'ottavo elemento, al quale si accede per mezzo dell'operatore di indirezione. Il valore del puntatore `s` è incrementato di 7 unità, cioè di 7 volte la dimensione dell'oggetto corrispondente al tipo base del puntatore (Figura 9.1).

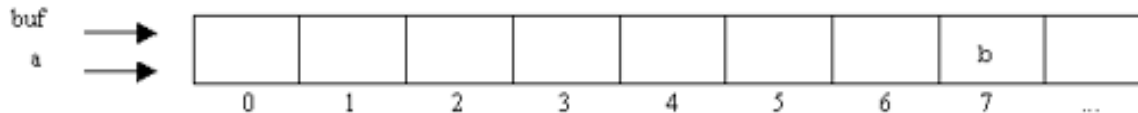


Figura 9.1 Si può far riferimento a `buf[7]` scrivendo `*(s+7)`

il compilatore non avrebbe segnalato alcun errore, ma avremmo avuto problemi in esecuzione perché si sarebbe inizializzata con 'K' una regione di memoria al di là del limite allocato con `buf`. Che si usi una variabile puntatore o si faccia riferimento alla notazione con indice, è sempre dovere del programmatore assicurarsi che le dimensioni di un array vengano rispettate.

## 9.3 Aritmetica dei puntatori

Un puntatore contiene un indirizzo e le operazioni che possono essere compiute su un puntatore sono perciò quelle che hanno senso per un indirizzo. Le uniche operazioni ammissibili sono dunque: l'incremento, per andare da un indirizzo più basso a uno più alto, e il decremento, per andare da un indirizzo più alto a uno più basso. Gli indirizzi sono per la memoria quello che sono gli indici per un array.

Gli operatori ammessi per una variabile di tipo puntatore sono:

`+`   `++`   `-`   `--`

Ma qual è l'esatto significato dell'incremento o decremento di un puntatore? Non pensi il lettore che il valore numerico del puntatore corrispondente a un indirizzo venga incrementato come una qualunque altra costante numerica. Per esempio, se `pc` vale 10, dove `pc` è stato dichiarato:

```
char *pc;
```

non è detto che `pc++` valga 11 !

Nell'aritmetica dei puntatori quello che conta è il tipo base. Incrementare di 1 un puntatore significa far saltare il puntatore alla prossima locazione corrispondente a un elemento di memoria il cui tipo coincide con quello base. Per esempio, in:

```
int a[10];
```

```

int v1[10];
int v2[10];
int i;
int *p;

i = &v1[5] - &v1[3];  /* 1 ESEMPIO */
printf("%d\n", i);    /* i vale 2 */

i = &v1[5] - &v2[3];  /* 2 ESEMPIO */
printf("%d\n", i);    /* il risultato è indefinito */

/* 3 ESEMPIO */
p = v2 - 2;           /* dove va a puntare p ? */

```

1. 1. *sottrazione tra indirizzi dello stesso vettore:*

```
i = &v1[5] - &v1[3];
```

corrispondente a un caso perfettamente legale;

2. 2. *sottrazione tra indirizzi di array diversi:*

```
i = &v1[5] - &v2[3];
```

corrispondente a un caso il cui risultato non è prevedibile;

3. 3. *sottrazione di una costante da un indirizzo ma nella direzione sbagliata:*

```
#include <stdio.h>

void scambia(int, int);

main()
{
    int x, y;

    x = 8;
    y = 16;
    printf("Prima dello scambio\n");
    printf("x = %d, y = %d\n", x, y);

    scambia(x, y);

    printf("Dopo lo scambio\n");
    printf("x = %d, y = %d\n", x, y);
}

/* Versione KO di scambia */
void scambia(int a, int b)
{
    int temp;

    temp = a;

    a = b;
    b = temp;
```



```

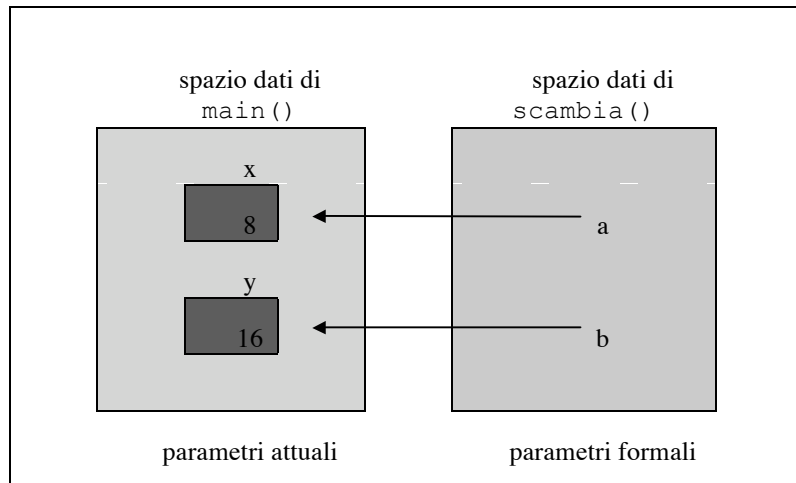
}

/* Versione OK di scambia */
void scambia(int *a, int *b)
{
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;
}

```

Listato 9.2 Ancora sullo scambio di valori



L'array dell'esempio è una stringa, cioè un array di `char` che termina con il carattere terminatore `\0`. Con l'inizializzazione

```
char str[] = "BATUFFO";
```

il primo elemento dell'array di caratteri `str` è inizializzato a puntare al primo elemento della costante di tipo stringa `"BATUFFO"`.

L'accorgimento di valutare una stringa per mezzo del puntatore `char *` è particolarmente utile nella scrittura di funzioni che manipolano stringhe. Nell'esempio la funzione `strlen` conta il numero di caratteri (escluso `\0`) di una stringa:

```
int strlen( char *p)
{
    int i = 0;
    while (*p++) i++;
    return i;
}
```

La funzione pone il contatore `i` a zero e comincia a contare caratteri finché non trova il carattere nullo. Una implementazione alternativa di `strlen` che usa la sottrazione di puntatori è:

```
int strlen( char *p )
{
    char *q = p;
    while ( *q++);
    return (q-p-1);
}
```

Le funzioni di manipolazione stringa gestiscono le stringhe sempre per mezzo di puntatori a carattere. Il C fornisce un

```
int strlen(const char *string);
```

Conta il numero di caratteri di *string*, escluso il carattere nullo.

```
char *strchr(const char *string, int c);
```

Ritorna il puntatore alla prima occorrenza in *string* del carattere *c*.

```
char *strrchr(const char *string, int c);
```

Ritorna il puntatore all'ultima occorrenza del carattere *c* nella stringa *string*.

```
char *strpbrk(const char *string1, const char *string2);
```

Ritorna un puntatore alla prima occorrenza della stringa *string2* in *string1*.

```
int strspn(const char *string1, const char *string2);
```

Trova la posizione del primo carattere in *string1* che non appartiene all'insieme di caratteri di *string2*.

```
char *strtok(char *string1, const char *string2);
```

Trova la prossima sequenza di caratteri (*token*) circonscritta dai caratteri *string2* nella stringa *string1*.

Il lettore può facilmente verificare l'uso di puntatori a `char` che si ha nelle funzioni di manipolazione stringa. Si osservi inoltre l'uso della parola chiave `C const`. Essa sta a indicare che, anche se l'indirizzo `char *` è passato alla funzione, la funzione non può andare a modificare le locazioni di memoria puntate da tale indirizzo, può solamente andare a leggere le locazioni puntate da quell'indirizzo. Con tale precisazione, semplicemente leggendo il prototype della funzione si capisce quali sono le stringhe che vengono modificate dalla corrispondente funzione di manipolazione.

Il valore di puntatore nullo NULL è una costante universale che si applica a qualsiasi tipo di puntatore (puntatore a char, a int ecc.). Generalmente la sua definizione è:

```
#define NULL 0
```

ed è contenuta in `<stdio.h>`.

Come detto, in C la memoria è allocata dinamicamente per mezzo delle funzioni di allocazione `malloc` e `calloc` che hanno le seguenti specifiche:

```
void *malloc(int num); /* num: quantità di memoria da allocare */

void *calloc(int numele, int eledim);
                /* numele: numero di elementi; eledim:
                quantità di memoria per ogni elemento */
```

Sia `malloc` sia `calloc` ritornano un puntatore a carattere che punta alla memoria allocata. Se l'allocazione di memoria non ha successo – o perché non c'è memoria sufficiente, o perché si sono passati dei parametri sbagliati – le funzioni ritornano il puntatore NULL.

Per stabilire la quantità di memoria da allocare è molto spesso utile usare l'operatore `sizeof`, che si applica nel modo seguente:

```
sizeof( espressione )
```

nel qual caso restituisce la quantità di memoria richiesta per memorizzare *espressione*, oppure

```
sizeof( T )
```

essere rappresentato come segue ■.



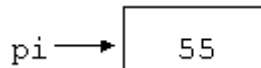
La scatola vuota simboleggia lo spazio riservato dall'intero. Per poter utilizzare le funzioni di allocazione è necessario includere la libreria `malloc.h` e/o `stdlib.h`; in implementazioni del C meno recenti la libreria da includere è `stddef.h`:

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

Si accede a un oggetto dinamico tramite un puntatore e l'operatore di indirezione. Così, nell'esempio, si accede all'intero puntato da `pi` con il nome `*pi`. Per esempio:

```
*pi = 55;
```

Graficamente l'effetto della precedente assegnazione può essere rappresentato come segue.



Lo stesso valore del puntatore può essere assegnato a più di una variabile puntatore. In tal modo si può far riferimento a un oggetto dinamico con più di un puntatore. Un oggetto cui si fa riferimento con due o più puntatori possiede degli *alias*. Per esempio, il risultato dell'assegnazione

```
qi = pi;
```

C è un linguaggio tipicamente usato per la programmazione di sistema, cioè per la programmazione di dispositivi hardware. Un classico problema della programmazione di sistema è l'indirizzamento diretto della memoria. Usando i puntatori e il cast è molto semplice fare riferimento a un indirizzo assoluto di memoria. Per esempio, supponiamo che `pt` sia un puntatore di tipo `T *`; questo puntatore lo si fa puntare alla locazione in memoria `0777000` nel seguente modo:

```
pt = (T *) 0777000;
```

Questa tecnica è comunemente usata nella costruzione di driver. Per esempio, nel caso del sistema operativo MS-DOS è pratica comune accedere direttamente alla memoria video per la gestione degli output su video. Occorre però tenere presente che la maggior parte dei sistemi operativi impedisce al programmatore la gestione diretta dell'hardware. È infatti il sistema operativo che offre l'interfaccia verso l'hardware, mettendo a disposizione della funzioni le cui invocazioni sono dette *chiamate di sistema*.

## 9.7 Gestione di una sequenza

Nell'ultimo paragrafo del Capitolo 7 abbiamo esaminato un programma per la gestione di una sequenza tramite un menu con le opzioni di immissione, ordinamento, ricerca completa e ricerca binaria. In quella sede l'array che conteneva la sequenza era una variabile generale cui tutte le funzioni accedevano direttamente.

Adesso presentiamo le modifiche necessarie perché il tutto avvenga mediante un array locale alla funzione `gestione_sequenza` e il passaggio del suo indirizzo alle altre funzioni. Innanzitutto le dichiarazioni devono essere fatte in modo da includere il parametro puntatore all'array:

```
int immissione( int, int * );  
void ordinamento( int, int * );  
int ricerca( int, int , int * );
```

- \*1. Scrivere un programma che esegua la scansione e la visualizzazione di un vettore di interi.
- \*2. Scrivere un programma che esegua la scansione e la visualizzazione di un vettore di stringhe.
- \*3. Scrivere una funzione che ritorni un puntatore alla prima occorrenza della stringa `t` in una stringa `s`. Se la stringa `t` non è contenuta in `s` allora la funzione ritorna un puntatore `NULL`.
- \*4. Scrivere almeno tre differenti versioni di una funzione che effettui la copia di una stringa su un'altra.
- \*5. Scrivere un programma che prenda in ingresso la dimensione di un buffer e la allochi dinamicamente.
- \*6. Modificare il programma `gestione_sequenza` cui si fa riferimento nell'ultimo paragrafo di questo capitolo in modo che la funzione di immissione sequenza non ritorni nessun valore ma abbia in ingresso il puntatore alla variabile intera `n` di `gestione_sequenza` per poterla modificare.

## 10.1 Iterazione e ricorsione

Quando si vuole ripetere l'esecuzione di un gruppo di istruzioni, un'alternativa alle strutture iterative come `for` o `while` è rappresentata dalle *funzioni ricorsive*. Una funzione si dice ricorsiva se chiama se stessa direttamente o indirettamente.

Per alcune classi di problemi le soluzioni ricorsive sono eleganti, sintetiche e più chiare delle altre. Un esempio di questo fatto si può trovare nel calcolo del fattoriale, esaminato nel Capitolo 3. Ricordiamo che il fattoriale  $n!$  del numero  $n$ , dove  $n$  è un intero maggiore o uguale a 2, è dato da:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot \dots \cdot 2 \cdot 1$$

Per restituire il fattoriale al programma chiamante la funzione utilizzerà l'istruzione `return`:

```
return (n*fat (n-1)) ;
```

che ritorna il valore di `n` moltiplicato per il fattoriale di `n-1`. Il calcolo del fattoriale di `n-1` lo si ottiene invocando ancora una volta la stessa funzione `fat` e passandole come argomento `n-1`; in questo modo si ottiene l'iterazione. Il ciclo, a un certo punto, deve terminare, per cui è necessaria una condizione di fine, che impostiamo così:

```
if (n==0)
    return (1) ;
else
    return (n*fat (n-1)) ;
```

Quando il valore passato alla funzione è uguale a 0, non ci sono valori da considerare e `fat` ritorna 1 (0!).

Confrontiamo dunque la funzione ricorsiva `fat` con il programma iterativo del Capitolo 3.

procedura ricorsiva

procedura iterativa

```
-----
fat (int n)
{
  if (n==0)
    return (1) ;
  else
    return (n*fat (n-1)) ;
}
```

```
if (n==0)
  fat = 1 ;
else
  for (fat=n; n>2; n--)
    fat = fat*(n-1) ;
```



l'esempio del calcolo di  $4!$  (Figura 10.2).

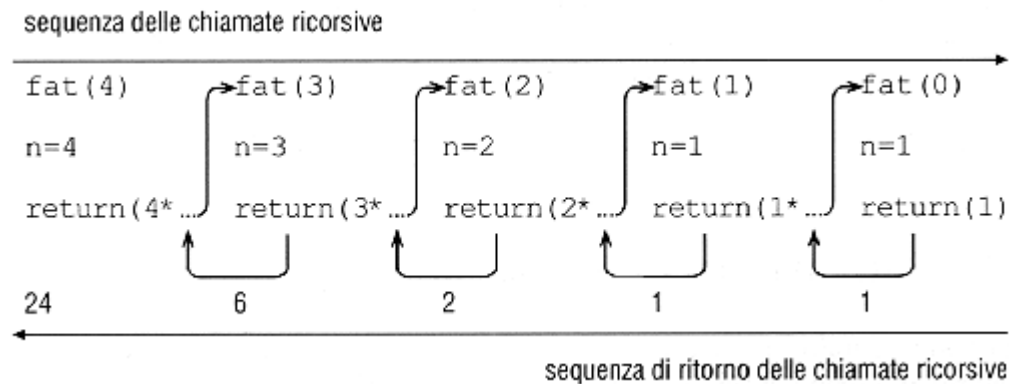


Figura 10.2 Ambienti creati dalla funzione `fat` con ingresso  $n=4$

Osservando la figura possiamo vedere che a ogni chiamata di `fat` viene creata una nuova variabile `n` locale a quell'ambiente. Quando termina il ciclo delle chiamate, ogni ambiente aperto si chiude e passa all'ambiente precedente il valore calcolato.

#### ✓ NOTA

La zona di memoria riservata alle chiamate viene gestita con la logica di una pila, concetto che tratteremo in modo specifico più avanti, quando parleremo di strutture dati. ■ A ogni invocazione di `fat`, il sistema alloca uno spazio di memoria libero in testa alla pila riservato al suo parametro formale `n`. In Figura 10.2 si osserva come la sequenza di chiamate e il ritorno delle stesse vengano gestiti come una pila, in cui l'ultimo elemento è il primo a essere eliminato. In altre parole, si tratta di una struttura a pila, o stack, che gestisce le chiamate e i ritorni.

Dati due oggetti e1, e2 si possono avere solamente due permutazioni; se gli oggetti sono tre le permutazioni semplici diventano sei; se gli oggetti sono quattro (e1 e2 e3 e4), si hanno le seguenti 24 possibilità:

e1 e2 e3 e4	e1 e2 e4 e3	e2 e1 e3 e4	e1 e2 e4 e3
e3 e1 e2 e4	e3 e1 e4 e2	e1 e3 e2 e4	e1 e3 e4 e2
e2 e3 e1 e4	e2 e3 e4 e1	e3 e2 e1 e4	e3 e2 e4 e1
e1 e4 e2 e3	e1 e4 e3 e2	e2 e4 e1 e3	e2 e4 e3 e1
e3 e4 e1 e2	e3 e4 e2 e1	e4 e1 e2 e3	e4 e1 e3 e2
e4 e2 e1 e3	e4 e2 e3 e1	e4 e3 e1 e2	e4 e3 e2 e1

In generale, il numero di permutazioni  $P$  di  $n$  oggetti è dato  $P_n = n!$ , da cui risultano appunto, nel nostro caso,  $4! = 24$  possibilità distinte.

Questo è un problema che abbiamo già risolto. Se desideriamo conoscere il numero di permutazioni di 13 oggetti è sufficiente mandare in esecuzione l'ultimo programma del paragrafo precedente, con l'accortezza di utilizzare un tipo dati adeguato, per scoprire che sono: 6 227 020 800.

Dati  $n$  oggetti distinti e detto  $k$  un numero intero positivo minore o uguale a  $n$ , si chiamano invece *disposizioni semplici* di questi  $n$  oggetti i gruppi distinti che si possono formare in modo che ogni gruppo contenga soltanto  $k$  oggetti e che differisca dagli altri o per qualche oggetto, o per l'ordine in cui gli oggetti stessi sono disposti. Le disposizioni di quattro oggetti ( $n=4$ ) presi uno a uno ( $k=1$ ) sono dunque i gruppi che contengono un solo oggetto:

e1            e2            e3            e4

cioè in totale quattro. Le disposizioni di quattro oggetti presi due a due ( $k=2$ ) sono invece 12:

e1 e2	e2 e1	e3 e1	e4 e1
e1 e3	e2 e3	e3 e2	e4 e2
e1 e4	e2 e4	e3 e4	e4 e3

Il calcolo delle disposizioni usa la formula generale:

$$D_{n,k} = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot (n-k+2) \cdot (n-k+1)$$

```

int dispo(int, int, int);

main()
{
    int n, k;

    printf("Disposizioni semplici di k su n oggetti\n");
    printf("Inser. n: \t");
    scanf("%d", &n);
    printf("Inser. k: \t");
    scanf("%d", &k);
    printf("Le dispos. sempl. di %d su %d sono: %d\n", k, n, dispo(k, n, n));
}

int dispo(int k, int n, int m)
{
    if(n==m-k)
        return(1);
    else
        return(n*dispo(k, n-1, m));
}

```

### Listato 10.2 Calcolo delle disposizioni semplici

Osserviamo che il calcolo delle disposizioni è simile a quello del fattoriale. In particolare, le disposizioni di  $n$  elementi presi  $n$  a  $n$  sono proprio pari a  $n!$ :

$$D_{n,n} = n!$$

Nel caso fosse  $n=4$  e  $k=4$  avremmo:

le disposizioni  $D_{n,k}$  e fat per calcolare il fattoriale, passando k come numero di elementi:

```
comb(int k, int n)
{
    return(dis(k, n)/fat(k));
}
```

Nel Listato 10.3 viene presentato il programma relativo al calcolo delle combinazioni semplici.

```
/* Calcolo delle combinazioni semplici di n oggetti presi k a k */

#include <stdio.h>

int comb(int, int);
int dispo(int , int, int);
int fat(int);

main()
{
    int n, k;

    printf("Combinazioni semplici di k su n oggetti\n");
    printf("Inserire n: \t");
    scanf("%d", &n);
    printf("Inserire k: \t");
    scanf("%d", &k);
    printf("Le combin. sempl. di %d su %d sono: %d\n", k, n, comb(k, n));
}
```

```

comb(int k, int n)
{
    return(dispo(k, n, n)/dispo(k, k, k));
}

```

Una seconda possibilità si ottiene sfruttando la funzione `dispo2` che, come abbiamo visto in precedenza, utilizzava a sua volta `fat` per calcolare le disposizioni:

```

/* Calcolo delle combinazioni semplici utilizzando
    dispo2() e fat() */

comb(int k, int n)
{
    return(dispo2(k, n, n)/fat(k));
}

/* Calcolo delle disposizioni semplici utilizzando fat() */

int dispo2(int k, int n)
{
    return(fat(n)/fat(n-k));
}

```

Così facendo abbiamo decomposto le formule risolutive di combinazioni e disposizioni rimandando il problema al calcolo del fattoriale. Attenzione, comunque: se  $n$  e  $k$  superano un certo valore, che dipende dalla dimensione degli `int` e dei `long int` dello specifico compilatore, si devono utilizzare funzioni e parametri di tipo `float`.

## 10.4 La successione di Fibonacci

```

printf("Successione di Fibonacci f(0)=1 f(1)=1 f(n)=f(n-1)+f(n-2)");
printf("\nInserire n: \t");
scanf("%d", &n);
printf("Il termine della successione di argomento %d è: %d\n", n, fibo(n));
}

long int fibo(int n)
{
if(n==0)      return(0);
else if(n==1) return(1);
    else      return(fibo(n-1)+fibo(n-2));
}

```

Listato 10.4 Calcolo della successione di Fibonacci

### ✓ NOTA

I programmi che sfruttano la ricorsività sono in larga misura inefficienti, tanto in termini di occupazione di memoria quanto in termini di velocità di esecuzione. La ragione principale è che nelle funzioni ricorsive spesso vengono ripetuti calcoli già eseguiti in precedenza. Si riprenda l'esempio delle combinazioni: nel caso di  $\text{comb}(5, 2)$ ,  $\text{comb}(2, 1)$  è calcolato tre volte!

Si può facilmente verificare il peso di queste ripetizioni inserendo, all'interno delle funzioni esaminate, prima della chiamata ricorsiva, una `printf` che visualizzi il valore delle variabili trattate dalle funzioni stesse.

## 10.5 Ordinamento con *quicksort*

Nel Capitolo 5 abbiamo esaminato due metodi di ordinamento: quello ingenuo e *bubblesort*; quest'ultimo è stato

```

int i;
for(i=0; i<N; i++) {
    printf("\nImmettere un intero n.%d: ",i);
    scanf("%d", &v[i]);
}

quick(0,N-1);          /* Chiamata della procedura quick */

for(i=0; i<N; i++)      /* Sequenza ordinata */
    printf("\n%d", v[i]);
putchar('\n');
}

/* Procedura ricorsiva "quick" */
void quick(int sin, int des)
{
    int i, j, media;
    media= (v[sin]+v[des]) / 2;
    i = sin;
    j = des;

    do {
        while(v[i]<media) i = i+1;
        while(media<v[j]) j = j-1;
        if(i<=j) {
            // swap v[i] and v[j]

```

valore minore di questo nella parte bassa del vettore, tutti quelli con valore maggiore nella parte alta. A ogni passo del processo quicksort ordina quindi attorno al pivot gli elementi del blocco del vettore esaminato. Man mano che i blocchi diventano sempre più piccoli il vettore tende a essere completamente ordinato.

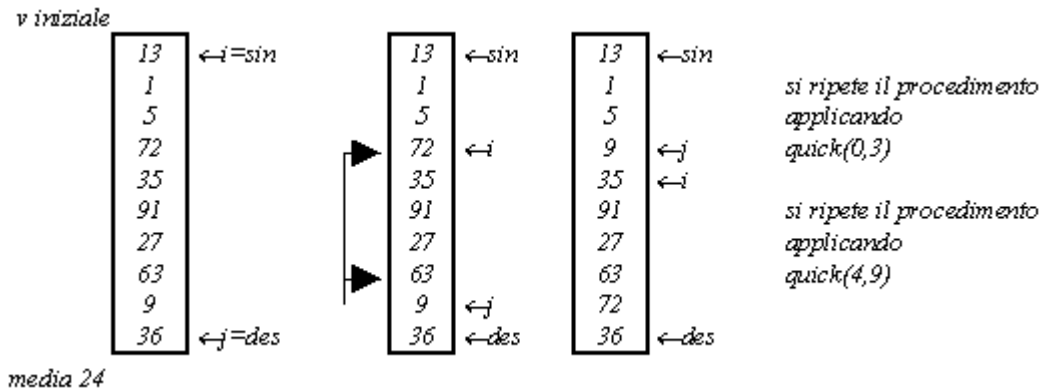


Figura 10.3 Fasi di lavoro di quicksort applicato a un vettore di 10 interi

La procedura *quick* del Listato 10.5 viene inizialmente applicata sull'intero vettore (dall'elemento di indice 0 a quello di indice  $N-1$ ); è calcolato il valore medio *media* tra *vet[sin]* e *vet[des]* che alla prima chiamata è 24, la *media* appunto tra i due estremi del vettore: 13 (*vet[0]*) e 36 (*vet[9]*) (Figura 10.3). Successivamente *quick* scorre gli elementi del blocco a partire dal basso, fintantoché si mantengono minori della *media*:

```
while (v[i]<media) i = i+1;
```

Analogamente, la procedura scorre gli elementi del blocco a partire dall'alto, fintantoché si mantengono maggiori della *media*:

```
while (media<v[i]) i = i-1;
```



di *time*, perché il numero effettivo dei confronti dipende comunque dai particolari valori del vettore da ordinare.

Questa valutazione è stata ottenuta in base a considerazioni matematiche di tipo teorico ma molti programmatori trovano più semplice confrontare i metodi di ordinamento misurando direttamente il tempo impiegato da ognuno di loro per ordinare il medesimo vettore. Anche in questo caso si mantiene la stessa classifica nelle prestazioni; per esempio, con 1000 elementi bubblesort è in media 70 volte più lento di quicksort. Notiamo ovviamente che il tempo assoluto occorrente dipende dall'elaboratore su cui si sta lavorando.

Il vantaggio della ricorsione risiede nella possibilità di realizzare algoritmi sintetici ed eleganti. Inoltre, la scrittura di funzioni ricorsive permette al programmatore di verificare a fondo le proprie conoscenze e possibilmente di migliorarle. Nei capitoli successivi vedremo l'uso della ricorsività nell'ambito delle strutture dati, come liste e alberi, dove a volte la soluzione migliore, per la natura stessa dei problemi che si affronteranno, sarà appunto quella ricorsiva.

## 10.6 Mutua ricorsività

Concludiamo questo capitolo introducendo brevemente il concetto di *mutua ricorsività*.

In una funzione  $x$  può essere presente una chiamata a una funzione  $y$ . Se  $y$  contiene a sua volta una chiamata a  $x$ , si ha una *ricorsività indiretta*. In generale, in questo processo di mutua ricorsività possono essere coinvolte più di due funzioni che indirettamente invocano se stesse tramite una successione di chiamate: dalla funzione  $x$  viene chiamata  $y$ , da  $y$  viene chiamata  $t$ , da  $t$  viene chiamata  $z$ , da  $z$  ancora  $x$  ecc. Nella Figura 10.4 viene mostrato un esempio di ricorsività indiretta.

```
x (int a) ←  
{  
  float m;  
  int n;  
  ...  
  m = y (n);
```

\* 6. Risolvere l'Esercizio 5 con una funzione ricorsiva.

\* 7. Scrivere una funzione ricorsiva che calcoli il *massimo comune divisore* di due numeri interi positivi utilizzando l'algoritmo euclideo per cui:

$$\text{MCD}(t, k) = \begin{cases} t & \text{se } k = 0 \\ \text{MCD}(k, t) & \text{se } k > t \\ \text{MCD}(k, t \% k) & \text{altrimenti} \end{cases}$$

8. Scrivere una funzione ricorsiva che calcoli il *massimo comune divisore* di due numeri interi positivi ricordando che

$$\text{MCD}(t, k) = \begin{cases} \text{MCD}(t-k, k) & \text{se } t > k \\ t & \text{se } t = k \\ \text{MCD}(k, t) & \text{se } t < k \end{cases}$$

9. Confrontare gli algoritmi dei due precedenti esercizi e dire qual è più veloce; motivare la risposta.

10. Scrivere una versione ricorsiva della ricerca binaria su un vettore ordinato.

11. Scrivere una funzione ricorsiva che calcoli

$$f(x, n) = 1 \cdot x + 2 \cdot x^2 + 3 \cdot x^3 + \dots + (n-1) \cdot x^{n-1} + n \cdot x^n$$

con  $x$  float e  $n$  int richiesti all'utente.

12. Scrivere un programma che calcoli i numeri ottenuti in base alla seguente definizione:

$$a_1 = 3;$$

$$a_2 = 7;$$

$$a_n = 2 \cdot a_{n-1} - 3 \cdot a_{n-2} \quad \text{per } n \geq 3$$

```
extern int codice_errore;
```

Alcune di queste dichiarazioni sono anche delle definizioni, cioè hanno l'effetto di creare la relativa regione di memoria, detta "oggetto"; esattamente si tratta di:

```
char    c;  
int     conta = 0;  
char    *animale = "Anatra";  
double  cubo(float c) { return c*c*c }
```

Le prime due riservano una determinata zona di memoria della dimensione di un `char` e di un `int` cui associare rispettivamente il nome `c` e il nome `conta`. L'ultima associa al nome `cubo` una porzione di programma, corrispondente alle istruzioni che formano la funzione `cubo`. Le dichiarazioni:

```
double pot(float, int);  
extern int codice_errore;
```

invece non corrispondono ad alcuna allocazione di memoria. Entrambe le istruzioni introducono un nome, ma rimandano la definizione del nome, cioè l'allocazione dell'oggetto corrispondente, a un'altra parte del programma. Nel Capitolo 7 abbiamo già visto la differenza tra dichiarazione e definizione di una funzione; questa differenza può esistere anche tra variabili che non siano funzioni, come nel caso della variabile `codice_errore` il cui tipo è `int`, e la cui definizione è `extern` (esterna) cioè definita in un qualche altro file, diverso da quello in cui è dichiarata per mezzo dell'istruzione ■.

```
extern int codice_errore;
```

Si ricordi che tutte le dichiarazioni che specificano un valore per il nome che introducono sono sempre anche delle definizioni. Per esempio:

```
int     conta = 0;
```

operazioni tipiche di un intero, cioè somma, sottrazione, divisione, moltiplicazione e così via. Allora, anche se raramente usato, non è un errore assegnare un valore negativo a una variabile di tipo `char`:

```
char raro = -1;
```

Il tipo `int`, invece, è un intero la cui dimensione è di solito pari alla *parola* (*word*) della macchina; dunque nella maggioranza dei casi una variabile di tipo `int` occupa quattro byte (32 bit). La dimensione di una variabile di tipo `int`, comunque, cambia da macchina a macchina e l'unica relazione universale è:

```
sizeof(char) <= sizeof(int)
```

Ossia: la dimensione di un `char` è minore o uguale alla dimensione di un `int`.

Il tipo `float` corrisponde a un dato numerico di singola precisione in virgola mobile. Anche la dimensione di un `float` dipende dall'architettura della macchina in esame: spesso un `float` occupa 4 byte (32 bit).

Il tipo `double` corrisponde a un dato numerico in virgola mobile, ma in doppia precisione. Anche la dimensione del `double` dipende dalla macchina: in molti casi un `double` occupa 8 byte (64 bit). L'unica relazione universale è:

```
sizeof(float) <= sizeof(double)
```

I quattro tipi fondamentali possono essere ulteriormente specificati per mezzo di due tipi di qualificatori, i *qualificatori di dimensione*

```
short    e        long
```

e il *qualificatore aritmetico*

```
unsigned
```

I qualificatori di dimensione si applicano al tipo fondamentale `int`:

```
unsigned short    <---->    unsigned short int  
unsigned long     <---->    unsigned long int
```

Il tipo `long double` non può invece essere abbreviato.

Abbiamo visto come la dimensione dei tipi dipenda dalle caratteristiche del processore della macchina ospite. Nella tabella seguente si riportano i dati che si ritrovano in alcune delle più diffuse architetture.

<u>Tipo</u>	<u>Dimensione</u>
<code>char</code>	1 byte
<code>short</code>	2 byte
<code>int</code>	4 byte
<code>long</code>	4 byte
<code>float</code>	4 byte
<code>double</code>	8 byte
<code>long double</code>	8 byte

Con il semplice programma illustrato nel Listato 11.1 si può verificare su qualunque macchina la dimensione dei tipi fondamentali.

```
#include <stdio.h>

main()
{
    int ch, in, sh, lo, fl, dd, ld;

    ch = sizeof(char);
    in = sizeof(int);
    sh = sizeof(short);
    lo = sizeof(long);
    fl = sizeof(float);
    dd = sizeof(double);
    ld = sizeof(long double);
}
```

- • le costanti decimali;
- • le costanti ottali;
- • le costanti esadecimali;
- • le costanti carattere.

Le costanti decimali sono quelle più comunemente usate e sono della forma:

0                      1234                      976                      12345678901234567890

Il tipo di una costante decimale è `int` se `int` è sufficiente a contenerla, altrimenti è `long`. Il compilatore avverte se si stanno trattando costanti più grandi di quelli rappresentabili sulla macchina. Per facilitare la programmazione di basso livello il linguaggio consente la definizione di costanti numeriche in sintassi ottale ed esadecimale (purtroppo manca la sintassi binaria):

- • ottale                      aggiungere una cifra 0 prima della costante
- • esadecimale                      aggiungere 0x o 0X prima della costante

Per esempio: 012, 077, -05 sono costanti ottali, mentre 0xAA, 0xddL, 0xF sono costanti esadecimali.

Una costante floating point è di tipo `double`. Il compilatore, in genere, avverte anche in questo caso se si stanno trattando costanti più grandi di quelli rappresentabili sulla macchina. Esempi di costanti floating point sono:

1.23                      .23                      0.23                      1.                      1.0                      1.2e10                      1.23e-5

Non si possono inserire spazi nel mezzo di una costante floating point. Per esempio, la sequenza di caratteri

65.43 e - 21

non è una costante floating point ma rappresenta quattro simboli diversi: 65.43, e, -, 21, che provocheranno un errore

```
lettera = 'A';
```

Questa rappresentazione è comoda quando si devono utilizzare combinazioni non alfanumeriche, come quelle presenti nelle linee della tabella successive alla prima. Si provi a verificarne la rappresentazione nel codice ASCII.

Per i compilatori conformi allo standard ANSI è prevista anche la codifica esadecimale per le costanti `char`:

```
\xhhh
```

che si ottiene facendo precedere il valore da `\x: '\xfa', '\1a'`. Comunque, poiché esistono molte costanti carattere non alfanumeriche di uso comune, il C aiuta ulteriormente il programmatore nella loro definizione.

<code>\'</code>	Apice singolo
<code>\"</code>	Doppio apice
<code>\?</code>	Punto interrogativo
<code>\\</code>	Backslash – carattere \
<code>\a</code>	“Bell”
<code>\b</code>	“Backspace” (^H)
<code>\f</code>	“Form feed” (^L)
<code>\n</code>	New line (^J)
<code>\r</code>	Carriage return (^M)
<code>\t</code>	“Tab” (^I)
<code>\v</code>	Tabulazione verticale (^V)

Si ricorda che, nonostante l'apparenza, questi sono tutti caratteri singoli.

Le costanti cui si fa riferimento con un nome (o simbolo) sono dette costanti *simboliche*. Una costante simbolica è quindi un nome il cui valore non può essere ridefinito nell'ambito di validità del nome stesso. In C esistono tre tipi di costanti simboliche:

definisce tre costanti intere dette enumeratori alle quali assegna implicitamente dei valori interi sequenziali e crescenti a partire da zero; `enum { QUI, QUO, QUA }` è equivalente a:

```
const QUI = 0;
const QUO = 1;
const QUA = 2;
```

A una enumerazione può essere associato un nome:

```
enum papero { QUI, QUO, QUA };
```

che non è un nuovo tipo ma un sinonimo di `int`. Agli enumeratori possono essere anche assegnati esplicitamente valori:

```
enum valore_simbolo {
    NOME, NUMERO, FINE,
    PIU = '+', MENO = '-', PER = '*', DIV = '/'
};
```

La dichiarazione

```
valore_simbolo x;
```

costituisce un utile suggerimento sia per il lettore sia per il compilatore!

## 11.4 Il trattamento dei bit

Per scrivere applicazioni che controllano dispositivi hardware è necessario disporre di operatori in grado di lavorare sui singoli bit di un registro. Se per esempio si ha una centralina di controllo di un impianto di illuminazione, dove un



1	0	0
0	1	0
1	1	1

Se per esempio volessimo mettere a 0 i quattro bit meno significativi di una variabile `x` di tipo `unsigned char`, e lasciare inalterato lo stato di quattro bit più significativi, basterebbe mettere in `AND` `x` con la sequenza:

1 1 1 1 0 0 0 0

il cui valore ottale è 360. In pratica si tratta di formare una sequenza di bit dove sia posto il valore 1 in corrispondenza dei bit di `x` che devono rimanere invariati e il valore 0 in corrispondenza dei bit di `x` che devono essere trasformati in zero. Se lo stato iniziale dei bit di `x` fosse

1 0 1 0 1 0 1 0

dopo l'istruzione

```
x = x & '\360';
```

lo stato dei bit di `x` diventerebbe:

1 0 1 0 0 0 0 0

L'istruzione di OR bit a bit è usata per *accendere* uno o più bit di una variabile. Infatti, nel confronto bit a bit dell'operazione booleana OR basta avere uno dei due operandi a 1 per produrre 1.

b1	b2	b1   b2
0	0	0
1	0	1

1	0	1
0	1	1
1	1	0

Se per esempio volessimo scambiare 0 con 1 e 1 con 0 nei primi quattro bit più significativi di una variabile `unsigned char`, e lasciare inalterato lo stato dei quattro bit meno significativi, basterebbe usare l'istruzione:

```
x = x ^ '\360';
```

Se prima dell'istruzione lo stato dei bit di `x` fosse

```
1 0 1 0 1 0 1 0
```

corrispondendo la costante `'\360'` alla sequenza

```
1 1 1 1 0 0 0 0
```

dopo l'istruzione

```
x = x ^ '\360';
```

lo stato dei bit di `x` sarebbe:

```
0 1 0 1 1 0 1 0
```

Si osservi che effettuare l'OR esclusivo di una variabile con se stessa ha l'effetto di mettere a 0 la variabile. L'istruzione

```
x = x ^ x;
```

trasforma sempre `x` in:

- se si esegue lo shift a destra di una variabile con segno i bit vacanti a sinistra sono rimpiazzati con il bit del segno su certe macchine (shift aritmetico) e con 0 su altre (shift logico);
- • se si esegue lo shift a sinistra di una variabile `unsigned`, i bit vacanti a destra sono rimpiazzati con degli 0;
- • se si esegue lo shift a sinistra di una variabile con segno, i bit vacanti a destra sono rimpiazzati con degli 0 e il bit del segno rimane inalterato.

Vediamo ora alcuni esempi di operazioni di shift.

**1)**        `char c;`  
              `c = c << 1;`

Se prima dello shift i bit di `c` fossero stati

1 0 0 0 1 0 0 1

dopo lo shift avremmo avuto:

1 0 0 1 0 0 1 0

Avremmo cioè mantenuto a 1 il primo bit di segno.

**2)**        `char c;`  
              `c = c >> 1;`

Se prima dello shift i bit di `c` fossero stati

1 0 0 0 1 0 0 1

dopo lo shift avremmo avuto

Nell'esempio abbiamo raccolto `x>>3` all'interno di parentesi tonde: infatti avendo l'operatore `>>` minore priorità di `&`, in caso contrario sarebbe stato effettuato all'interno dell'espressione prima l'AND bit a bit tra 3 e 01 e poi lo shift a destra di `x`.

È il momento di ampliare la tavola gerarchica delle priorità degli operatori (Figura 11.1), dove inseriamo anche la trasformazione di tipo indicata nella tabella con `(tipo)`, che studieremo approfonditamente nel prossimo paragrafo. Si tenga conto che per gli operatori binari per il trattamento dei bit, come per quelli aritmetici, è possibile utilizzare la notazione:

```
variabile [operatore]= espressione
```

quando una variabile appare sia a sinistra sia a destra di un operatore di assegnamento. Pertanto l'istruzione:

```
x = x << 2;
```

può essere scritta come ■

```
x <<= 2;
```

!	-	++	--	sizeof	(tipo)
		*	/	%	
		+	-		
		>>	<<		
		>	>=	<	<=
		==	!=		

convertito in `int`, `short int` e `long int`. Il risultato non è definito quando il valore `float` da convertire è troppo grande, perché cambia da macchina a macchina;

- 5) 5) il tipo `double` può essere convertito in `float`, potendo provocare troncamento nell'arrotondamento. Inoltre, per il `double` vale quanto detto per il `float` a proposito di `int`, `short int` e `long int`.

### ✓ NOTA

Le conversioni implicite di tipo vengono effettuate per rendere conformi, se possibile, i tipi di due operandi in un'espressione e i tipi dei parametri attuali e formali nel passaggio dei parametri di una funzione. In C sono possibili molte conversioni implicite di tipo, ma ci limiteremo a considerare solo quelle che il programmatore può ragionevolmente ammettere. Per tutte le altre conversioni, non è saggio affidarsi al compilatore, soprattutto per motivi di leggibilità e portabilità delle applicazioni, e, pertanto, si raccomanda di usare la conversione esplicita di tipo.

In C l'esecuzione degli operatori aritmetici effettua la conversione implicita degli operandi se questi non sono omogenei. Lo schema di conversione usato è comunemente detto *conversione aritmetica normale*, ed è descritto dalle regole seguenti.

- • Convertire gli operandi `char` e `short int` in `int`; convertire gli operandi `float` in `double`.
- • Se uno degli operandi è di tipo `float`, allora l'altro operando, se non è già `double`, è convertito in `double`, e il risultato è `double`.
- • Se uno degli operandi è di tipo `long`, allora l'altro operando, se non è già `long`, viene convertito in `long`, e il risultato è `long`.
- • Se un operando è di tipo `unsigned`, allora l'altro operando, se non è già `unsigned`, è convertito in `unsigned`, e il risultato è `unsigned`.
- • Se non siamo nella situazione descritta dai casi 2 e 4, allora entrambi gli operandi debbono essere di tipo `int` e il risultato è `int`.

In forma più sintetica, queste regole di conversione possono essere illustrate dal diagramma seguente.

```
double dd;  
int i;  
...  
i = i + dd;
```

la variabile intera `i` è convertita in `double`, sommata alla variabile `double dd` e il risultato della somma, prima dell'assegnazione, è convertito in `int`, con eventuale perdita di precisione.

Un'espressione di un certo tipo può essere esplicitamente convertita in un altro tipo per mezzo dell'operazione detta di *cast*. Abbiamo già incontrato il cast nel Capitolo 9, a proposito delle funzioni di allocazione e deallocazione dinamica della memoria `malloc`, `calloc` e `free`, e dell'indirizzamento assoluto della memoria. La sintassi generale del cast è:

*(nome\_tipo) espressione*

Il valore di *espressione* è trasformato in un valore il cui tipo è *nome\_tipo*. Esempi di *nome\_tipo* sono:

```
char  
char[8]  
char *  
char ()  
int  
void
```

In pratica *nome\_tipo* è lo stesso che si usa nella dichiarazione delle variabili. Si consideri il semplice esempio:


```
char *pc;  
int a;  
...  
a = 0177777;  
pc = (char *)a;
```

## 11.6 Funzione di visualizzazione

Abbiamo ripetutamente utilizzato negli esempi la funzione di immissione dati, `scanf`, e quella di visualizzazione, `printf`. La sintassi di queste funzioni è stata usata solo sommariamente, ed è ora giunto il momento di definirla (in questo paragrafo e nel prossimo) in modo più completo e generale.

La sintassi della funzione `printf` è:

```
int printf(char *format, arg1, arg2, ...)
```

La funzione `printf` converte, ordina in un formato e stampa sul video (detto standard output ) un numero variabile di argomenti `arg1`, `arg2`, ... sotto il controllo della stringa `format`. Questa stringa dettaglia due diverse informazioni: un insieme di caratteri ordinari, che vengono direttamente inviati sul video, e una specifica di conversione per ognuno degli argomenti da visualizzare. Ogni specifica di conversione inizia con il carattere `%` e termina con il carattere di conversione. Dopo il carattere `%` possono essere presenti i simboli illustrati nei punti successivi.

- 1) 1) Zero, uno o più *flag* che modificano il significato della specifica di conversione. Tipici flag per la modifica delle conversioni sono i segni meno, più e il carattere vuoto:

-	il risultato della conversione sarà accostato a sinistra nel campo;
+	il risultato di una conversione con segno inizierà sempre con il segno (+ o -);
carattere	se il primo carattere di una conversione con segno non è un segno, un carattere
vuoto	vuoto precede la visualizzazione del risultato. Ciò significa che se sono specificati sia il carattere vuoto sia il +, il carattere vuoto sarà semplicemente ignorato.

- 2) 2) Una stringa opzionale di cifre decimali che specifica l'ampiezza minima del campo riservato al corrispondente argomento. Se il valore convertito ha meno caratteri di quelli specificati nell'ampiezza del campo, il valore è accostato a sinistra o a destra del campo, dipendentemente dal flag di accostamento. Se il flag di accostamento non è presente il valore viene posto a destra del campo, se è - (vedi punto precedente) il valore viene posto a sinistra.
- 3) 3) Un punto che separa l'ampiezza di campo dalla successiva stringa di cifre, detta precisione.

visualizzato. Alcuni esempi di specifica relativi a una stringa sono:

```
%10s      |Ciao, lettore|
%-10s     |Ciao, lettore|
%20s      |      Ciao, lettore|
%-20s     |Ciao, lettore  |
%20.10s   |      Ciao, lett|
%-20.10   |Ciao, lett      |
%.10s     |Ciao, lett|
```

■ dove la dimensione del campo è stata convenzionalmente delimitata con il simbolo `|`.

## 11.7 Funzione di immissione

La sintassi di `scanf` è:

```
int scanf(char *format, puntat1, puntat2, ...)
```

La funzione `scanf` legge un insieme di caratteri da tastiera (detta standard input ■), li interpreta secondo il formato specificato dalla stringa `format` e li memorizza negli argomenti puntati da `puntat1`, `puntat2` ecc. La stringa di formato, detta anche controllo, contiene le specifiche di conversione, che sono usate da `scanf` per interpretare la sequenza di immissione. Nel controllo possono essere presenti i simboli riportati nei punti successivi:

- • caratteri vuoti (blank, tab, newline, formfeed), che sono ignorati;
- • caratteri normali escluso `%`, che dovrebbero coincidere con il prossimo carattere non vuoto della sequenza di ingresso;
- • le specifiche di conversione formate dal carattere `%`, da un carattere `*` opzionale di soppressione assegnamento, un'ampiezza massima di campo opzionale, un carattere `l` o un carattere `o` opzionali che denotano la



La conversione effettuata dalla funzione `scanf` sullo standard input termina quando si incontra una costante EOF, alla fine della stringa di controllo, o quando si incontra un carattere in immissione che è in contraddizione con la stringa di controllo. In quest'ultimo caso il carattere che ha provocato la contraddizione non viene letto dallo standard input. La funzione `scanf` ritorna il numero di immissioni che è riuscita a concludere con successo. Questo numero può anche essere zero, nel caso in cui si verifichi immediatamente un conflitto tra un carattere in immissione e la stringa di controllo. Se l'immissione si conclude prima del primo conflitto o della prima conversione, `scanf` restituisce un EOF. Esempi classici di uso di `scanf` sono:

```
int i; float x; char nome[30];  
...  
scanf("%d%f%s", &i, &x, nome);
```

Sulla linea di immissione si potrebbe avere

```
8 54.52E-1  Abel
```

memorizzando 8 in `i`, 5.452 in `x`, e la stringa "Abel" in `nome`. Usando sempre la medesima definizione di variabili

```
scanf("%2d%f%d %s", &i, &x, &nome);
```

e avendo sulla linea di immissione

```
56789 0123 babbo
```

verrà assegnato 56 a `i`, 789.0 a `x`, verrà ignorato 0123 e assegnato `babbo\0` a `nome`.

Infine si ricordi che l'argomento di `scanf` deve essere sempre un puntatore. Purtroppo è molto facile scordarlo e di programmi in cui si scrivono istruzioni tipo

```
int n;
```

```

#include <stdio.h>

int    i = 80;
double d = 3.14546;
main()
{
    int  numusc;
    char bufusc[81];

    numusc = sprintf(bufusc, "Il valore di i = %d e \
il valore di d = %g\n", i, d);

    printf("sprintf() ha scritto %d caratteri e il \
buffer contiene:\n%s", numusc, bufusc);
}

```

### Listato 11.2 Copia su stringa

La funzione `sscanf` ha lo scopo di leggere dei caratteri da una stringa, `s`, che possiamo chiamare buffer di caratteri, convertirli e memorizzarli nelle locazioni puntate da `punt1`, `punt2`, ... secondo il formato specificato dalla stringa `format`. Per ciò che riguarda la specifica di formato valgono le medesime convenzioni di `scanf`. Tipicamente `sscanf` è usata per la conversione di caratteri in valori. Generalmente si leggono le stringhe da decodificare, per esempio, con la funzione `gets`, e poi si estraggono dei valori dalla stringa con `sscanf`.

La funzione `sscanf` restituisce il numero di campi che sono stati letti, convertiti e assegnati a variabili con successo. Se la stringa dovesse finire prima della fine dell'operazione di lettura, `sscanf` ritornerebbe il valore costante `EOF`, definito in `stdio.h`. A titolo di esempio si consideri il programma del Listato 11.3, che accetta in ingresso assegnazioni di variabili secondo il formato "`nome = <valore>`". La funzione `gets` brutalmente legge tutta la stringa corrispondente all'assegnazione e la funzione `sscanf` la scompone nei due elementi `nome` e `valore` del `nome`.

```
unsigned char c;  
c = '\166' & '\360';  
printf("c: %o\n", c);
```

che cosa visualizzerà la printf, e perché? E se al posto di '\166' avessimo scritto '\100' oppure '\0' oppure '\111' o '\110'?

4. Dato il seguente frammento di programma

```
unsigned char c;  
c = '\111' ^ '\360';  
printf("c: %o\n", c);
```

che cosa visualizzerà la printf, e perché? E se al posto di '\111' avessimo scritto '\321' o '\350'?

5. Dato il seguente frammento di programma

```
unsigned char c;  
c = ~'\351';  
printf("c: %o\n", c);
```

che cosa visualizzerà la printf? E se al posto di '\351' avessimo scritto '\222' o '\123'?

5. Se la variabile c di tipo unsigned char ha valore '\123', quali valori stamperebbe l'istruzione printf("c: %o\n", c) dopo ognuno dei seguenti singoli assegnamenti?

```
c = c >> 1;  
c = c >> 2;  
c = c >> 3;
```

Per mezzo di un array è possibile individuare mediante un nome e un indice un insieme di elementi dello stesso tipo. Se per esempio volessimo rappresentare un'area di memoria di 200 byte su cui andare a scrivere o a leggere dei dati, potremmo definire una variabile `buf` :

```
int buf[100];
```

Ogni locazione di questa memoria verrebbe individuata attraverso un indice. Così

```
buf[10]
```

rappresenterebbe l'undicesima locazione di memoria del buffer.

Ci sono però problemi in cui è necessario aggregare elementi di tipo diverso per formare una struttura. Per esempio, se si vuol rappresentare il concetto di data basta definire una *struttura* siffatta:

```
struct    data    {  
    int    giorno;  
    char    *mese;  
    int    anno;  
};
```

Si osservi come per effetto di questa dichiarazione si sia introdotto nel programma un nuovo tipo, il tipo `data`. D'ora in poi sarà possibile definire variabili in cui tipo è `data`:

```
struct data oggi;
```

Come deve essere interpretata la variabile `oggi` di tipo `data`? Semplicemente come una variabile strutturata, composta di tre parti: due di tipo `int` - `giorno` e `anno` - e una di tipo `stringa` - `mese` -. La sintassi generale per la definizione di una struttura è:

```
struct nome_struttura {  
    tipo_membro nome_membro1;  
    tipo_membro nome_membro2;
```

```
};  
struct automobile a1, a2;
```

È lasciato alla sensibilità del lettore stabilire quale delle due convenzioni usare per definire una variabile struttura.  
Per poter accedere ai campi di una variabile di tipo struttura si fa uso dell'operatore punto (.):

```
oggi.giorno = 25;  
oggi.mese = "Dicembre";  
oggi.anno = 2001;  
ieri.anno = oggi.anno;
```

Con le prime tre istruzioni si assegna la terna di valori <25, "Dicembre", 2001> alla variabile oggi e con la quarta si rendono uguali l'anno di ieri con quello di oggi.

La sintassi generale con cui si fa riferimento a un membro è

*nome\_variabile\_struttura.nome\_membro*

Per esempio, quando si vuole visualizzare il contenuto di una variabile struttura può essere necessario ricorrere all'operatore punto: si veda il Listato 12.1, la cui esecuzione produrrà il seguente risultato:

```
marca auto  = FERRARI  
modello auto  = F40  
vendute      = 200  
marca auto  = OPEL  
modello auto  = ASTRA  
vendute      = 1200
```

```
/* Esempio di definizione di una struttura */
```

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

ai singoli membri. Per esempio, in C è possibile effettuare assegnazioni tra variabili struttura dello stesso tipo:

```
struct data compleanno, oggi;  
    compleanno = oggi;
```

è una assegnazione consentita che effettua una copia di tutti i valori dei membri di `oggi` nei corrispondenti valori dei membri di `compleanno`. Due strutture corrispondono a due tipi differenti anche se hanno gli stessi membri. Per esempio:

```
struct s1 {  
    int a;  
};  
struct s2 {  
    int a;  
};
```

sono due tipi struttura differenti:

```
struct s1 bob;  
    struct s2 alex;  
    ...  
    alex = bob; /* errore: tipi dati discordi */
```

Quest'ultima assegnazione darebbe luogo a un errore, poiché si è tentato di assegnare una variabile di tipo `s1` a una di tipo `s2`. Inoltre si ricordi che i tipi struttura sono diversi anche dai tipi fondamentali. Per esempio, in

```
struct s1 bobo;  
    int i;  
    ...  
    bobo = i; /* errore: tipi dati discordi */
```

l'ultima assegnazione provocherebbe un errore a causa della disomogeneità di tipo

```

...
else
    return(0);
}

```

Alla funzione `numero_mese` viene passato un puntatore a variabile di tipo `data` e si ottiene il numero del mese relativo alla data puntata dal puntatore. Nel caso in cui il nome del mese non corrisponda ad alcuno di quelli conosciuti, la funzione ritorna un codice di errore pari a 0.

Infine, una ultima considerazione a proposito della inizializzazione di una variabile di tipo struttura. Abbiamo visto come per mezzo dell'operatore `punto` sia possibile assegnare dei valori a una variabile struttura. In alternativa si può usare una sintassi analoga a quella usata per inizializzare gli array:

```

struct automobile {
    char *marca;
    char *modello;
    int venduto;
};
struct automobile a = {"FERRARI", "F40", 200};

```

oppure

```

struct data {
    int giorno;
    char *mese;
    int anno;
} oggi = {25, "Dicembre", 2001};

```

Questo tipo di inizializzazione è ammesso solo se le corrispondenti variabili sono di tipo globale, cioè se sono definite all'esterno di un blocco. ■

contiene il valore 2 (figura 12.1).

`a.i = 2;`

2

a

`a.d = 3L;`

3
---

a

`a.c = 'A'L;`

A	

a

Figura 12.1 Rappresentazione di una unione

Le unioni vengono preferite alle strutture quando si hanno delle variabili che possono assumere più tipi e/o dimensioni diverse a seconda delle circostanze. Per esempio, supponiamo di voler rappresentare per mezzo di una struttura una tavola di simboli, cioè una tabella i cui elementi siano una coppia `<nome, valore>` e dove il valore associato a un nome può essere o una stringa o un numero intero. Una possibile struttura per la tavola dei simboli potrebbe essere:



```

        char val_str; /* usato se tipo == 's' */
        int val_int; /* usato se tipo == 'i' */
    };
}

```

Con questa soluzione i programmi che inseriscono, cancellano, ricercano o stampano una voce della tavola dei simboli rimangono invariati, e per mezzo del concetto di unione si ha che i due attributi `val_str` e `val_int` hanno lo stesso indirizzo, ovvero sono allocati nella medesima area di memoria.

Talvolta le unioni sono usate anche per effettuare conversioni di tipo. Questa pratica può essere fonte di ambiguità e quindi di errore. Il lettore si attenga a quanto detto a proposito delle conversioni implicite ed esplicite di tipo.

## 12.4 Campi

I *campi* sono un costrutto del C che consente di far riferimento in modo simbolico ai singoli bit di una variabile:

```

struct {
    unsigned k: 1;
    unsigned j: 6;
} var_bit_bit;

```

La variabile `var_bit_bit` occupa uno spazio di memoria pari alla somma dei bit utilizzati da ogni campo arrotondata alla *word* (parola) della macchina. In altri termini, supponendo che la parola della macchina sia di 16 bit, si ha che:

- • il campo `k` di `var_bit_bit` occupa 1 bit;
- • il campo `j` di `var_bit_bit` occupa 6 bit;
- • 9 bit rimangono inutilizzati.

I campi possono essere trattati alla stregua dei membri di una struttura e quindi essere utilizzati in espressioni come la

```
typedef char * Stringa;  
    Stringa s1, s2;
```

dapprima il tipo `char *`, cioè il tipo puntatore a carattere, viene ribattezzato `Stringa`. Successivamente si definiscono due variabili, `s1` e `s2`. La sintassi di `typedef` è

```
typedef nome_tipo nuovo_nome_tipo;
```

dove *nome\_tipo* è il nome simbolico del tipo che si vuol ribattezzare e *nuovo\_nome\_tipo* è il nome che si associa a *nome\_tipo*. Ribattezzare un tipo può essere molto utile soprattutto per rendere più leggibile un programma, e per evitare espressioni altrimenti complesse. Per esempio con il frammento di programma

```
typedef char * Stringa;  
    Stringa p;  
  
    int strlen(Stringa);  
    p = (Stringa)malloc(100);
```

si introduce il nome `Stringa` per rappresentare un generico puntatore a carattere. Da questo punto in avanti `Stringa` può essere usato come nome di un qualsiasi tipo; infatti ne facciamo uso nella definizione della variabile `p`, del parametro formale della funzione `strlen` e nella conversione esplicita di tipo che alloca 100 caratteri e li fa puntare da `p`.

L'uso di `typedef` può risultare utile anche quando si ha a che fare con le strutture. Per esempio si potrebbe avere

```
struct automobile {  
    char *marca;  
    char *modello;
```

2. 2. si interpretano le eventuali parentesi tonde o quadrate, e poi si guarda alla sinistra per vedere se c'è un asterisco;
3. 3. se si incontra durante un qualsiasi passo una parentesi tonda chiusa, si torna indietro e si riapplicano le regole 1 e 2 a tutto ciò che si trova all'interno delle parentesi;
4. 4. infine si applica lo specificatore di tipo.

Proviamo subito ad applicare queste regole al precedente esempio:

```
char  * (* (*frog) ()) [10];
      7   6 4 2 1   3     5
```

I vari passi sono etichettati da 1 a 7 e sono interpretati nel modo seguente:

- 1 l'identificatore di variabile `frog` è un
- 2 **puntatore a**
- 3 una funzione che ritorna un
- 4 **puntatore a**
- 5 un array di 10 elementi, che sono
- 6 **puntatori a**
- 7 **valori di tipo `char`**

È lasciato al lettore il compito di decifrare le seguenti dichiarazioni:

```
int *frog[5];
    int (*frog)[5];
```

Di tutte le possibili combinazioni di tipi derivati ne esistono alcune più frequentemente usate:

1. 1. tipi derivati composti tramite struttura;
2. 2. tipi derivati composti tramite funzione;

```

        char telefono[10];
        char parentela[2];
    };

```

Le strutture `data`, `ind` e `persona` hanno membri di tipo e dimensione differenti. In particolare si osserva che la struttura `persona` ha dei membri che sono a loro volta delle variabili struttura: `data_nasc` è di tipo `data` e `indirizzo` è di tipo `ind`. In C è possibile definire strutture di strutture purché le strutture che intervengono come membri siano state definite prima della struttura che le contiene.

Questo, ricordando le regole di visibilità di una variabile, è assolutamente normale. Ma cosa succede se una generica struttura `S` ha un membro che a sua volta è una struttura dello stesso tipo `S`? Se per esempio volessimo mantenere una lista di persone ordinata per ordine alfabetico, dovremmo prevedere nella struttura `persona` un membro di tipo `persona` che dice qual è la prossima persona nell'elenco ordinato. In C, però, una struttura `S` non può contenere un membro di tipo puntatore a `S`. Così il problema della lista ordinata potrebbe essere risolto con:

```

struct persona {
    char nome[30];
    char cognome[30];
    struct data data_nasc;
    char comune_nasc[30];
    struct ind indirizzo;
    char telefono[10];
    char parentela[2];    /* CF capofamiglia, CG coniuge ecc. */
    struct persona *link; /* punta alla persona seguente */
}

```

Il caso delle strutture ricorsive è molto più comune di quanto si possa pensare. Alcune delle più importanti strutture dati quali le liste, le pile e gli alberi sono rappresentabili mediante strutture ricorsive; a tale proposito il lettore troverà numerosi esempi in successivi capitoli di questo testo.

Se la struttura è lo strumento con cui si rappresenta una classe di oggetti (le automobili, gli impiegati, i nodi di un albero ecc.) aggregando un insieme di membri è logico aspettarsi una molteplicità di elementi, cioè di variabili

4. Visualizza Anagrafe  
0. Fine

Scegliere un'opzione:

Con la scelta 1, Immissione Persona, si lancia la funzione `ins_per` che inserisce i dati di una persona nell'array `anag`; la funzione è molto semplice: non effettua controllo sull'esistenza di una persona nell'array `anag`, ma inserisce dati finché lo consentono le dimensioni dell'array. Con la scelta 2, Cancellazione Persona, viene invocata la funzione `can_per` che richiede dapprima all'utente i seguenti dati:

CANCELLA PERSONA

```
-----  
Cognome  :  
Nome    :  
Età     :
```

e poi invoca la funzione `cer_per`. Quest'ultima prende in ingresso le variabili `cognome`, `nome` ed `età` inserite dall'utente, effettua una ricerca sequenziale nell'array `anag`, e se trova la persona corrispondente ai dati forniti in ingresso restituisce il puntatore, `ps`, alla prima occorrenza dell'array che corrisponde ai dati richiesti dall'utente. Il lettore osservi come `cer_per` sia un classico esempio di funzione che restituisce un puntatore a variabile di tipo struttura:

```
struct per *cer_per(char *cg, char *nm, int et)  
{  
    ...  
}
```

Se `per` `cer_per` non trova nessuna persona con i dati richiesti allora restituisce un puntatore `NULL`. Nel caso in cui invece `cer_per` trovi una occorrenza valida, la funzione `can_per` prende il puntatore all'occorrenza e lo passa a `vis_per`, che ha lo scopo di visualizzare i dati della persona che si vuol eliminare. La funzione `vis_per` è un classico esempio di funzione che prende in ingresso un puntatore a struttura:

una funzione che ritorna un `int` e ha un solo parametro di tipo `float`. In generale la sintassi di dichiarazione di un puntatore a funzione è

```
tipo_funzione (*nome_puntatore_funzione) (tipo_parametri);
```

dove *tipo\_funzione* è il tipo della funzione che agisce sui parametri il cui tipo è *tipo\_parametri* ed è puntata da *nome\_puntatore\_funzione*.

A che cosa serve un puntatore di funzione? Prima di tentare una risposta si consideri che accedere al contenuto di un puntatore funzione tramite il consueto operatore `*` equivale a invocare la funzione:

```
double risultato;  
    float base = 3.0;  
    double (*pf) (float);  
    double cubo(float);  
  
    pf = cubo; /* inizializzazione del puntatore */  
    risultato = (*pf) (base);  
    printf("Il cubo di %f è %f", base, risultato)
```

L'istruzione

```
risultato = (*pf) (base);
```

è perfettamente equivalente a

```
risultato = cubo(base);
```

Vediamo ora con il Listato 12.3 un esempio completo volto a chiarire l'importanza dei puntatori di funzione.

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

```

    if(scelta==0)          /*uscita programma */
        loop = 1;
    else
        /* esecuzione della funzione associata alla scelta */
        (*menu[scelta-1].fun) ();
}
}

void fun1(void)
{printf("\n\n Sto eseguendo fun1\n\n\n");}

void fun2(void)
{printf("\n\n Sto eseguendo fun2\n\n\n");}

void fun3(void)
{printf("\n\n Sto eseguendo fun3\n\n\n");}

```

### Listato 12.3 Puntatori di funzione per la creazione di un menu

Questo semplice programma presenta il seguente menu:

1. Funzione fun1
2. Funzione fun2
3. Funzione fun3
0. Fine

Scegliere l'opzione desiderata:

```

    int *pi;
    char *pausa;

    pi = &a;
    ppi = &pi;
    printf("%d", **ppi);
    gets(pausa);
}

```

la variabile `pi` contiene l'indirizzo della variabile intera `a`, e `ppi` contiene l'indirizzo di `pi`. Conseguentemente `*ppi` corrisponde al contenuto di `pi`, cioè all'indirizzo di `a`, e `**ppi` corrisponde al contenuto di `a`. Infatti l'istruzione

```
printf("%d", **ppi);
```

visualizza il numero 3 (Figura 12.2).

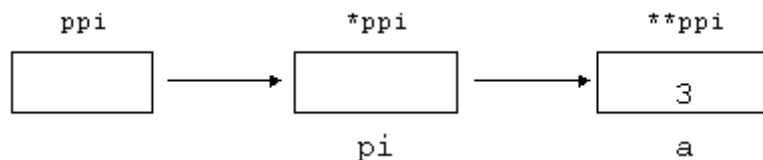


Figura 12.2 Puntatore di puntatore a intero

È naturalmente possibile dichiarare puntatori di puntatori di puntatori. Il programmatore deve però prestare molta attenzione perché una doppia, o peggio ancora, tripla indirezione è difficile da controllare in C; pertanto i puntatori debbono essere usati con parsimonia. Esistono comunque casi in cui è necessario usare una doppia indirezione, cioè un puntatore di puntatore: un tipo derivato composto comunemente usato in C è l'array di puntatori, e per scandire un array di puntatori si fa uso del tipo `array di puntatori di puntatori` (Figura 12.4).



N. Voce di Menu N

Come si può osservare, menu è un array di puntatori a carattere. Infatti il lettore ricordi che ogni costante stringa riportata tra doppi apici corrisponde all'indirizzo del primo carattere della stringa. Per contrassegnare la fine dell'array di puntatori a carattere menu si è usato il puntatore NULL. Per visualizzare le varie voci di menu basta passare a `printf` l'indirizzo delle voci di menu raggiungibile tramite `*ppc`. Inoltre `ppc` deve essere incrementato per scandire gli elementi dell'array (Figura 12.3).

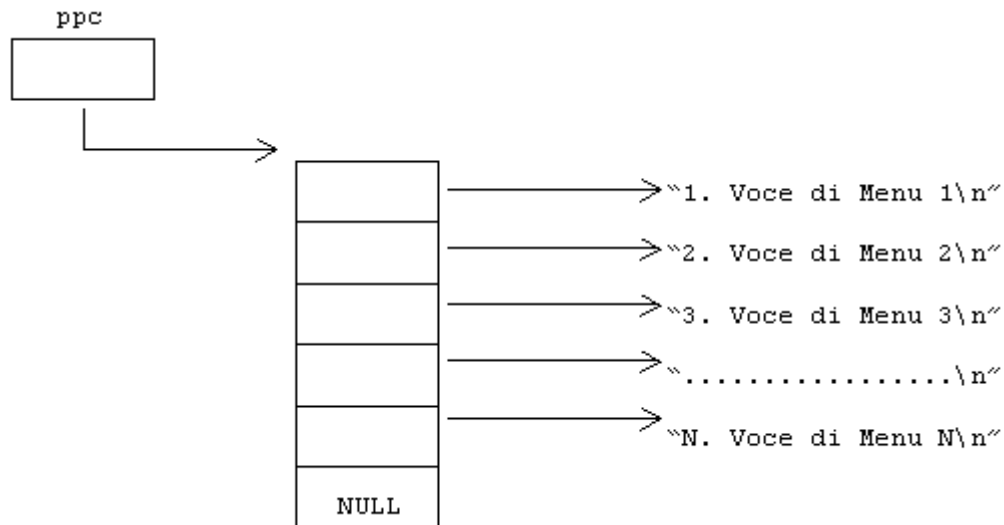


Figura 12.3 Rappresentazione grafica di un array di puntatori

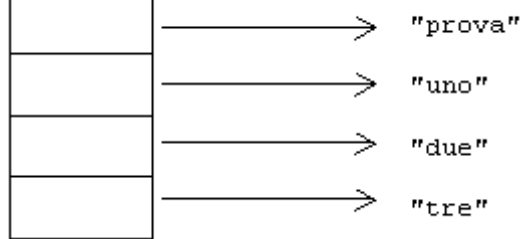


Figura 12.4 Rappresentazione del passaggio di parametri a main

Nel programma `prova.c` attraverso l'indice `i` si scandisce l'array standard `argc` e l'indirizzo delle stringhe `uno`, `due` e `tre` viene iterativamente passato alla funzione `printf`. Dunque lanciando `prova`:

```
C:\>prova uno due tre
```

si ottiene

```
unoduetre
```

## 12.10 Classificazione delle variabili

Oltre a essere classificata in base al tipo (fondamentale, derivato e derivato composto), in C una variabile può essere anche classificata secondo la visibilità all'interno del programma e secondo la classe di memoria.

Del concetto di visibilità di una variabile abbiamo già trattato a proposito delle funzioni, dove abbiamo imparato a distinguere tra variabili locali e globali. L'ambito di definizione o *scope* di un nome globale si estende dal punto di definizione sino alla fine del file, mentre l'ambito di validità di un nome locale si estende dal punto di definizione sino

```

void vis_per(void)
{
char pausa; int i;

for(i=0; strcmp(anag[i].cognome, TAPPO)!=0; i++) {
    printf("\n\n-----\n");
    printf("\n\t\tCognome : %s", anag[i].cognome);
    printf("\n\t\tNome : %s", anag[i].nome);
    printf("\n\t\tIndirizzo : %s", anag[i].ind);
    printf("\n\t\tEtà : %d", anag[i].eta);
    printf("\n\n-----\n");

    scanf("%c", &pausa);
}
}

```

### Listato 12.5 Variabili locali e globali

Consideriamo il Listato 12.5. In questo programma la variabile `anag` è globale mentre le variabili `i` e `pausa` sono locali rispettivamente alla funzione `vis_per`. Finora abbiamo sempre considerato programmi C contenuti in un solo file sorgente. In generale, però, un programma C di media-alta complessità si estende su più file. Per esempio, il programma precedente potrebbe essere diviso su due file, uno contenente il `main` e l'altro la funzione `vis_per` e tutte le altre eventuali funzioni che vorremo far operare su `anag` (Listati 12.6 e 12.7).

```

/* programma principale: MAIN.C */

extern void vis_per(void);

main()

```

```

void vis_per(void)
{
char pausa; int i;

for(i=0; strcmp(anag[i].cognome, TAPPO)!=0; i++) {
    printf("\n\n-----\n");
    printf("\n\t\tCognome : %s", anag[i].cognome);
    printf("\n\t\tNome : %s", anag[i].nome);
    printf("\n\t\tIndirizzo : %s", anag[i].ind);
    printf("\n\t\tEtà : %d", anag[i].eta);
    printf("\n\n-----\n");

    scanf("%c", &pausa);
}
}

```

### Listato 12.7 File VIS\_PER.C

Il nome simbolico `vis_per` dichiarato nel file `MAIN.C` con

```
extern void vis_per(void);
```

non ha all'interno del file una corrispondente definizione. Per tale motivo il nome è dichiarato come `extern`. La parola chiave `extern` si usa per informare il compilatore del fatto che la definizione del simbolo dichiarato `extern` è presente in qualche altro file. Nel nostro esempio la definizione della funzione `vis_per` si trova nel file `VIS_PER.C`.

Possiamo ora definire con esattezza il concetto di variabile globale.

In C una variabile globale è visibile in tutti i file sorgenti che compongono il programma. Per poter far riferimento a una

```

    {TAPPO, TAPPO, TAPPO, 0}
};

void vis_per(void)
{
char pausa; int i;

for(i=0; strcmp(anag[i].cognome, TAPPO)!=0; i++) {
    printf("\n\n-----\n");
    printf("\n\t\tCognome : %s", anag[i].cognome);
    printf("\n\t\tNome : %s", anag[i].nome);
    printf("\n\t\tIndirizzo : %s", anag[i].ind);
    printf("\n\t\tEtà : %d", anag[i].eta);
    printf("\n\n-----\n");

    scanf("%c", &pausa);
}
}

```

### Listato 12.8 Esempio di variabili static

```
static struct per anag[];
```

Allora se nel file MAIN.C tentassimo di far riferimento alla variabile `anag` mediante una dichiarazione del tipo:

```
extern struct per anag[];
```

non otterremmo altro effetto se non quello di provocare un errore in fase di link.

Come abbiamo anticipato, oltre a essere classificate in base al tipo le variabili C sono classificate anche in base alla *classe di memoria*. La classe di memoria è un attributo dell'oggetto, cioè dell'area di memoria, associato a ogni variabile. Le variabili statiche sono quelle che sono definite in un solo file sorgente e che sono definite in un solo file sorgente.

la variabile `a` è “statica” o – come si dice – non automatica. Essa nasce all’atto dell’esecuzione dell’intero programma e resta attiva per tutto il tempo in cui il programma è in esecuzione.

Le variabili non automatiche sono molto usate per tutte le funzioni che hanno necessità di mantenere memoria di un valore che esse stesse hanno definito. Si consideri per esempio il caso di un semplice generatore di numeri casuali (Listato 12.9). L’esecuzione del programma produce il seguente risultato:

```
Il numero casuale 1 è 18625
  Il numero casuale 2 è 16430
  Il numero casuale 3 è 8543
  Il numero casuale 4 è 43172
  Il numero casuale 5 è 64653
  Il numero casuale 6 è 2794
  Il numero casuale 7 è 27083
  Il numero casuale 8 è 3200
  Il numero casuale 9 è 23705
  Il numero casuale 10 è 34534
```

```
#include <stdio.h>

#define FATTORE 25173
#define MODULO 65535
#define INCREMENTO 13849
#define SEME_INIZIALE 8
#define LOOP 10

unsigned rand(void);

void main()
{
```

fine del programma.

Per concludere con la classificazione delle variabili, osserviamo che esiste un altro attributo che può caratterizzare una variabile: l'attributo `register`. Questo è sempre riferito a variabili automatiche e dice al compilatore di allocare il relativo oggetto direttamente nei registri macchina dell'unità di elaborazione centrale (CPU). Per mezzo della direttiva `register`, in teoria, si velocizzano le operazioni sulle relative variabili. Per esempio la funzione:

```
void strcpy(register char *S, register char *t)
{
    while(*s++ = *t++);
}
```

viene in teoria eseguita più velocemente di

```
void strcpy(char *s, char *t)
{
    while(*s++ = *t++);
}
```

In realtà i moderni compilatori sono talmente ottimizzati che da soli provvedono ad allocare nei registri di memoria le variabili più frequentemente usate. Addirittura talvolta, esagerando con il numero delle variabili dichiarate `register`, si può sortire l'effetto contrario e rendere più inefficiente il programma. Infatti il numero dei registri macchina è limitato, e spingere il compilatore ad allocare molte variabili con pochi registri significa perdere tempo in una gestione di scarsa utilità.

```

1      long *pony(long, long);
2      long (*dog)(long, long);
3      struct cat {
4          int a;
5          char b;
6      } (*dog[5]) (struct cat, struct cat);
7      double (*mpuse(double(*)[3]))[3];
8      union rabbit {
9          int x;
10         unsigned y;
11     } **frog[5][5];
12     union rabbit *(*frog[5])[5];

```

## 13.1 Apertura e chiusura di file

Il C, al contrario di altri linguaggi, offre soltanto alcune funzioni di base per operare sui file: non esiste il concetto di record e di chiave di ricerca; non esistono pertanto neppure le classiche funzioni di inserimento, modifica, cancellazione e ricerca di record all'interno di un file. È consentito operare solamente sui file costituiti da sequenze lineari di byte, mediante funzioni per la creazione, rimozione di file, lettura e scrittura di byte da/su file. La prima operazione da eseguire prima di poter scrivere o leggere da file è l'apertura:



	lettura. Se il file al momento dell'apertura non esiste sarà automaticamente creato, in caso contrario il contenuto del file preesistente sarà perso.
"r+"	<i>lettura e scrittura</i> – sul file sarà possibile eseguire operazioni sia di lettura sia di scrittura. Se il file non esiste la funzione <code>fopen</code> ritornerà il codice di errore <code>NULL</code> .
"w+"	<i>scrittura e lettura</i> – sul file sarà possibile eseguire operazioni sia di scrittura sia di lettura . Se il file non esiste verrà automaticamente creato, in caso contrario il contenuto del file preesistente verrà perso.
"a"	<i>append</i> – sul file sarà possibile eseguire soltanto operazioni di scrittura a fine file: tutte le scritture verranno sempre eseguite in coda al contenuto attuale del file. Se il file non esiste verrà creato automaticamente, in caso contrario il contenuto del file preesistente verrà mantenuto.
"a+"	<i>lettura e append</i> – sul file sarà possibile eseguire operazioni sia di lettura sia di scrittura. Se il file non esiste verrà automaticamente creato, in caso contrario il contenuto del file preesistente verrà mantenuto.

## 13.2 Lettura e scrittura su file

Dopo aver imparato come si apre e si chiude un file, vediamo come utilizzare le funzioni che consentono di leggere il contenuto di un file e di modificarlo. Partiamo al solito con un esempio:

```
int elementi, dimensione;
char buf[100];
FILE *fp;
int n;
...
elementi = 100;
dimensione = 1;

n = fread(buf, dimensione, elementi, fp);
```

```

long nc;          /* Contatore caratteri */
int n;            /* Numero caratteri letti con fread() */
int fine_file = 0; /* Indica la fine della lettura del file */

fp = fopen("clienti", "r"); /* Apertura del file clienti */

if( fp == NULL )
/* Si è verificato un errore: il file non esiste */
    printf("Errore : il file ordini non esiste\n");
else {
    nc = 0L;          /* Inizializza il contatore */
    do {              /* Ciclo di lettura */
        /* Legge 100 caratteri dal file ordini */
        n = fread(buf, 1, 100, fp);
        if(n==0)      /* Controllo di fine file */
            fine_file = 1;
        nc += n;      /* Incremento del contatore */
    }
    while(fine_file==0);

    fclose(fp);      /* Chiusura del file clienti */
    printf("Il file clienti contiene %ld caratteri\n", nc);
}
}

```

Listato 13.1 Conta il numero di caratteri nel file `clienti`

Probabilmente i programmatori C avrebbero preferito scrivere il ciclo di lettura in questo modo:

Scriviamo ora due semplici programmi: il primo acquisisce una stringa da tastiera e la memorizza all'interno del file di nome fornitori (Listato 13.2); il secondo copia il contenuto del file ordini nel file ordini.bak (Listato 13.3) ■

```
/* Scrittura di una stringa in un file */

#include <stdio.h>
#include <string.h>

main()
{
    char buf[100];          /* Buffer */
    FILE *fp;               /* File pointer */
    int len;

    /* Legge da tastiera il nome del fornitore */
    printf("Inserisci un fornitore : ");
    scanf("%s",buf);
    len = strlen(buf);
    fp = fopen("fornitori", "w"); /* Crea il file fornitori */

    /* Memorizza il nome del fornitore nel file */
    fwrite(buf, 1, len, fp);
    fclose(fp);             /* Chiude il file */
}
```

Listato 13.2 Programma per l'acquisizione di una stringa da tastiera e sua scrittura in un file

```
    printf("Il file ordinato non esiste\n");  
}
```

## Listato 13.3 Programma per la copia di un file su un altro

### 13.3 Posizionamento del puntatore

In C è possibile operare sui file di byte non solo in modo strettamente sequenziale ma anche con modalità *random*. La funzione `fseek` consente infatti di muovere il puntatore di lettura e/o scrittura in una qualunque posizione all'interno del file.

`fseek` si usa come illustrato nel seguente esempio:

```
int err, mode;  
FILE *fp;  
long n;  
  
mode = 0;  
n = 100L;  
  
err = fseek(fp, n, mode);
```

in cui il file pointer `fp` è posizionato sul centesimo byte. I parametri della funzione `fseek` hanno il seguente significato:

<code>fp</code>	è il file pointer;
<code>n</code>	indica di quanti byte il file pointer deve essere spostato; se <code>n</code> è negativo significa che il file pointer deve essere spostato indietro invece che in avanti;
<code>mode</code>	indica a partire da quale posizione muovere il file pointer; se <code>mode</code> vale 0 significa che lo spostamento è a partire dall'inizio, se vale 1 è dalla posizione corrente e se, infine, vale 2 è a partire dalla fine del

```

    if( fp != NULL ) {
        fseek(fp,0L,2);    /* Puntatore alla fine del file */
        n = ftell(fp);    /* Lettura posizione del puntatore */
        fclose(fp);        /* Chiusura del file */
        printf("La dimensione del file è %ld\n", n);
    }
    else
        printf("Errore : il file %s non esiste\n", argv[1]);
}
}

```

Listato 13.4 Visualizzazione della dimensione di un file con `fseek` e `ftell`

## 13.4 Lettura e scrittura formattata

Nel Capitolo 11 avevamo approfondito l'uso delle funzioni di libreria `printf` e `scanf`, che consentono di scrivere a video e di acquisire dati da tastiera, e introdotto le funzioni `fprintf` e `fscanf`; riprendiamo queste ultime in relazione alla scrittura e lettura su file:

- `fprintf`      scrive sui file in modo formattato
- `fscanf`      legge da file in modo formattato

La funzione `fprintf` si usa come nell'esempio seguente:

```

FILE *fp;
int i;

```

```

char buf[100];
FILE *fp;
...
fputs(buf, fp);

```

dove `buf` è il vettore che contiene la riga da scrivere sul file `fp` terminata con il carattere di fine stringa “\0”.  
 Scriviamo ora un semplice programma di esempio che conta il numero di righe di un file (Listato 13.5).

```

/* Determinazione del numero di linee contenute in un file.
   Ogni linea è definita dal carattere di newline \n */

#include <stdio.h>

main(int argc, char **argv)
{
    char buf[100];
    int linee;
    FILE *fp;

    if( argc < 2 )
        printf("Errato numero di parametri\n");
    else {
        fp = fopen(argv[1], "r");          /* Apre il file */
        if(fp!= NULL) {                   /* Il file esiste? */
            linee = 0;                     /* Inizializza contatore di linea */
            for(;;) {                      /* Ciclo di lettura da file */
                if( fgets(buf,100,fp) == NULL )
                    break;                 /* Fine file */
            }
        }
    }
}

```

```
c = 'A';
fputc(c, fp);
```

dove *c* rappresenta il carattere da scrivere sul file pointer *fp*.

Scriviamo ora un programma che conta il numero di caratteri numerici contenuti in un file (Listato 14.6).

```
/* Determinazione del numero di caratteri numerici
   (cifre decimali) presenti in un file          */

#include <stdio.h>

main(int argc, char **argv)
{
    FILE *fp;
    int c;
    int nc;

    if( argc < 2 )
        printf("Errato numero di parametri\n");
    else {
        fp = fopen(argv[1], "r");
        if(fp != NULL) {
            nc = 0;
            while((c = fgetc(fp)) != EOF)
                if(c >= '0' && c <= '9') nc++;
            fclose(fp);
            printf("Il numero di caratteri numerici è: %d\n", nc);
        }
    }
}
```

pointer:

```
#include <stdio.h>

main()
{
    FILE *fp;

    fp = fopen("ordini", "w");
    fprintf(fp, "Giovanni Fuciletti");

    fflush(fp);
    ...
}
```

Nel caso in cui il programmatore desideri disabilitare la bufferizzazione su di un file pointer può usare la funzione `setbuf`:

```
FILE *fp;
...
setbuf(fp, NULL);
```

dove `fp` è il file pointer, mentre il secondo parametro `NULL` richiede la disabilitazione completa della bufferizzazione.

## 13.5 Procedura anagrafica

Vediamo, a titolo di esempio, la gestione di un'anagrafica che abbiamo già esaminato nel Paragrafo 12.7 parlando di strutture dati composte tramite struttura, questa volta appoggiandoci su file in modo da rendere i dati persistenti (si veda



```

#define MENU      0
#define INS       1
#define CAN       2
#define RIC       3
#define VIS       4
#define OUT      100

/* Semplice struttura che modella una persona */
struct per {
    char cognome[DIM];
    char nome[DIM];
    char ind[DIM];
    int  eta;
};

/* Puntatore al file */
FILE *fp;

/* La variabile di appoggio anag per le operazioni
sul file
struct per anag;
*/

int men_per(void);
void ins_per(void);
void ric_per(void);
void can_per(void);
long cer_per(char *, char *, int);
void eli_per(long pos);
void vis_per(void);
void vis_anagrafe(void);

```

```
int men_per(void)
{
int scelta;
char invio;
int true = 1;

while(true){
    printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
    printf("\t\t\t ANAGRAFE");
    printf("\n\n\n\t\t\t 1. Immissione Persona");
    printf("\n\n\n\t\t\t 2. Cancellazione Persona");
    printf("\n\n\n\t\t\t 3. Ricerca Persona");
    printf("\n\n\n\t\t\t 4. Visualizza anagrafe");
    printf("\n\n\n\t\t\t 0. Fine");
    printf("\n\n\n\t\t\t\t Scegliere una opzione: ");
    scanf("%d", &scelta);
    scanf("%c", &invio);
    printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
    switch(scelta) {
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
            return(scelta);
        default:
            break;
    }
}
return(0);
}
```

```

printf("\n\t\tCANCELLA PERSONA ");
printf("\n\t\t\t-----\n\n");
printf("\nCognome : ");
gets(cognome);
printf("\nNome : ");
gets(nome);
printf("\nEtà : ");
scanf("%d", &eta);
scanf("%c", &pausa);
/* invoca ricerca persona */
pos = cer_per(cognome, nome, eta);
if(pos == -1) {
    printf("\nPersona non presente in anagrafe");
    scanf("%c", &pausa);
    return;
}
/* invoca visualizza persona */
vis_per();
printf("\nConfermi cancellazione ? (S/N) ");
scanf("%c", &pausa);
if(pausa=='S' || pausa=='s') {
    eli_per(pos);
    return;
}
}

/* Elimina persona dall'anagrafe */
void eli_per(long pos)
{
    strcpy(anag.cognome, "");
    strcpy(anag.nome, "");

```

```

vis_per();
scanf("%c", &pausa);
}

/* Effettua una scansione sequenziale del file anag.dat alla ricerca di una
persona che abbia determinati cognome, nome ed età */
long cer_per(char *cg, char *nm, int et)
{
int n;
long pos = 0L;

fp = fopen("anag.dat", "r");

for(;;) {
n = fread(&anag, sizeof(struct per), 1, fp);
if(n==0) {
fclose(fp);
pos = -1;
return (pos);
}
else
if(strcmp(cg, anag.cognome) == 0)
if(strcmp(nm, anag.nome) == 0)
if(et == anag.eta) {
pos = ftell(fp);
fclose(fp);
return(pos-sizeof(struct per));
}
}
}

```

## 13.6 Standard input e standard output

Quando un programma va in esecuzione il sistema apre automaticamente tre file pointer, mediante i quali è possibile scrivere messaggi a video e acquisire dati dalla tastiera. Questi tre file pointer prendono il nome di Standard Input (`stdin`), Standard Output (`stdout`) e Standard Error (`stderr`) e possono essere utilizzati dalle funzioni di accesso ai file descritte nei precedenti paragrafi.

Il file pointer `stdin` è associato dal sistema alla tastiera, i due file pointer `stdout` e `stderr` sono entrambi assegnati al video. Per scrivere un messaggio a video si può allora utilizzare, per esempio, la funzione `fprintf`:

```
#include <stdio.h>
main()
{
    fprintf(stdout, "Ciao lettore\n");
}
```

Dunque le funzioni che abbiamo utilizzato per accettare valori da tastiera e mandare uscite al video corrispondono a usi particolari delle funzioni di uso generale esaminate in questo capitolo. Si hanno le seguenti equivalenze:

<code>printf(...)</code>	->	<code>fprintf(stdout,...)</code>
<code>scanf(...)</code>	->	<code>fscanf(stdin,...)</code>
<code>getchar()</code>	->	<code>fgetc(stdin)</code>
<code>putchar(c)</code>	->	<code>fputc(c, stdout)</code>
<code>eof()</code>	->	<code>feof(stdin)</code>

La sintassi delle funzioni a sinistra è più sintetica, perché quelle sulla destra devono specificare che desiderano operare sullo standard input o sullo standard output.

Il programmatore deve prestare molta attenzione all'utilizzo delle due funzioni con sintassi abbreviata `gets` e `puts`, il

```
main()
{
int fd;
fd = open("clienti", O_RDONLY);
...
```

Tale funzione associa il file descriptor `fd` al file di nome `clienti` aprendolo in modalità di sola lettura. Il valore di ritorno della `open` può essere negativo nel caso in cui si sia verificato un errore, per esempio se il file `clienti` non esiste. L'uso generale della funzione `open` su `clienti` è:

```
int fd, modo, diritti;
...
fd = open("clienti", modo [diritti] );
```

dove *modo* rappresenta la modalità di apertura del file e può avere il valore di una o più delle seguenti costanti simboliche:

<code>O_RDONLY</code>	apre il file in sola lettura; se il file non esiste la <code>open</code> ritorna errore
<code>O_WRONLY</code>	apre in file in sola scrittura; se il file non esiste la <code>open</code> ritorna errore
<code>O_RDWR</code>	apre il file in lettura e scrittura; se il file non esiste la <code>open</code> ritorna errore
<code>O_CREAT</code>	crea il file se non esiste
<code>O_TRUNC</code>	distrugge il contenuto del file preesistente
<code>O_APPEND</code>	apre il file in modalità append; tutte le scritture sono automaticamente eseguite a fine file
<code>O_EXCL</code>	la <code>open</code> ritorna errore se il file esiste già al momento dell'apertura

Se il programmatore desidera specificare più di una modalità di apertura lo può fare utilizzando l'operatore binario di OR bit a bit; per esempio

```
fd = open("clienti", O_WRONLY | O_TRUNC);
```

Dunque con 640 abbiamo i permessi di lettura e scrittura per il proprietario (4 permesso di lettura + 2 permesso di scrittura), di sola lettura per il gruppo e nessun permesso agli altri.

Vediamo un esempio dove definiamo più di una modalità e anche i diritti: il codice

```
int fd;  
fd = open("clienti", O_RDWR | O_CREAT | O_TRUNC, 0640);
```

crea il file `clienti` aprendolo in lettura e scrittura con diritti di lettura/scrittura per il proprietario (6), con soltanto diritto di lettura per il gruppo di utenti cui appartiene il proprietario (4) e con nessun diritto per tutti gli altri utenti (0). La funzione `close` ha un comportamento analogo a quello della funzione `fclose`: chiude un file descriptor aperto dalla `open`:

```
int fd;  
...  
close(fd);
```

Per quanto riguarda le operazioni di lettura e scrittura su file con utilizzo dei file descriptor, le funzioni che le eseguono si chiamano rispettivamente `read` e `write`. La funzione `read` opera nel modo seguente:

```
char buf[1000];  
int elementi;  
int fd;  
int n;  
...  
elementi = 1000;  
n = read(fd, buf, elementi);
```

dove `fd` è il file descriptor da cui si desidera leggere, `buf` è il vettore dove i dati letti devono essere trasferiti ed

```
main(argc,argv)
int argc;
char **argv;
{
static char buf[BUFSIZ];
int fdin, fdout, n;

if( argc != 3 ) {
    printf("Devi specificare file sorgente e destinazione\n");
    exit(1);
}

/* Apre il file sorgente */
fdin = open(argv[1],O_RDONLY);
if( fdin < 0 ) {
    printf("Non posso aprire %s\n",argv[1]);
    exit(2);
}

/* Apre il file destinazione */
fdout = open(argv[2],O_WRONLY|O_CREAT|O_TRUNC,0600);
if( fdout < 0 ) {
    printf("Non posso creare %s\n", argv[2]);
    close(fdin);
    exit(3);
}

/* Esempio ciclo di lettura e scrittura */
```



Se `n` è negativo lo spostamento del puntatore avviene all'indietro invece che in avanti. Il parametro `mode` indica a partire da quale posizione iniziare a muovere il puntatore: se `mode` vale 0 significa che ci si deve muovere a partire dall'inizio del file, se vale 1 a partire dalla posizione corrente e infine se vale 2 a partire dalla fine. Il valore di ritorno della `lseek` contiene la posizione corrente del puntatore dopo lo spostamento. Allora:

```
lseek (fd, 0L, 1)      restituisce la posizione corrente
lseek (fd, 0L, 2)      restituisce la dimensione del file in byte
```

Utilizzando i file pointer, avevamo visto come il sistema offriva tre file pointer automaticamente aperti al momento del lancio del programma (`stdin`, `stdout`, `stderr`) mediante i quali era possibile lavorare su tastiera e video. Anche usando i file descriptor il sistema mette a disposizione tre descrittori aperti per default al momento del lancio del programma e associati a tastiera e video:

```
0      standard input associato a tastiera
1      standard output associato al video
2      standard error associato al video
```

Questi tre numeri interi possono essere utilizzati dal programmatore per leggere e scrivere da tastiera e video senza dover eseguire le relative `open`. Scriviamo dunque un programma di esempio che memorizza all'interno di un file informazioni su un gruppo di alunni inserite da tastiera (Listato 13.9).

```
/* Memorizza in un file le informazioni passate dall'utente sugli alunni di un
classe */

#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

/* La struttura alunno contiene nome, cognome */
```

```

    printf("Eta: ");
    scanf("%d", &alunno.eta);
    write(fd, &alunno, sizeof(struct alunno));
}
close(fd);
}

```

Listato 13.9 Programma che memorizza all'interno di un file informazioni su un gruppo di alunni inserite da tastiera

## 13.8 Embedded SQL

I programmi C possono interagire con i sistemi di gestione di basi di dati attraverso il linguaggio SQL (*Structured Query Language*), in modalità *embedded* (letteralmente: “incastrato”), cioè accogliendo al loro interno pezzi di codice SQL, passando dei parametri e ricevendo dei risultati.

SQL è il linguaggio standard, utilizzato dai DBMS relazionali (RDBMS), che consente l'esecuzione di tutte le operazioni necessarie nella gestione e nell'utilizzo di una base di dati; permette infatti, oltre alla consultazione del database, anche la definizione e gestione dello schema, la manipolazione dei dati e l'esecuzione di operazioni amministrative.

### ✓ NOTA

Con il termine database (base di dati) si indica un insieme di dati rivolti alla rappresentazione di uno specifico sistema informativo di tipo aziendale, scientifico, amministrativo o altro, mentre con sistema di gestione di base di dati o Data Base Management System (DBMS) ci si

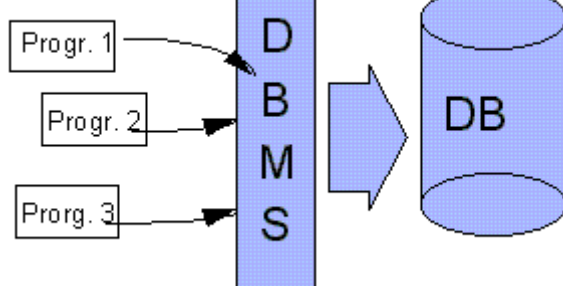


Figura 13.1 I programmi accedono ai dati attraverso il gestore di basi di dati

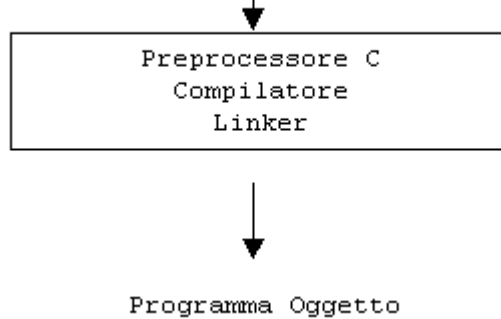


Figura 13.2 La prima fase dell'interazione fra C e DBMS è l'espansione delle direttive Embedded SQL

```
/* Esempio di Embedded SQL */

exec sql include sqlca;

main()
{
/* Dichiarazione delle variabili */
exec sql begin declare section;
CS_CHAR utente[31]; password[31];
exec sql end declare section;

/* Inizializzazione del sottoprogramma che
   gestisce gli errori SQL */
exec sql whenever sqlerror perform err p();
```

Nel programma del Listato 13.10 inizialmente viene incluso il file `sqlca` con la direttiva

```
exec sql include sqlca;
```

in modo analogo a come facciamo con la direttiva `#include` del preprocessore C. Il file `sqlca` contiene strutture dati necessarie al sistema per eseguire i comandi SQL. Successivamente vengono dichiarate le variabili `utente` e `password` di tipo array di `CS_CHAR`, che è equivalente in C a un array di `char`, dunque a una stringa.

```
exec sql begin declare section;  
CS_CHAR utente[31]; password[31];  
exec sql end declare section;
```

Il programma richiede l'identificativo utente e la password e la comunica al database con altra direttiva:

```
exec sql connect :user identified by :password;
```

Per ragioni di sicurezza, infatti, si inizia normalmente una sessione di lavoro facendosi riconoscere dal gestore di basi di dati. Ed ecco che finalmente eseguiamo un'operazione direttamente sulla base di dati con un comando SQL:

```
exec sql update auto set prezzo = prezzo *1.10;
```

Il comando `update` (aggiorna) modifica la tabella `auto` aumentando il valore della colonna `prezzo` del 10%. L'ultima direttiva chiude la connessione con il database:

```
exec sql disconnect;
```

risultato.

\* 5. Aprire un file ASCII, per esempio `autoexec.bat`, e leggere e visualizzare i primi 10 gruppi di caratteri separati da blank e newline.

6. Scrivere un programma che permetta di gestire una rubrica telefonica in modo che i dati vengano memorizzati in forma permanente sul file `rubrica`. Offrire all'utente un menu con le opzioni: inserimento, modifica, cancellazione e visualizzazione dell'intera rubrica.

7. Scrivere una funzione che, dato in ingresso il nome, cerchi in `rubrica` il corrispondente numero di telefono. Aggiungere al menu del programma dell'Esercizio 6 l'opzione che richiama tale procedura.

8. Scrivere una funzione che permetta l'ordinamento del file `rubrica` rispetto al nome. Aggiungere al menu del programma dell'Esercizio 6 l'opzione che richiama tale procedura.

9. Scrivere una funzione che permetta, una volta che il file `rubrica` è ordinato per nome, di effettuare una ricerca binaria. Aggiungere al menu del programma dell'Esercizio 6 l'opzione che richiama tale procedura.

10. Scrivere una procedura che visualizzi tutti i dati delle persone del file `rubrica` i cui nomi iniziano con una lettera richiesta all'utente. Aggiungere al menu del programma dell'Esercizio 6 l'opzione che richiama tale procedura.

11. Modificare il programma dell'Esercizio 6 in modo tale che il file `rubrica` contenga, oltre al nome e al numero di telefono, anche il cognome e l'indirizzo (via, CAP, città e stato) dei conoscenti memorizzati.

12. Un'azienda vuole memorizzare nel file `dipendenti` i dati relativi a ogni dipendente. In particolare si vogliono archiviare nome, cognome, sesso (M,F), anno di nascita e città di residenza. Scrivere un programma che crei un tale file e memorizzi i dati relativi ad alcuni dipendenti.

13. Rispetto al file dell'esercizio precedente scrivere le funzioni per stampare:

Le strutture dati individuate per la risoluzione dei problemi sono dette “astratte”. Una *struttura dati astratta* definisce l’organizzazione delle informazioni indipendentemente dalla traduzione in uno specifico linguaggio di programmazione; la rappresentazione che una struttura astratta prende in un linguaggio di programmazione viene detta *implementazione*.

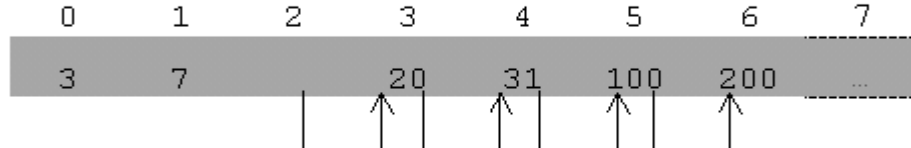
Le strutture astratte dipendono dal problema in considerazione; alcune di esse, come le pile, le code e gli alberi, rivestono però un’importanza generale per l’uso comune che se ne fa nella soluzione di intere classi di problemi. Nella fase di analisi il programmatore si può accorgere che proprio una di queste strutture fa al caso suo; successivamente, dovrà scegliere la tecnica migliore per implementarle. Per facilitare il lavoro del programmatore, il C mette a disposizione una serie di strumenti, tra i quali gli array, le `struct` e i puntatori. In questo capitolo e nel seguente faremo pratica di programmazione implementando alcune tra le più importanti strutture dati.

Fino qui, i dati omogenei sono stati memorizzati con l’array. Questi ultimi, però, possono porre dei problemi, in merito a:

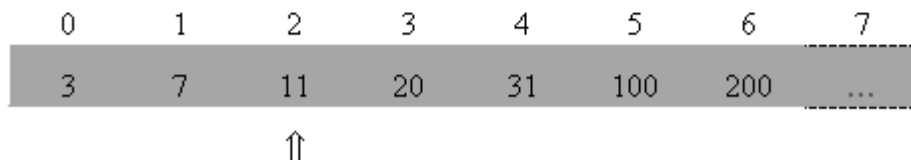
- • occupazione di memoria;
- • velocità di esecuzione;
- • chiarezza della soluzione proposta.

Consideriamo in primo luogo l’*occupazione di memoria*. Il numero di elementi di un array viene definito nelle parti dichiarative del programma. È essenziale dunque, in fase di analisi del problema, effettuare una stima sul massimo numero di elementi che eventualmente potranno essere utilizzati. In molti casi non è possibile dare una valutazione esatta, per cui si dovrà sovrastimare l’array, cioè definirlo con un numero di elementi probabilmente molto superiore al necessario, occupando più memoria di quella realmente indispensabile.

Per esempio, si supponga di elaborare le prenotazioni di 700 studenti delle scuole medie relativamente a una delle tre gite scolastiche previste nell’anno: la prima a Vienna, la seconda a Parigi e la terza a Roma. Per formare le tre liste di studenti si possono utilizzare tre array di caratteri: Vienna, Parigi e Roma. Ognuno di essi dovrà essere definito con 700 elementi, perché non si conosce a priori la distribuzione delle scelte degli studenti, per cui si deve considerare il caso peggiore, in cui tutti gli studenti scelgono la stessa gita. In questo caso si avrà una occupazione di memoria pari (o superiore, se qualche studente non partecipa a nessuna gita) a tre volte quella effettivamente richiesta.



b) Spostamento dei valori in posizioni successive al punto di inserimento



c) Inserimento del valore 11

Figura 14.1 Inserimento ordinato di un elemento nella sequenza memorizzata in un array.

Consideriamo infine il terzo fattore: la *chiarezza della soluzione proposta*. Più l'organizzazione della struttura corrisponde alla logica di utilizzazione delle informazioni in essa contenute, più è chiara la soluzione. Questo fattore ha un riflesso determinante sulla bontà della programmazione, in termini di diminuzione dei tempi di revisione e modifica dei programmi. In sintesi, i limiti che vengono ravvisati nell'utilizzazione degli array si collegano alla poca elasticità/flessibilità insita in questa struttura. Non sempre l'array corrisponde alla scelta migliore.



La parte informazione dell'elemento dipende dal tipo di dati che stiamo trattando. In C può essere costituita da uno qualsiasi dei tipi semplici che conosciamo: `int`, `float` ecc. Nel caso della lista lineare di Figura 14.2 il cui campo informazione è di tipo intero, la dichiarazione della struttura di ogni elemento può essere la seguente:

```
struct elemento {
    int inf;
    struct elemento *pun;
};
```

dove `inf` è di tipo `int` e `pun` è un puntatore a una struttura di tipo `elemento`. La scelta del nome della struttura e dei nomi dei campi, in questo caso `elemento`, `inf` e `pun`, è libera, nell'ambito degli identificatori ammessi dal linguaggio. La precedente dichiarazione descrive la struttura di `elemento` ma non alloca spazio in memoria. La definizione:

```
struct elemento *punt_lista;
```

stabilisce che `punt_lista` è un puntatore che può riferirsi a variabili `elemento`. Non esistono variabili puntatore in generale, ma variabili puntatore che fanno riferimento a oggetti di un determinato tipo.

La parte informazione può essere anche composta da più campi, ognuno dei quali di un certo tipo. Per esempio, se si desiderano memorizzare nella lista nomi ed età degli amici la parte informazione diventa:

```
informazione = nome + anni
```

con `nome` di tipo array di `char` e `anni` di tipo `int`. La dichiarazione di un elemento diventa allora:

```
struct amico {
    int anno;
    char nome[30];
    struct amico *pun;
};
```

mentre la definizione di un puntatore alla struttura `amico` è:

Si affida quindi la soluzione dei due sottoproblemi a due funzioni, la cui dichiarazione prototype è:

```
struct elemento *crea_lista();  
void visualizza_lista(struct elemento *);
```

Nel main viene definito il puntatore che conterrà il riferimento al primo elemento della lista:

```
struct elemento *punt_lista;
```

Le due funzioni vengono chiamate in sequenza dallo stesso main; il programma completo è riportato nel Listato 14.1.

```
punt_lista = crea_lista();  
visualizza_lista(punt_lista);
```

La procedura `crea_lista` restituisce al main il puntatore alla lista che è assegnato a `punt_lista` e che viene successivamente passato a `visualizza_lista`.

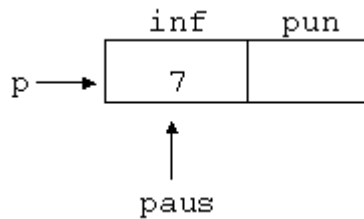
La funzione `crea_lista` è di tipo puntatore a strutture elemento. Non prevede il passaggio di valori dal chiamante, per cui una sua dichiarazione più appropriata sarebbe stata:

```
struct elemento *crea_lista(void);
```

dove `void` esplicita la mancanza di parametri. Nel nostro caso, dunque, il compilatore potrebbe segnalare un *warning* (attenzione!).

La funzione `crea_lista` deve comprendere la dichiarazione di `p`, puntatore alla testa della lista, e di `paus`, puntatore ausiliario, che permette la creazione degli elementi successivi al primo, senza perdere il puntatore iniziale alla lista. In primo luogo si deve richiedere all'utente di inserire il numero di elementi da cui è composta la lista. Questa informazione viene memorizzata nella variabile `n`, di tipo `int`. Dopo di ciò, se `n` è uguale a zero, si assegna a `p` il valore `NULL`, che corrisponde a lista vuota. In questo caso il sottoprogramma termina. Si osservi che `n` è una variabile locale alla funzione.

Se il numero di elementi (valore di `n`) è maggiore di zero `crea_lista` deve creare il *primo elemento*:



Finita la gestione del primo elemento, deve iniziare un ciclo per la creazione degli *elementi successivi al primo*. Questo ciclo si ripete  $n-1$  volte, dove  $n$  è la lunghezza della sequenza in ingresso. Il ciclo è così costituito:

```
for(i = 2; i<=n; i++) {
    a) crea il nuovo elemento concatenato al precedente
    b) sposta di una posizione avanti paus
    c) chiedi all'utente la nuova informazione della sequenza; inserisci l'informazione nel campo inf del nuovo
    elemento
}
```

Nel caso dell'esempio proposto, il ciclo si ripete due volte (da  $i=2$  a  $i=n=3$ ); a ogni iterazione le operazioni descritte corrispondono a:

```
a) paus->pun = (struct elemento *)malloc(sizeof(struct elemento));
b) paus = paus->pun;
c) printf("\nInserisci la %d informazione: ", i);
    scanf("%d", &paus->inf);
```

In a) viene creato un nuovo elemento di lista, connesso al precedente. In questo caso il puntatore ritornato dall'operazione di allocazione di memoria e successivo *cast* viene assegnato al campo puntatore della variabile puntata

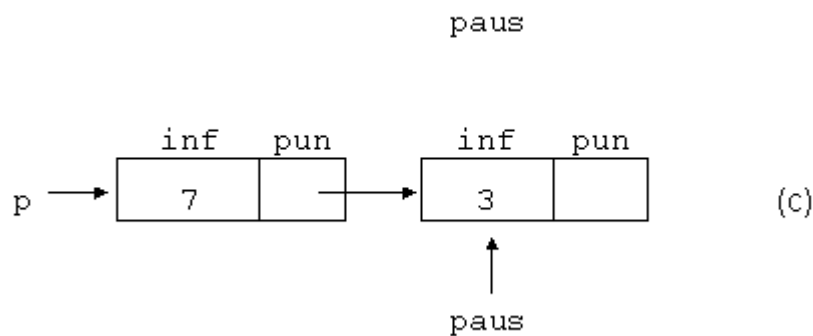


Figura 14.3 Sequenza di creazione del secondo elemento della lista

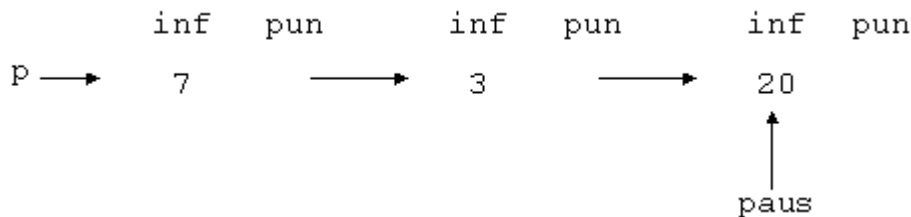


Figura 14.4 Situazione dopo l'ultimo ciclo

Infine la funzione assegna al campo puntatore dell'ultimo elemento il valore `NULL` e termina passando al chiamante il valore di `p`, cioè il puntatore alla lista:

permesso perché abbiamo avuto cura di porre il segnale di fine lista nella funzione `crea_lista`. La scansione degli elementi è consentita dall'operazione di assegnamento:

```
p = p->pun;
```

Si provi a eseguire manualmente l'algoritmo di scansione sulla lista di Figura 14.5. La procedura funziona anche nel caso particolare di lista vuota. Non c'è bisogno di un puntatore ausiliario, dato che il riferimento all'inizio della lista è nella variabile `punt_lista` del `main`, mentre `p` è una variabile locale al sottoprogramma ■.

```
/* Accetta in ingresso una sequenza di interi e li memorizza in
   una lista. Il numero di interi che compongono la sequenza
   è richiesto all'utente. La lista creata viene visualizzata */
```

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

```
/* struttura degli elementi della lista */
struct elemento {
    int inf;
    struct elemento *pun;
};
```

```
struct elemento *crea_lista();
void visualizza_lista(struct elemento *);
```

```
main()
{
    struct elemento *punt_lista; /* Puntatore alla testa
```

```

    paus->pun = (struct elemento *)malloc(sizeof(struct elemento));
    paus = paus->pun;
    printf("\nInserisci la %d informazione: ", i);
    scanf("%d", &paus->inf);
}
paus->pun = NULL;          /* Marca di fine lista */
}
return(p);
}

/* Funzione per la visualizzazione della lista.
   Il parametro in ingresso è il puntatore alla testa */
void visualizza_lista(struct elemento *p)
{
    printf("\npunt_lista---> ");

    /* Ciclo di scansione della lista */
    while(p!=NULL) {
        printf("%d", p->inf);    /* Visualizza il campo informazione */
        printf("---> ");
        p = p->pun;              /* Scorri di un elemento in avanti */
    }
    printf("NULL\n\n");
}

```

Listato 14.1 Creazione e visualizzazione di una lista

entrata.

Si richiede all'utente di inserire la prima informazione: se questa è uguale a zero, si assegna a p il valore NULL e la funzione ha termine:

```
printf("\nInserisci un'informazione (0 per fine lista): ");
scanf("%d", &x.inf);
if(x.inf==0) p = NULL;
```

Nel caso in cui il valore della prima informazione in entrata sia diverso da zero si crea il primo elemento di lista, si assegna x.inf e si inizializza il puntatore ausiliario paus al valore p (punt\_lista):

```
p = (struct elemento *)malloc(sizeof(struct elemento));
p->inf = x.inf;
paus = p;
```

Inizia poi il ciclo di ingresso delle altre informazioni. Se il valore dell'informazione è diverso da zero si procede alla creazione di un nuovo elemento di lista e all'inserimento dell'informazione nel corrispondente campo:

```
/* creazione dell'elemento successivo */
paus->pun = (struct elemento *)malloc(sizeof(struct elemento));

paus = paus->pun; /* Attualizzazione di paus */
paus->inf = x.inf; /* Inserimento dell'informazione
                  nell'elemento */
```

La soluzione è simile a quella vista per la visualizzazione. Si tratta di scandire la lista per cercare il valore maggiore. Il main passa alla funzione il puntatore alla lista punt\_lista e il valore di ritorno della funzione è il maggiore che viene visualizzato:

```
printf("\nIl maggiore è: %d", maggiore_lista(punt_lista));
```

```

void visualizza_lista(struct elemento *);
int maggiore_lista(struct elemento *);

main()
{
    struct elemento *punt_lista; /* Puntatore alla testa
                                   della lista */
    punt_lista = crea_lista2(); /* Chiamata funzione per
                                   creare la lista */
    visualizza_lista(punt_lista); /* Chiamata funzione per
                                   visualizzare la lista */
    /* Stampa il valore di ritorno della funzione maggiore_lista() */
    printf("\nIl maggiore e': %d\n\n", maggiore_lista(punt_lista));
}

/* Accetta in ingresso una sequenza di interi e li
   memorizza in una lista. Il numero di interi che compongono
   la sequenza termina con il valore zero */
struct elemento *crea_lista2()
{
    struct elemento *p, *paus;
    struct elemento x;

    printf("\nInserisci un'informazione (0 per fine lista): ");
    scanf("%d", &x.inf);

    if(x.inf==0) p = NULL; /* Lista vuota */
    else {

```



```

/* Ciclo di scansione della lista */
while(p != NULL) {
    if(p->inf > max)
        max = p->inf;
    p = p->pun;          /* Scorre di un elemento in avanti */
}
return(max);
}

/* Visualizza la lista */
void visualizza_lista(struct elemento *p)
{
    printf("\npunt_lista---> ");

    /* Ciclo di scansione della lista */
    while(p!=NULL) {
        printf("%d", p->inf);    /* Visualizza il campo informazione */
        printf("---> ");
        p = p->pun;              /* Scorre di un elemento in avanti */
    }
    printf("NULL\n\n");
}

```

Listato 14.2 Programma che crea, visualizza la lista e ne determina il maggiore. L'immissione dei valori da parte dell'utente termina con zero

Per costruire la lista risultato della somma delle prime due costruiremo la funzione `somma_liste`, Quest'ultima riceve in ingresso i puntatori alle due liste da sommare e restituisce il puntatore alla nuova lista:

```
punt_lista3 = somma_liste(punt_lista1, punt_lista2);
```

Utilizzeremo la procedura `visualizza_lista` per visualizzarla:

```
visualizza_lista(punt_lista3);
```

Nel Listato 14.3 viene riportato il main e la funzione `somma_liste`.

Si tratta di scandire in parallelo le due liste, sommare il campo informazione degli elementi corrispondenti, creare un elemento della terza lista e inserirvi il risultato ottenuto. Si effettueranno somme finché una delle due liste non termina. I parametri attuali `punt_lista1`, `punt_lista2` diventano i parametri formali `p1`, `p2` e il valore di ritorno `punt_lista3` è il puntatore alla lista somma delle precedenti:

```
punt_lista3 = somma_liste(punt_lista1, punt_lista2);
```

Si utilizzano due puntatori `p1` e `p2` per visitare le due liste, `p3` per creare il primo elemento della terza lista e `paus3` per creare i successivi. Il ciclo ha termine quando `paus1` o `paus2` diventa uguale a `NULL`. Il puntatore `p3` alla terza lista viene restituito al programma chiamante.

```
/* Crea due liste e le visualizza. Somma gli elementi  
corrispondenti delle due liste, inserisce il risultato  
in una terza lista e la visualizza */
```

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

```

/* Somma gli elementi corrispondenti di due liste
e inserisce il risultato in una terza lista */
struct elemento *somma_liste(struct elemento *p1, struct elemento *p2)
{
    struct elemento *p3 = NULL, *p3aus;

    if(p1!=NULL && p2!=NULL) {
        /* Creazione primo elemento */
        p3 = (struct elemento *)malloc(sizeof(struct elemento));
        p3->inf = p1->inf + p2->inf;          /* somma */
        p3aus = p3;                         /* p3aus punta III lista */
        p1 = p1->pun;                        /* Scorrimento I lista */
        p2 = p2->pun;                        /* Scorrimento II lista */

        /* Creazione elementi successivi */
        for(; p1!=NULL && p2!=NULL;) {
            p3aus->pun = (struct elemento *)malloc(sizeof(struct elemento));
            p3aus = p3aus->pun;              /* Scorrimento III lista */
            p3aus->inf = p1->inf + p2->inf;    /* Somma */
            p1 = p1->pun;                    /* Scorrimento I lista */
            p2 = p2->pun;                    /* Scorrimento II lista */
        }
        p3aus->pun = NULL;                  /* Marca di fine III lista */
    }
    return(p3);                          /* Ritorno del puntatore alla III lista */
}

/* ATTENZIONE: devono essere aggiunte le definizioni delle

```

```

        vis2_lista(p->pun);
    }
    else
        printf("NULL");
}

```

L'ultima chiamata avviene quando la lista è terminata e la funzione stampa NULL per indicare questo fatto.

Analoghe considerazioni si possono fare per la funzione che determina il maggiore della lista, dato che anche in questo caso si tratta di scandirla. La differenza consiste nel fatto che la variabile `max` che contiene il maggiore deve essere inizializzata fuori dalla funzione, prima della sua chiamata:

```

maggiore = INT_MIN;
printf("\nIl maggiore è: %d", mag2_lista(punt_lista, maggiore));

```

Ovviamente maggiore è una variabile intera:

```

mag2_lista(struct elemento *p, int max)
{
    if(p!=NULL) {
        if(p->inf>max)
            max = p->inf;
        max = mag2_lista(p->pun, max);
    }
    return(max);
}

```

D F C <--- testa della pila

Si è cioè prelevato dalla sequenza l'ultimo elemento inserito, quello in testa alla pila. Se poi si viene incaricati di un ulteriore prelievo, la sequenza diventa:

D F <--- testa della pila

L'arrivo del vagone codificato con il carattere A corrisponde invece a un inserimento in testa alla pila:

D F A <--- testa della pila

Dunque, come detto, le inserzioni e le estrazioni avvengono sempre in testa alla pila. Un altro esempio di pila è dato dalle bambole russe, le famose *matrioske*, che vengono chiuse una sull'altra, in un ordine che va dalla più piccola alla più grande. Se consideriamo la sequenza delle bambole già racchiuse, ci rendiamo conto che possiamo effettuare solo due operazioni: inserire una bambola più grande in testa alla sequenza ed eliminare la più grande delle bambole presenti nella sequenza, appunto quella che è stata inserita per ultima.

## 14.8 Gestione di una pila mediante array

Passiamo ora a considerare il problema seguente: far gestire all'utente una struttura astratta di tipo pila, per mezzo delle operazioni di inserimento ed eliminazione di elementi, e visualizzare la pila stessa. La parte informazione di un elemento della pila è costituito da un valore intero. Bisogna implementare la pila per mezzo di un array. La soluzione del problema ha una notevole valenza didattica, perché mostra visivamente la logica di funzionamento di una pila.

Dovendo utilizzare un array, è necessario stimare il numero massimo di elementi che la pila può contenere. Si utilizzerà la variabile di tipo intero `punt_testa` come indice (riferimento) alla testa della pila, cioè all'elemento dove si effettuerà il prossimo inserimento (Figura 14.6). Se si suppone che sia  $n=5$ , `punt_testa` potrà assumere valori compresi da 0 a 5, dove 0 indicherà pila vuota, 5 pila piena. Se `pila` è il nome dell'array le dichiarazioni

Una prima suddivisione del problema porta a individuare i seguenti sottoproblemi:

- • inserzione in testa alla pila (*push*);
- • eliminazione in testa alla pila (*pop*);
- • visualizzazione della pila.

Nel problema si richiede che l'utente gestisca la pila, ovvero si possano effettuare le tre operazioni in qualsiasi sequenza, un numero qualunque di volte. Per facilitare il compito si visualizzano le opzioni in un menu con le possibili scelte:

ESEMPIO UTILIZZO STRUTTURA ASTRATTA: PILA

1. Per inserire un elemento
2. Per eliminare un elemento
3. Per visualizzare la pila
0. Per finire

Scegliere una opzione:

L'utente deve inserire il numero corrispondente all'opzione desiderata (1 per inserire, 2 per eliminare ecc.). La funzione `gestione_pila` presiede al trattamento della scelta:

```
while(scelta!=0) {  
    visualizza menu;  
    leggi l'opzione dell'utente nella variabile scelta;  
    switch(scelta) {  
        case 1:  
            // inserimento  
        case 2:  
            // eliminazione  
        case 3:  
            // visualizzazione  
        case 0:  
            // fine  
    }
```

punt\_testa ha valore zero. Quindi un ultimo sottoproblema consiste nel determinare se la pila è vuota. Queste osservazioni permettono di dettagliare gestione\_pila:

```
void gestione_pila(void)
{
    int pila[LUN_PILA];
    int punt_testa = 0;
    int scelta = -1, ele;
    char pausa;

    while(scelta!=0) {
        visualizza menu
        leggi l'opzione dell'utente nella variabile scelta;
        switch(scelta) {
            case 1:
                Se (la pila è piena):
                    l'inserimento è impossibile;
                Altrimenti:
                    leggi l'informazione da inserire;
                    Esegui l'inserimento in testa alla pila;
                break;
            case 2:
                Se (la pila è vuota):
                    l'eliminazione è impossibile;
                Altrimenti:
                    esegui l'eliminazione in testa alla pila;
                    visualizza l'informazione eliminata;
                break;
            case 3:
```

Inserisce il valore di `ele` in `pila[*p]`, incrementa di uno il valore del puntatore alla testa e lo restituisce al chiamante (Figura 14.8). Si ricordi che `punt_testa` è passato per indirizzo e quindi per far riferimento al suo valore si deve scrivere `*p`.



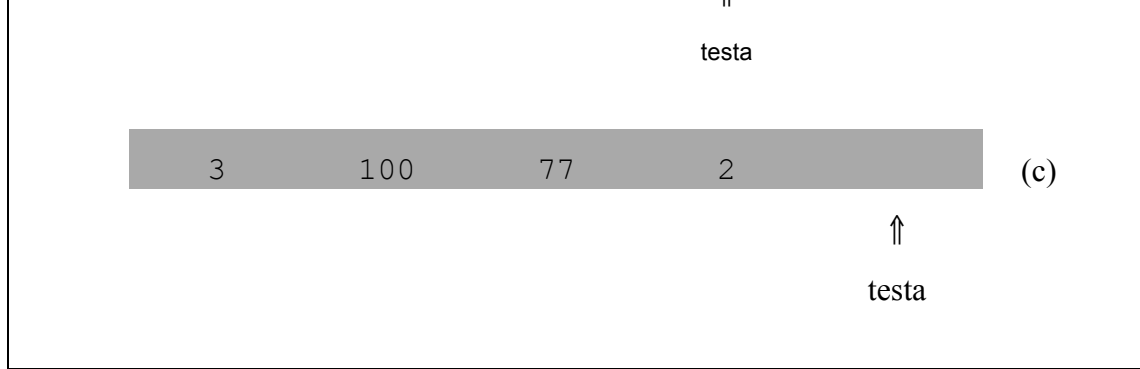


Figura 14.8 a) Stato della pila prima dell'inserimento. b) Si inserisce la nuova informazione (ele=2) nell'elemento dell'array indicato da \*p (punt\_testa). c) Si incrementa di uno il valore di \*p (punt\_testa)

Alla funzione di eliminazione devono essere passati gli stessi parametri della funzione inserimento:

```
punt_testa = eliminazione(pila, &punt_testa, &ele);
```

Questa volta il parametro ele viene passato per indirizzo in quanto la procedura restituisce il valore dell'elemento eliminato al main:

```
eliminazione(int *pila, int *p, int *ele)
```

La funzione decrementa di uno il valore di \*p, assegna il valore dell'elemento in testa alla pila a ele e restituisce il nuovo puntatore alla testa:

```
--*p;
```

Figura 14.9 a) Stato della pila prima della eliminazione. b) Si decrementa di uno il valore di \*p (punt\_testa).  
c) L'informazione presente in testa è assegnato alla variabile ele

Per controllare se la pila è piena è sufficiente verificare che il puntatore alla testa sia uguale a  $n$ , nell'esempio  $n=5$ . Effettuiamo questo controllo all'interno di gestione\_pila:

```
if(punt_testa == LUN_PILA)
    inserimento impossibile, pila piena
```

Per verificare se la pila è vuota invochiamo la funzione pila\_vuota, passandole punt\_testa:

```
if(pila_vuota(punt_testa) )
    eliminazione impossibile, pila vuota
else
    eliminazione dell'elemento in testa
```

L'if risponde falso solamente quando l'espressione è uguale a 0, nel qual caso è possibile procedere a una eliminazione. In effetti la funzione controlla se il puntatore alla testa è uguale a 0, nel qual caso restituisce 1:

```
if (p==0)
    return(1);
else
    return(0);
```

Per quel che riguarda il sottoproblema 5 – la visualizzazione della pila – si deve percorrere il vettore utilizzando un indice che va da 1 al valore del puntatore alla testa della pila, stampando di volta in volta il suo valore.

```

while (scelta != 0) {
    printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
    printf("\t\tESEMPIO UTILIZZO STRUTTURA ASTRATTA: PILA");
    printf("\n\n\n\t\t\t1. Per inserire un elemento");
    printf("\n\n\n\t\t\t2. Per eliminare un elemento");
    printf("\n\n\n\t\t\t3. Per visualizzare la pila");
    printf("\n\n\n\t\t\t0. Per finire");
    printf("\n\n\n\t\t\t\t\tScegliere una opzione: ");
    scanf("%d", &scelta);
    printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");

    switch( scelta ) {
        case 1:
            if(punt_testa >= LUN_PILA) {
                printf("Inserimento impossibile: ");
                printf("memoria disponibile terminata");
                printf("\n\n Qualsiasi tasto per continuare...");
                scanf("%c%c", &pausa, &pausa);
            }
            else {
                printf("Inserire un elemento: ");
                scanf("%d", &ele);
                punt_testa = inserimento(pila, &punt_testa, ele);
            }
            break;
        case 2:
            if(pila_vuota(punt_testa)) {
                printf("Eliminazione impossibile: pila vuota");
                printf("\n\n Qualsiasi tasto per continuare...");
            }
            else {
                printf("Eliminare un elemento: ");
                scanf("%d", &ele);
                punt_testa = eliminazione(pila, &punt_testa, ele);
            }
            break;
        case 3:
            printf("Visualizzare la pila");
            printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
            visualizza(pila, punt_testa);
            printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
            break;
        case 0:
            printf("Fine");
            break;
    }
}

```

```

    if (*p == 0)
        return (*p);
    }

    eliminazione(int *pila, int *p, int *ele)
    {
        --*p;
        *ele = pila[*p];
        return (*p);
    }

    int pila_vuota(int p)
    {
        if (p == 0)
            return (1);
        else
            return (0);
    }

```

Listato 14.4 Gestione di una pila implementata mediante un array

## 14.9 Gestione di una pila mediante lista lineare

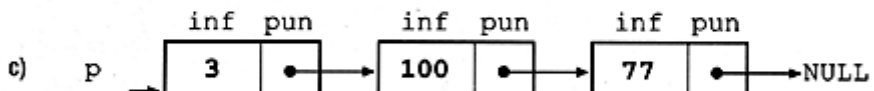
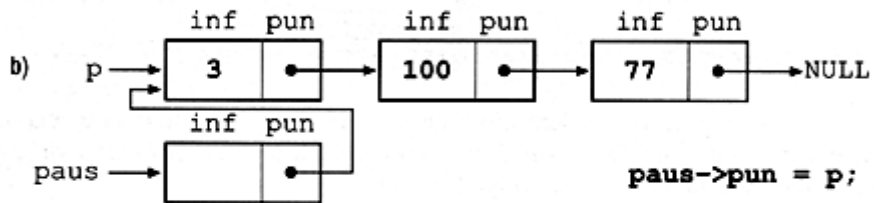
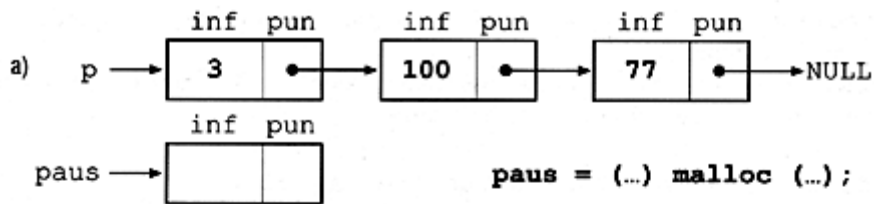
Il problema di questo paragrafo è: gestire una struttura astratta di tipo pila per mezzo delle operazioni di inserimento ed

Il puntatore alla testa della pila deve riferirsi al nuovo elemento creato (Figura 14.10c):

```
p = paus;
```

Finalmente inseriamo nel nuovo elemento il valore dell'informazione passata dall'utente (Figura 14.10d):

```
p->inf = ele;
```





```

        visualizzazione_pila(punt_testa);
        printf("\n\n Qualsiasi tasto per continuare...");
        scanf("%c%c", &pausa, &pausa);
        break;
    }
}
}

void visualizzazione_pila(struct elemento *p)
{
    struct elemento *paus = p;

    printf("\n<----- Testa della pila ");
    while(paus!=NULL) {
        printf("\n%d", paus->inf);
        paus = paus->pun;
    }
}

struct elemento *inserimento(struct elemento *p, int ele)
{
    struct elemento *paus;

    paus = (struct elemento *)malloc(sizeof(struct elemento));

    if(paus==NULL) return(NULL);

```

## 14.10 Coda

La coda (*queue*) è una struttura composta da una sequenza di elementi omogenei ordinati. L'operazione di inserimento di un elemento avviene dopo l'ultimo elemento inserito, quella di eliminazione è effettuata sul primo elemento della sequenza.

La coda è una struttura astratta, monodimensionale e dinamica. Il tipo di logica che si applica alle code è detto FIFO (*First In First Out*): il primo elemento inserito è quello che per primo viene estratto. Un esempio dell'applicazione del metodo descritto è dato da una fila di persone in attesa a uno sportello, in cui chi arriva primo “dovrebbe” essere servito per primo e chi arriva ultimo “dovrebbe” mettersi in coda. Usando una rappresentazione simile a quella adoperata per la pila, abbiamo:

coda --> D F C R --> testa

Il prossimo elemento da servire è R, che sta in testa:

coda --> D F C --> testa

Il successivo è C:

coda --> D F --> testa

Se poi arriva J, deve mettersi in coda:

coda ---> J D F ---> testa

Le implementazioni di una coda mediante un vettore e una lista sono relativamente simili a quelle viste per la pila, salvo per alcune differenze legate alla necessità di permettere, nelle code, l'accesso diretto a entrambe le estremità della sequenza.



Punto di eli-  
minazione

Punto di inse-  
rimento

Figura 14.12 Successivi inserimenti ed eliminazioni nella coda tendono a muovere la zona occupata verso l'alto

Diventa quindi molto difficile stabilire la grandezza dell'array ed è probabile avere uno spreco di memoria. Per esemplificare quanto affermato, chiamiamo `puntCoda` il puntatore all'ultimo elemento inserito e `puntTesta` il puntatore all'elemento che deve essere estratto per primo. Dopo otto inserzioni successive avremo la coda effettivamente piena (Figura 14.13); se avvengono due eliminazioni la nuova situazione sarà quella di Figura 14.14.

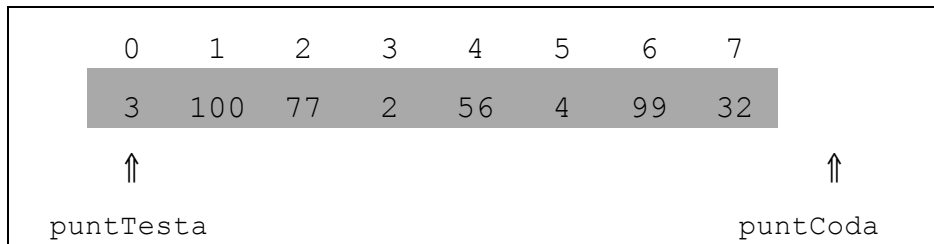
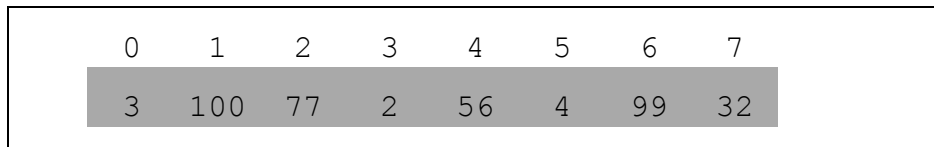


Figura 14.13 Caso di coda piena



Nel caso dell'esempio  $n=8$  e il vecchio valore di `puntCoda` è uguale a 8, per cui il nuovo valore di `puntCoda` è 0 (Figura 14.16).

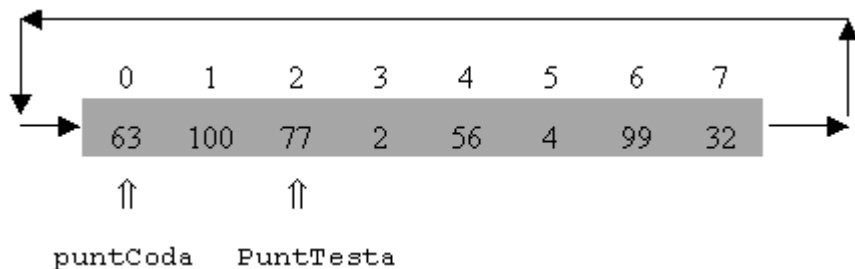


Figura 14.16 Inserimento dell'informazione (63) in testa alla coda

La condizione di coda piena equivale a `puntTesta` uguale a `puntCoda` più uno, considerando l'array in modo circolare, dove la posizione  $n$ -esima precede la prima posizione dell'array:

`if(puntTesta==((puntCoda % n)+1) coda piena...`

## 14.12 Gestione di una coda mediante liste

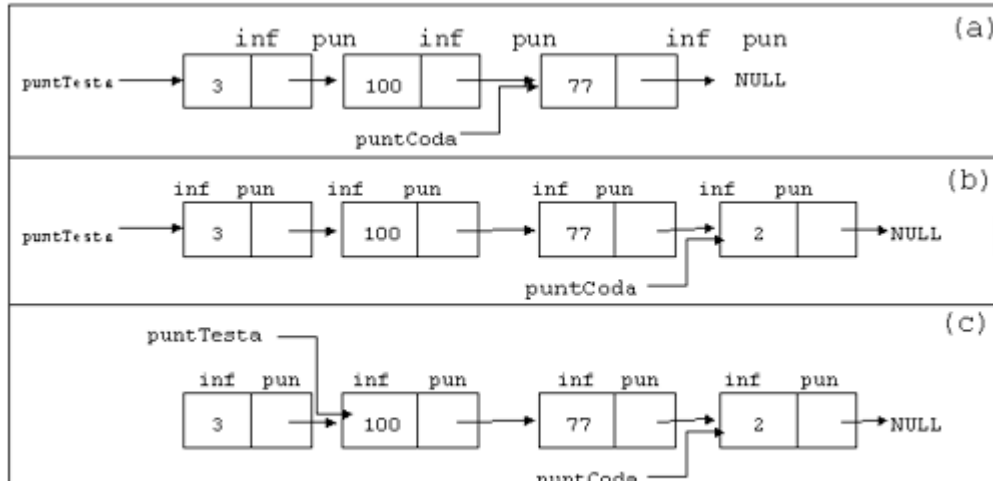
Consideriamo una lista che contiene gli elementi della coda in sequenza. Il puntatore `puntTesta` si riferisce alla posizione del primo elemento che verrà estratto (Figura 14.17).

```

else {
    /* creazione primo elemento */
    puntTesta = (struct elemento *)malloc(sizeof(struct elemento));
    puntTesta->inf = ele;          /* inserisce l'informazione */
    puntTesta->pun = NULL;        /* marca di fine lista */
}

```

Questo implica che per effettuare un inserimento si deve scandire la lista, il che è arduo se la coda contiene centinaia o migliaia di elementi. Per rendere più veloce l'operazione di inserimento si utilizza un ulteriore puntatore `punt_coda` che fa riferimento alla coda della sequenza (Figura 14.18).



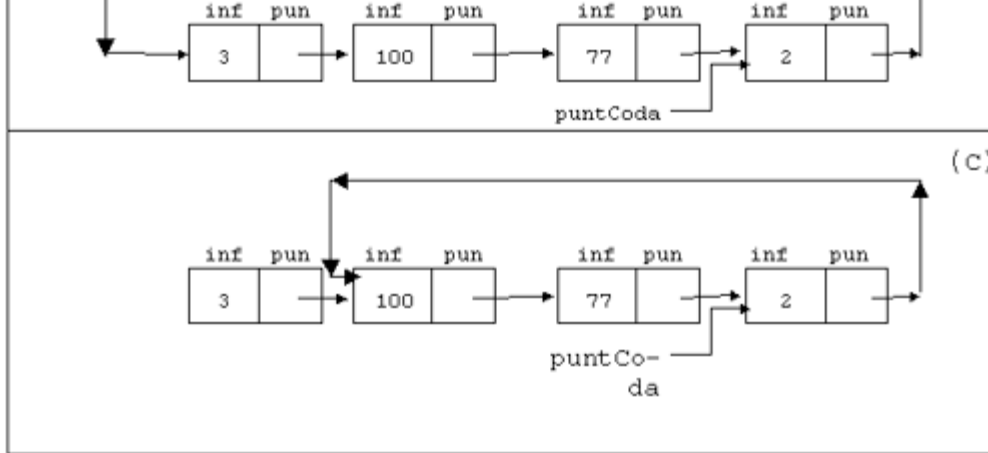


Figura 14.19 a) Implementazione di una coda mediante una lista circolare. b) Inserimento di un elemento nella coda. c) Eliminazione di un elemento dalla coda

In questo caso è necessario mantenere il solo puntatore alla coda `punta_coda`, per mezzo del quale possono essere effettuate le operazioni di inserimento (Figura 14.19b) ed eliminazione (Figura 14.19c).

```
/* INSERIMENTO creazione del nuovo elemento */
paus = (struct elemento *)malloc(sizeof(struct elemento));
paus->inf = ele;          /* inserimento dell'informazione */
paus->pun = puntaCoda->pun; /* nuovo elemento alla testa */
puntaCoda->pun = paus;    /* punta_coda al nuovo elemento */
puntaCoda = puntaCoda->pun; /* attualizzazione puntaCoda */

/* ELIMINAZIONE */
paus = puntaCoda->pun;    /* salvataggio puntatore alla testa */
```

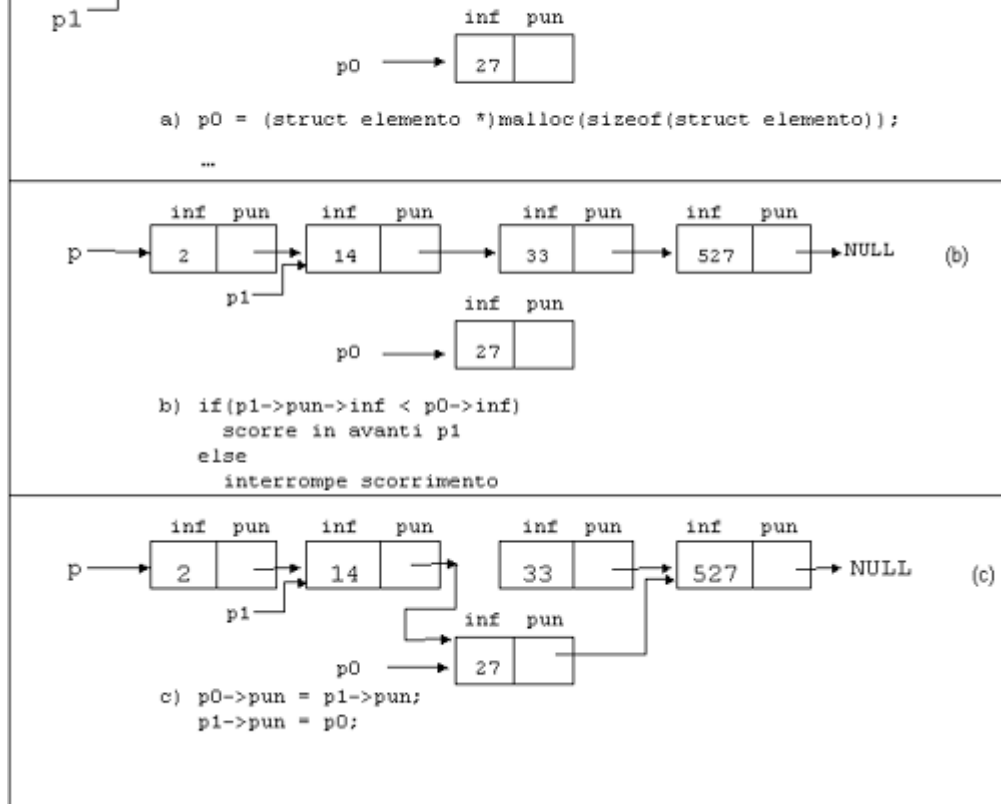


Figura 14.20 Inserimento di un elemento in una lista ordinata; nel caso preso in esame l'inserimento avviene in una posizione intermedia della lista

```

b) if(p1->pun->inf != e.inf)
    scorre in avanti p1
    else {
        ...
        p2 = p1->pun;
        p1->pun = p1->pun->pun;
    }

```

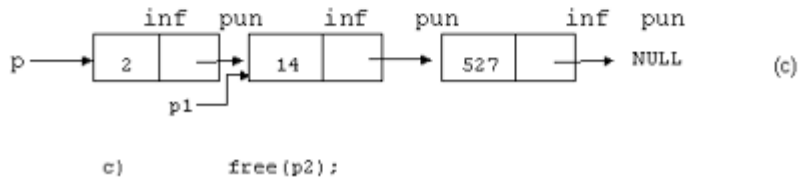


Figura 14.21 Eliminazione di un elemento da una lista ordinata; nel caso in esame l'eliminazione avviene in una posizione intermedia della lista

```

/* GESTIONE DI LISTA ORDINATA
   Operazioni di inserimento, eliminazione e visualizzazione
   Utilizza una lista lineare per implementare la pila */

#include <stdio.h>
#include<malloc.h>

struct elemento {
    int inf;

```

```

switch(scelta) {
    case 1:
        punt_lista = inserimento(punt_lista);
        break;
    case 2:
        punt_lista = eliminazione(punt_lista);
        break;
    case 3:
        visualizzazione(punt_lista);
        printf("\n\nQualsiasi tasto per continuare...");
        scanf("%c%c", &pausa, &pausa);
        break;
}
}
}

/* Visualizzazione della lista */
void visualizzazione(struct elemento *p)
{
    struct elemento *paus = p;

    printf("\npunt_lista---> ");
    while(paus!=NULL) {
        printf("%d---> ", paus->inf);
        paus = paus->pun;
    }
}

```

```

        if(p1->pun == p0->inf) {
            p1 = p1->pun;          /* Scorre in avanti p1 */
        }
        else
            posizione = 1;          /* Interrompe lo scorrimento */
    }
    p0->pun = p1->pun; /* Connessione all'elemento successivo */
    p1->pun = p0;      /* Connessione dall'elemento precedente */
}
}
return(p); /* Ritorno del puntatore all'inizio della lista */
}

```

/\* Eliminazione dell'elemento richiesto dalla lista ordinata \*/

```

struct elemento *eliminazione(struct elemento *p)
{

```

```

    struct elemento *p1 = p, *p2;

```

```

    struct elemento e;

```

```

    int posizione = 0;

```

```

    char pausa;

```

```

    printf("\nInserisci l'informazione da eliminare: ");

```

```

    scanf("%d", &e.inf);

```

```

    if(p1!=NULL) {

```

```

        if(p1->inf == e.inf) { /* Se è il primo da eliminare */

```

```

            p2 = p1;

```

```

            p = p->pun; /* si modifica il puntatore alla testa */

```

```

            free(p2);

```



1. Scrivere un programma che accetti in ingresso una sequenza di valori interi terminante con zero e la memorizzi in una lista lineare. Successivamente il programma deve determinare il numero di pari e di dispari presenti nella lista. Risolvere il problema anche con funzioni ricorsive.

\* 2. Scrivere un programma che accetti in ingresso una sequenza di valori interi terminante con zero e la memorizzi in una lista lineare. Successivamente il programma deve eliminare dalla lista i numeri pari.

3. Scrivere un programma che accetti in ingresso una sequenza di valori interi terminante con zero e la memorizzi in una lista lineare. Successivamente eliminare dalla lista creata quelli che non sono divisori di  $n$ , dove  $n$  è un numero intero passato in ingresso dall'utente.

\* 4. Data in ingresso una sequenza di valori interi terminante con zero, costruire due liste lineari, una contenente i valori positivi e una i valori negativi. Visualizzare le liste costruite.

5. Definire una funzione che calcoli il numero totale degli elementi che compongono una lista. Scrivere un'altra funzione che stampi l' $n$ -esimo elemento di una lista se questo esiste, altrimenti visualizzi il messaggio: "La lista non contiene un numero sufficiente di elementi".

6. Scrivere una funzione che cancelli da una lista tutte le occorrenze di un particolare elemento se questo è presente nella lista, altrimenti visualizzi il messaggio: "Elemento non presente nella lista".

7. Realizzare una funzione che ordini una lista in modo crescente. Scrivere un'altra funzione che inserisca in una lista ordinata al posto opportuno un nuovo elemento richiesto all'utente.

8. Scrivere una funzione che visualizzi in ordine inverso una lista.

9. Scrivere una funzione che inverta l'ordine di una lista.

10. Scrivere una funzione che, a partire da due liste, ne costruisca una terza ottenuta alternando gli elementi delle altre due.

evidente.

Un esempio di albero binario è presentato in Figura 15.1. Il nodo designato come radice ha etichetta 104, da esso si diparte il sottoalbero sinistro con radice 32 e il sottoalbero destro con radice 121. Si dice che i nodi con etichette 32 e 121 sono *fratelli* e sono rispettivamente il figlio sinistro e il figlio destro del nodo con etichetta 104. L'albero che ha come radice 32 ha ancora due sottoalberi con radici 23 e 44, i quali non hanno figli o, in altre parole, hanno come figli alberi vuoti. L'albero con radice 121 non ha figlio sinistro e ha come figlio destro il sottoalbero con radice 200, il quale a sua volta ha come figlio sinistro il sottoalbero con radice 152 e non ha figlio destro. Il sottoalbero che ha come radice 152 non ha figli.

I nodi da cui non si dipartono altri sottoalberi (non vuoti) sono detti *nodi terminali* o *foglie*. Nell'esempio sono quelli etichettati con: 23, 44 e 152.

Si chiamano *visite* le scansioni dell'albero che portano a percorrerne i vari nodi. L'ordine in cui questo avviene distingue differenti tipi di visite.

La visita *in ordine anticipato* di un albero binario viene effettuata con il seguente algoritmo:

```
anticipato( radice dell'albero )  
  Se l'albero non è vuoto:  
    Visita la radice  
    anticipato( radice del sottoalbero sinistro )  
    anticipato( radice del sottoalbero destro )
```

in cui abbiamo evidenziato la ricorsività della visita. Nel caso dell'albero di Figura 15.1 la visita in ordine anticipato dà la sequenza: 104, 32, 23, 44, 121, 200, 152.

La visita *in ordine differito* invece è così descritta:

```
differito( radice dell'albero )  
  Se l'albero non è vuoto:  
    differito( radice del sottoalbero sinistro )  
    differito( radice del sottoalbero destro )  
    Visita la radice
```

La visita in ordine differito dell'albero di Figura 15.1 dà la sequenza: 23, 44, 32, 152, 200, 121, 104.

## 15.2 Implementazione di alberi binari

Consideriamo ora il seguente problema: memorizzare i dati interi immessi dall'utente in un albero binario tale che il valore dell'etichetta di un qualsiasi nodo sia maggiore di tutti i valori presenti nel suo sottoalbero sinistro e minore di tutti i valori del suo sottoalbero destro. Per esempio, se l'utente immette in sequenza 104, 32, 44, 121, 200 152, 23, un albero conforme con la definizione è quello di Figura 15.1. La sequenza di immissione termina quando l'utente inserisce il valore zero; in caso di immissione di più occorrenze dello stesso valore, soltanto la prima verrà inserita nell'albero. Si chiede di visitare l'albero in ordine anticipato.

Una soluzione è ottenuta definendo ciascun nodo come una struttura costituita da un campo informazione, contenente l'etichetta, e due puntatori al sottoalbero sinistro e al sottoalbero destro:

```
struct nodo {  
    int inf;  
    struct nodo *alb_sin;  
    struct nodo *alb_des;  
};
```

L'albero binario di Figura 15.1 verrebbe così memorizzato come in Figura 15.2. I puntatori che non fanno riferimento a nessun nodo devono essere messi a valore NULL, in modo da permettere una corretta terminazione delle visite. Il programma completo è presentato nel Listato 15.1.

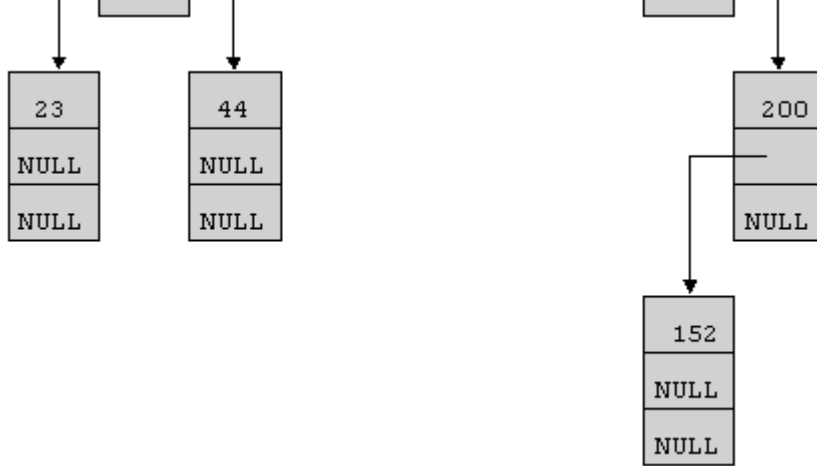


Figura 15.2 Esempio di memorizzazione dell'albero binario di Figura 15.1 in una lista doppia

Nel `main` è dichiarata la variabile `radice` che conterrà il riferimento alla radice dell'albero; essa è un puntatore a oggetti di tipo `nodo`:

```
struct nodo *radice;
```

Il `main` invoca la funzione `alb_bin` che crea l'albero e ritorna il puntatore alla radice:

```
radice = alb_bin();
```

Successivamente chiama la funzione di visita, che visualizza le etichette in ordine anticipato:

```
anticipato(radice);
```

Se al contrario `val` è minore di `inf`, allora viene richiamata ricorsivamente la funzione sul sottoalbero sinistro:

```
p->alb_sin = crea_nodo( p->alb_sin, val);
```

Se nessuno dei due casi risulta vero, significa che `val` è uguale a `inf` e dunque non deve essere inserito nell'albero perché, come specificava il testo del problema, le occorrenze multiple devono essere scartate.

Si noti come, nel caso di un valore non ancora memorizzato nell'albero, il procedimento ricorsivo termina sempre con la creazione di una foglia, corrispondente alle istruzioni di “creazione del nodo” che abbiamo elencato in precedenza. La connessione del nodo creato al padre avviene grazie al valore di ritorno delle chiamate ricorsive, che è assegnato a `p->alb_des` o a `p->alb_sin` secondo il caso.

La funzione `anticipato` corrisponde in maniera speculare alla definizione di visita in ordine anticipato data precedentemente. L'intestazione della definizione della funzione è la seguente:

```
void anticipato(struct nodo *p);
```

Il parametro attuale `p` assume il valore della radice, se non è uguale a `NULL` (cioè se l'albero non è vuoto), ne viene stampata l'etichetta, e viene invocata ricorsivamente `anticipato` passandole la radice del sottoalbero sinistro:

```
anticipato(p->alb_sin);
```

Successivamente viene richiamata la funzione passandole la radice del sottoalbero destro:

```
anticipato(p->alb_des);
```

## 15.3 Visita in ordine simmetrico

Il problema di questo paragrafo è: ampliare il programma del paragrafo precedente in modo che venga effettuata anche la visita in ordine simmetrico e inoltre sia ricercato nell'albero un valore richiesto all'utente. Nel Listato 15.2 sono riportate le modifiche da apportare al programma e le nuove funzioni.

La funzione di visita `simmetrico` è analoga ad `anticipato` esaminata precedentemente. La differenza sta nel fatto

```

else
    printf("\n Elemento non presente\n");

/* Funzione che visita l'albero binario in ordine simmetrico.
   Da aggiungere al Listato 15.1 */

void simmetrico(struct nodo *p)
{
    if(p!=NULL) {
        simmetrico(p->alb_sin);
        printf("%d  ", p->inf);
        simmetrico(p->alb_des);
    }
}

/* Funzione che ricerca un'etichetta nell'albero binario.
   Da aggiungere al Listato 15.1.
   Visita l'albero in ordine anticipato */

void ricerca(struct nodo *p, int val, struct nodo **p_ele)
{
    if(p!=NULL)
        if(val == p->inf)          /* La ricerca ha dato esito positivo */
            *p_ele = p;           /* p_ele è passato per indirizzo
                                   per cui l'assegnamento di p
                                   avviene sul parametro attuale */
        else {

```

```

    }
    else
    {
        if(val < p->inf) {
            printf("  sinistra");
            ric_bin(p->alb_sin, val, p_ele);
        }
        else {
            printf("  destra");
            ric_bin(p->alb_des, val, p_ele);
        }
    }
}

```

Listato 15.3 Ricerca di un valore nell'albero binario

### ✓ NOTA

Osserviamo che la ricerca ha le stesse prestazioni di quella binaria solamente se l'albero è bilanciato, cioè se, preso un qualsiasi nodo, il numero di nodi dei suoi due sottoalberi differisce al più di una unità. Il numero di livelli dell'albero, infatti, costituisce un limite superiore al numero di accessi effettuati dalla ricerca.

La struttura dell'albero creato con la funzione `crea_nodo` dipende dalla sequenza con cui vengono immessi i valori in ingresso; il caso peggiore si presenta quando questi sono già ordinati. Per esempio, se l'utente inserisse: 23, 32, 44, 104, 121, 152, 200 l'albero si presenterebbe totalmente sbilanciato, un albero cioè dove il numero di livelli è uguale al numero di elementi inseriti. Altre sequenze danno vita ad alberi più bilanciati.

Si potrebbero scrivere algoritmi che bilanciano l'albero mentre lo creano, ma spesso nella realtà – specialmente se si sta lavorando con la memoria secondaria – questo risulta pesante in termini di tempi di risposta. Un'altra soluzione è quella di prevedere funzioni che trasformino alberi qualsiasi in alberi bilanciati, per cui l'inserimento di elementi avviene così come

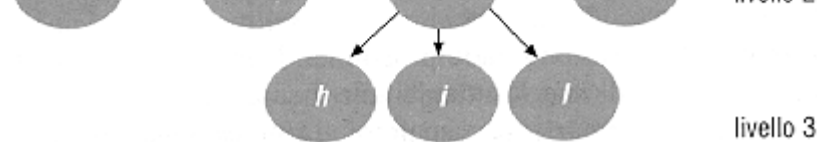


Figura 15.3 Rappresentazione di un albero

Nel caso si definisca l'ordinamento da sinistra verso destra, al livello 2 dell'albero della figura avremmo in sequenza: *c*, *d*, *g*, *m*. Analogamente al caso dell'albero binario, possiamo definire le visite per l'albero ordinato.

La visita in ordine anticipato è descritta da:

```
anticipato( radice dell'albero )
    Visita la radice
    fintantoché ci sono sottoalberi
        anticipato( radice del prossimo sottoalbero )
```

Nel caso dell'albero di Figura 15.3 avremmo: *a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *l*, *m*.

La visita in ordine differito è descritta invece da:

```
differito( radice dell'albero )
    fintantoché ci sono sottoalberi
        differito( radice del prossimo sottoalbero )
    Visita la radice
```

Sempre con riferimento all'esempio si avrebbe: *c*, *d*, *b*, *e*, *h*, *i*, *l*, *g*, *m*, *f*, *a*.

Per rappresentare un albero in forma compatta si può utilizzare la rappresentazione parentetica in ordine anticipato, che ha la seguente sintassi:

```
(radice(I sottoalbero)(II sottoalbero) ... (N-esimo sottoalbero))
```



della lista contiene l'etichetta del nodo, gli elementi successivi sono in numero uguale ai nodi figli. In ciascuno di essi viene memorizzato il puntatore alle liste di tali nodi.

```

struct nodo {
    char inf;
    struct nodo *figlio;
    struct nodo *p_arco;
};

```

inf	figlio	p_arco

nodo

Il campo `inf` viene utilizzato soltanto per il primo degli elementi di ogni lista, in cui il campo `p_arco` punta al secondo elemento della lista e `figlio` non viene utilizzato. Per gli altri elementi della lista `figlio` viene utilizzato per far riferimento alla lista del figlio e `p_arco` per puntare al successivo elemento della lista. In Figura 15.4 viene presentata la memorizzazione dell'albero di Figura 15.3.

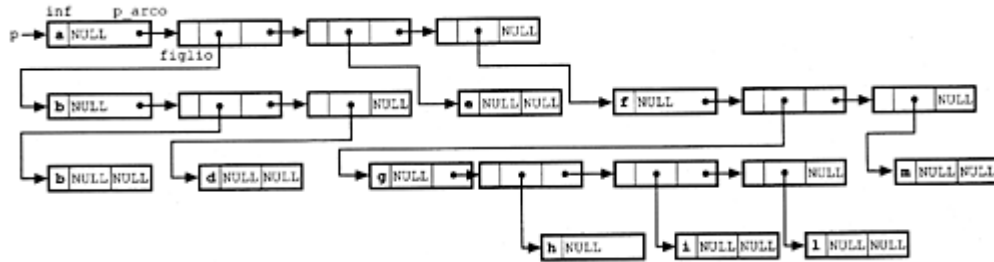


Figura 15.4 Memorizzazione dell'albero di Figura 15.3

L'albero è implementato con liste multiple

\*/

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

struct nodo {
    char inf;
    struct nodo *figlio;
    struct nodo *p_arco;
};

struct nodo *albero();
struct nodo *crea_albero(struct nodo *);
void anticipato(struct nodo *);

main()
{
    struct nodo *radice;
    radice = albero();                /* Creazione dell'albero */
    printf("\nVISITA IN ORDINE ANTICIPATO\n");
    anticipato(radice);
}

/* Legge il primo carattere della rappresentazione parentetica
   e invoca la funzione crea_albero() */
struct nodo *albero()
{
    struct nodo *p = NULL;
```

```

    }

    paus->p_arco = NULL;
    return( p );    /* Ritorna il puntatore alla radice al chiamante */
}

/* Visita ricorsivamente l'albero in ordine anticipato */
void anticipato(struct nodo *p)
{
    printf("(%c", p->inf);
    p = p->p_arco;

    while(p!=NULL) {
        anticipato(p->figlio);
        p = p->p_arco;
    }
    printf(")");
}

```

Listato 15.4 Creazione di un albero a partire dalla sua rappresentazione parentetica

## 15.7 Ricerca di un sottoalbero

Supponiamo di dover affrontare un ulteriore problema: dato l'albero allocato col procedimento del paragrafo precedente, ricercare il valore di una etichetta  $e$ , se presente, visitare il sottoalbero che da essa si diparte.

Considerando l'albero di Figura 15.3, se l'utente immette  $f$ , il programma deve visualizzare  $f, g, h, i, l, m$ . L'algoritmo di ricerca del Listato 15.5 si comporta come la visita in ordine anticipato, con l'eccezione che se il valore viene

```

if(sa == p->ini)
    *punt_sa = p;
else
{
    p = p->p_arco;
    while(p!=NULL) {
        ricerca(p->figlio, sa, punt_sa);
        p = p->p_arco;
    }
}
}
}

```

Listato 15.5 Ricerca di un nodo e visualizzazione del sottoalbero che si diparte da esso

## 15.8 Trasformazione di alberi

Da un albero ordinato  $A$  di  $n$  nodi è possibile ricavare un equivalente albero binario  $B$  di  $n$  nodi con la seguente regola:

- la radice di  $A$  coincide con la radice di  $B$ ;
- ogni nodo  $b$  di  $B$  ha come radice del sottoalbero sinistro il primo figlio di  $b$  in  $A$  e come sottoalbero destro il fratello successivo di  $b$  in  $A$ .

In Figura 15.5 vediamo la trasformazione dell'albero di Figura 15.3 in albero binario.

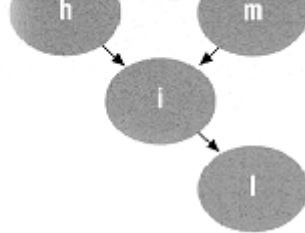


Figura 15.5 Albero binario corrispondente all'albero di Figura 15.3

Dalla regola di trasformazione si deduce che in un albero binario, ricavato da un albero ordinato, la radice può avere solamente il figlio sinistro.

Il programma del Listato 15.6 accetta in ingresso la rappresentazione parentetica di un albero ordinato e memorizza il corrispondente albero binario nella forma che abbiamo visto nei primi paragrafi di questo capitolo. Le funzioni utilizzate sono di tipo non ricorsivo e pertanto esiste il problema di memorizzare i puntatori degli elementi di livello superiore. A questo scopo si utilizza una pila di puntatori, dove si va a inserire il puntatore di un certo livello prima di scendere al livello sottostante e da dove si preleva il puntatore una volta che si debba risalire di livello.

La pila è implementata mediante una lista. Rimangono dunque valide le funzioni di inserimento, eliminazione e `pila_vuota` già trattate. L'unica differenza sta nella struttura di ogni elemento: la parte informazione è di tipo puntatore a un nodo dell'albero:

```
struct elemento {                /* Elemento della lista lineare */
    struct nodo    *inf;          /* con cui è implementata la pila */
    struct elemento *pun;
};
```

```

    struct nodo *ini, *fin; // con cui è implementata la pila */
    struct elemento *pun;
};

/* Funzioni per la gestione della pila */
struct elemento *inserimento(struct elemento *, struct nodo *);
struct elemento *eliminazione( struct elemento *, struct nodo **);
int pila_vuota(struct elemento *);

main()
{
    struct nodo *radice;

    radice = alb_bin_par();

    printf("\nVISITA IN ORDINE SIMMETRICO\n");
    simmetrico( radice );
    printf("\nVISITA IN ORDINE ANTICIPATO\n");
    anticipato( radice );
    printf("\nVISITA IN ORDINE DIFFERITO\n");
    differito( radice );
}

/* Dalla rappresentazione parentetica di un albero crea
   il corrispondente albero binario */
struct nodo *alb_bin_par()
{
    struct nodo *p;

```

```

    }
    else {
        paus->alb_des = pp;
        logica = 1;
    }
    paus = pp;
}
else
    if(logica)
        logica = 0;
    else
        /* Eliminazione dalla pila */
        punt_testa = eliminazione(punt_testa, &paus);
}
while(!pila_vuota(punt_testa));

return(p);
}

```

/\* Funzioni per la gestione della pila \*/

```

struct elemento *inserimento(struct elemento *p, struct nodo *ele)
{
    struct elemento *paus;

    paus = (struct elemento *)malloc(sizeof(struct elemento));
    if(paus==NULL) return(NULL);
}

```

```
/* Devono essere incluse le funzioni ricorsive di visita dell'albero binario già esaminate, dove il valore visualizzato è  
di tipo char */  
...
```

Listato 15.6 Creazione di un albero binario corrispondente all'albero ordinato (non binario) immesso in forma parentetica dall'utente

## 15.9 Grafi

Un *grafo* è costituito da un insieme di nodi e un insieme di archi che uniscono coppie di nodi tali che non c'è mai più di un arco che unisce una coppia di nodi. Il grafo è *orientato* quando viene attribuito un senso di percorrenza agli archi stessi: in tal caso è ammesso che tra due nodi vi siano due archi purché orientati in sensi opposti; è anche possibile che un arco ricada sullo stesso nodo (Figura 15.6). È evidente come gli alberi siano un caso particolare di grafi, in quanto rispondono a un insieme di regole più restrittivo.

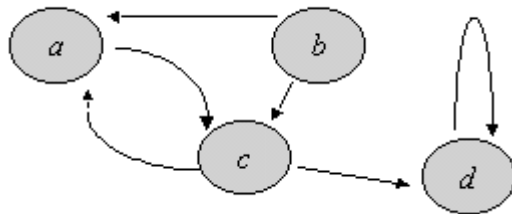


Figura 15.6 Rappresentazione di un grafo



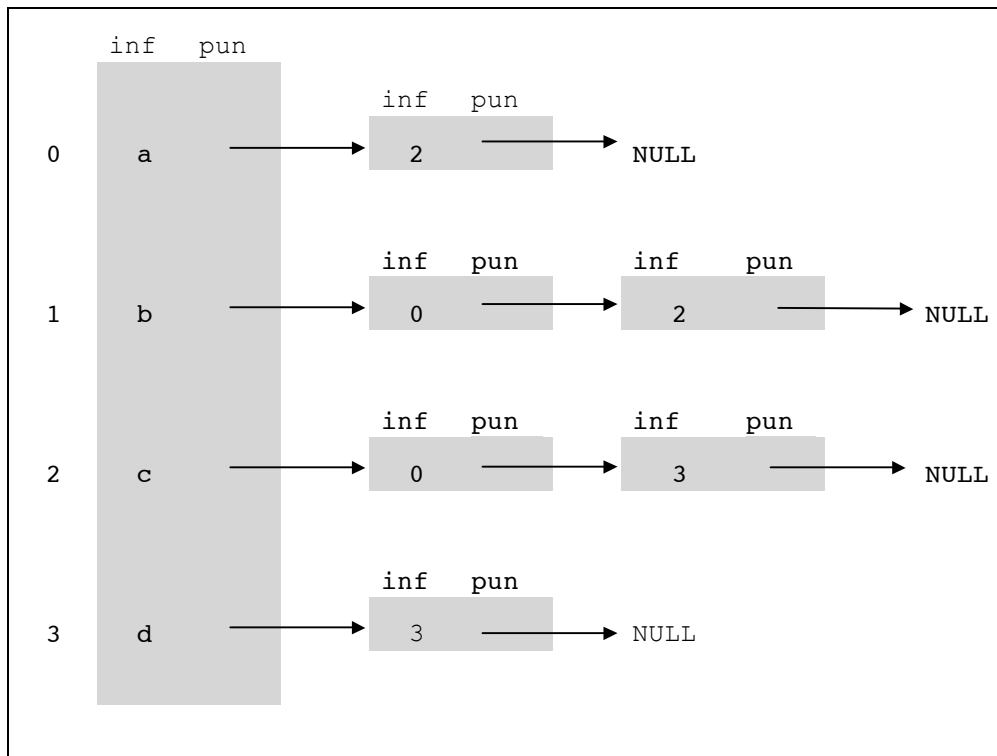


Figura 15.8 Grafo memorizzato mediante una lista di successori

memoria centrale al momento della loro utilizzazione.

I grafi sono spesso usati nella soluzione di problemi di ricerca operativa che implicano, per esempio, lo studio di cammini minimi.

```
/* Trasformazione della rappresentazione di un grafo
   da una matrice di adiacenze a una lista di successori */

#include <stdio.h>
#include <malloc.h>

struct nodo {                                /* Struttura di un nodo */
    char inf;
    struct successore *pun;
};

struct successore {                          /* Elemento della lista di successori */
    int inf;
    struct successore *pun;
};

int a[10][10];                               /* Matrice di adiacenze */
struct nodo s[10];                          /* Array di nodi */
int n;                                       /* Numero di nodi */

void mat_adiacenze(void);
void vis_mat_adiacenze(void);
void successori(void);
```

```

for(i=0; i<n; i++) {
    for(j=0; j<n; j++) {
        printf("\nArco orientato da [%c] a [%c] (0 no, 1 si) ? ",
            s[i].inf, s[j].inf);
        scanf("%d", &a[i][j]);
    }
}

/* Visualizza la matrice di adiacenze */

void vis_mat_adiacenze(void)
{
    int i, j;

    printf("\nMATRICE DI ADIACENZE\n");
    for(i=0; i<n; i++) /* Visualizza i nodi (colonne) */
        printf("    %c", s[i].inf);

    for(i=0; i<n; i++) {
        printf("\n%c ", s[i].inf); /* Visualizza i nodi (righe) */
        for(j=0; j<n; j++)
            printf("%d   ", a[i][j]); /* Visualizza gli archi */
    }
}

/* Crea le liste di successori. Per ogni arco rappresentato
   nella matrice di adiacenze chiama crea_succ() */

```

```

    p = p->pun;
    p->inf = j;
    p->pun = NULL;
}
}

/* Per ogni nodo del grafo restituisce i suoi successori.
   Lavora sulle liste di successori */

void visita(void)
{
    int i;
    struct successore *p;

    printf("\n");

    for(i=0; i<n; i++) {
        printf("\n[%c] ha come successori: ", s[i].inf);
        p = s[i].pun;
        while(p!=NULL) {
            printf(" %c", s[p->inf].inf);
            p = p->pun;
        }
    }
}

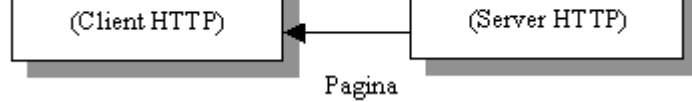
```

```

        struct nodo *figlio;
        struct nodo *p_arco;
    };

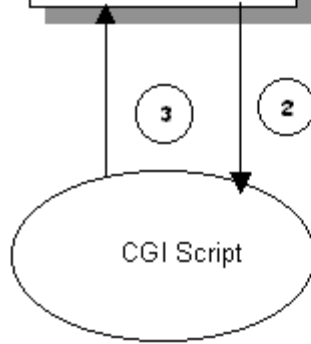
```

5. Scrivere una funzione che calcoli il numero di livelli di un albero binario memorizzato in una lista doppia.
6. Scrivere un programma che da un albero binario, memorizzato in una lista doppia, elimini (rilasciando opportunamente la memoria) il sottoalbero la cui etichetta della radice viene passata in ingresso dall'utente.
- \* 7. Ampliare il programma di implementazione di un grafo mediante una lista di successori esaminato nel capitolo, in modo che accetti in ingresso il valore di un'etichetta e visualizzi le etichette di tutti i nodi da esso raggiungibili. Per la scansione dei nodi connessi si utilizzi una funzione ricorsiva.  
[Prestare attenzione al fatto che la scansione può portare a un ciclo infinito se nel percorso si passa per più di una volta sullo stesso nodo.]
8. Scrivere una funzione che un dato grafo, memorizzato in una matrice di adiacenze, e date in ingresso le etichette di due nodi, verifichi se esiste un arco che li collega, nel qual caso lo cancelli.
9. Verificare se, dato un grafo memorizzato in una matrice di adiacenze, e date in ingresso le etichette di due nodi, dal primo nodo è possibile arrivare al secondo attraverso un cammino di archi orientati.
10. Risolvere i due esercizi precedenti ma con il grafo memorizzato in una lista di successori.
11. Disegnare l'albero binario che si ottiene fornendo al programma del Listato 15.1 i seguenti dati:  
35, 12, 91, 7, 13, 108, 64, 66, 19, 12, 8, 0.
12. Scrivere una funzione che stabilisca se due alberi binari sono uguali.
13. Dato l'albero binario costruito con il Listato 15.1 realizzare due funzioni, la prima che calcoli la somma totale delle informazioni contenute nei nodi, l'altra che determini il maggiore e il minore.
14. Scrivere una funzione che, dato il numero di livelli dell'albero binario, calcoli la somma totale delle informazioni contenute nei nodi.



In pratica quando sul vostro browser – Netscape Navigator, Internet Explorer, Opera, Tango o altro – digitate un indirizzo Web, per esempio `www.mcgraw-hill.com`, inviate attraverso la rete Internet una richiesta al Web Server della McGraw-Hill, il quale risponderà inviandovi la prima pagina, detta *home page*, di quel sito. Volendo usare un linguaggio tecnicamente più preciso, il browser, che svolge la funzione di un client HTTP, invia una *HTTP request* al Web Server, che svolge il ruolo di HTTP Server, il quale a sua volta risponde con una *HTTP response*.

Il protocollo HTTP stabilisce anche il formato che devono avere sia la request sia la response. Per esempio una response, che corrisponde a una pagina scritta in linguaggio HTML, sarà costituita da una intestazione convenzionale, giusto per dire che quella è una response, cui segue il corpo vero e proprio della risposta in forma di pagina HTML. Vedremo più avanti i dettagli di questa operazione. Comunque, nella sua vera essenza, il protocollo HTTP e con esso tutta la “magia” della navigazione Internet consiste in questo: una richiesta seguita da una risposta e niente più. È una specie di protocollo di tipo “botta e risposta” che – più propriamente – è detto protocollo *stateless* o privo di memoria.



`/cgi-bin/hello.exe`

Descriviamo ora passo per passo il meccanismo dei CGI prendendo a esempio un semplice programma – `hello.exe` – che costruisce una pagina dinamica HTML che presenta sul browser del richiedente il classico messaggio:

Hello, World!

**1. Invio della request** – Il client HTTP, ovvero un browser, effettua una request a un server HTTP identificato dal seguente indirizzo, detto URL:

`http://www.cyberscuola.org/cgi-bin/hello.exe?`

In questo indirizzoviene identificato il server HTTP

Vediamo ora nel concreto come si innesca un CGI da HTML e come si scrive un CGI in C, a partire proprio dal semplice esempio `hello.exe`.

## 16.4 Un semplice CGI: `hello.exe`

Prima di descrivere e commentare il CGI `hello.exe` vediamo a livello di utente quali sono le azioni che si debbono intraprendere. Per prima cosa invocheremo da browser il CGI `hello.exe` (Figura 16.3): in risposta otterremo la pagina di saluto di Figura 16.4.



Figura 16.3 Invocazione del CGI `hello.exe`



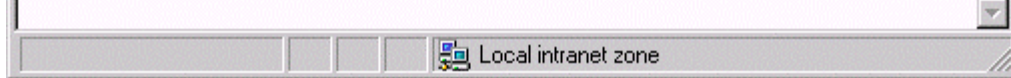


Figura 16.4 Pagina di saluto ottenuta con l'esecuzione del CGI `hello.exe`

Visti gli effetti, vediamo ora quali sono state le “cause” di detti effetti nel Listato 16.1 e commentiamo questo semplice programma CGI.

```
/* hello.c: un esempio di CGI */

/* Includere ciò che serve per lo Standard Input e Output */
#include <stdio.h>

/* Dichiarare il main, come sempre */
int main(int argc, char *argv[])
{
    /* Per prima cosa indicare un'informazione necessaria
    per l'intestazione della response HTTP */
    printf("Content-type: text/html\n\n");
    /* Inviare su standard Output i tag HTML */
    printf("<head>\n");
    printf("<title>Hello, World</title>\n");
    printf("</head>\n");
    printf("<body>\n");
```

```
        /  si notino le due righe vuote dopo  
        la scritta Content-type: text/html */
```

Come si può osservare, il programma invia su STDOUT, che – ricordiamo ancora una volta – viene intercettato dal server HTTP e poi inoltrato verso il browser, un messaggio che se visualizzato su monitor darebbe luogo alla scritta:

```
Content-type: text/html
```

Il significato di questo messaggio potrebbe essere così interpretato: “Il contenuto (*content-type*) di quello che sto inviando è un testo ASCII corrispondente alla specifica di una pagina HTML (*text/html*)”. Si noti la presenza delle due righe vuote `\n\n`, assolutamente necessarie e, purtroppo, spesso dimenticate.

Infine viene trasmessa, rigo per rigo, la pagina HTML dinamica che realizza il citato messaggio, dapprima inviando su standard output i tag di inizio pagina (in Appendice A riportiamo una breve introduzione al linguaggio HTML che chiarisce il significato dei tag utilizzati nel testo):

```
printf("<head>\n");  
printf("<title>Hello, World</title>\n");  
printf("</head>\n");
```

poi procedendo con la specifica del corpo della pagina:

```
printf("<body>\n");  
printf("<h1>Hello, World</h1>\n");  
printf("</body>\n");
```

```
/* Ora esatta (o quasi) */
#include <stdio.h>
#include <time.h>

int main(int argc, char *argv[])
{
/* Si dichiarano due variabili per memorizzare l'ora */
time_t bintime;
struct tm *curtime;

    printf("Content-type: text/html\n\n");

    printf("<head>\n");
    printf("<title>Ora (quasi) Esatta</title>\n");
    printf("</head>\n");
    printf("<body>\n");
    printf("<h1>\n");
    time(&bintime);
    curtime = localtime(&bintime);
    printf("Ora (quasi) esatta: %s\n", asctime(curtime));
    printf("</h1>\n");
    printf("</body>\n");
```

```

int tm_min;      /* minutes after the hour - [0,59] */
int tm_hour;     /* hours since midnight - [0,23] */
int tm_mday;     /* day of the month - [1,31] */
int tm_mon;      /* months since January - [0,11] */
int tm_year;     /* years since 1900 */
int tm_wday;     /* days since Sunday - [0,6] */
int tm_yday;     /* days since January 1 - [0,365] */
int tm_isdst;    /* daylight savings time flag */
};

```

Dopo aver costruito l'intestazione della response:

```
printf("Content-type: text/html\n\n");
```

provvediamo a costruire l'intestazione della pagina:

```

printf("<head>\n");
printf("<title>Ora (quasi) Esatta</title>\n");
printf("</head>\n");

```

Il corpo della pagina è stavolta leggermente più articolato rispetto al semplice `hello.exe`:

```

printf("<body>\n");
printf("<h1>\n");
time(&bintime);
curtime = localtime(&bintime);

```



Figura 16.5 Pagina con l'ora esatta ottenuta con il CGI del Listato 16.2

## 16.6 Il passaggio di parametri

Ora che abbiamo acquisito una certa confidenza con la programmazione CGI affrontiamo un altro tema di interesse: come si effettua il passaggio di parametri a un CGI. Per prima cosa occorre dire che il modo canonico di invocare un CGI passando dei parametri è quello di impiegare un FORM HTML. Il FORM è un tag del linguaggio HTML che consente di costruire moduli per l'inserimento di informazioni. Vediamo subito un esempio in Figura 16.6. I campi vuoti di questo FORM possono essere riempiti come in Figura 16.7.

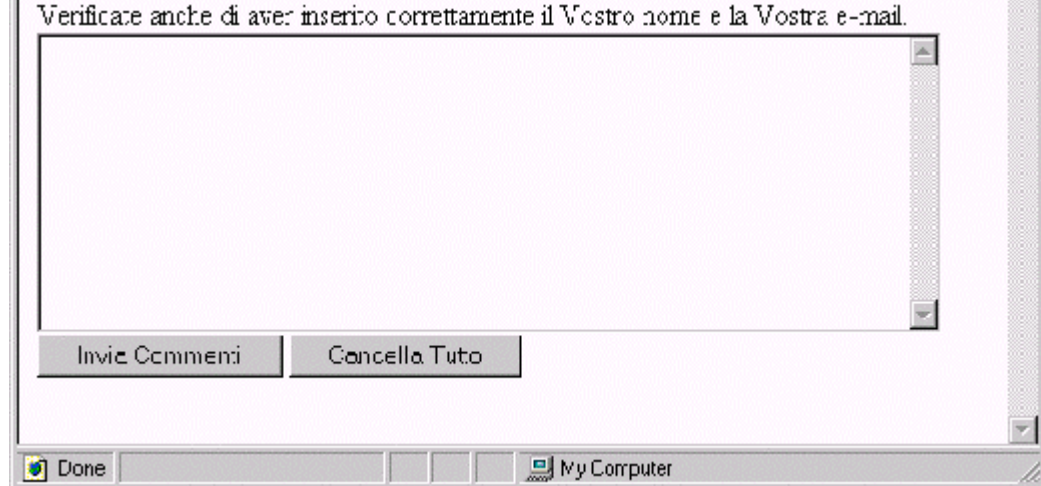


Figura 16.6 Esempio di FORM

e-mail:

Inserite di seguito i Vostri commenti.

Verificate anche di aver inserito correttamente il Vostro nome e la Vostra e-mail.

Mi è piaciuto tutto.

Complimenti e grazie,

Andrea Bellini|

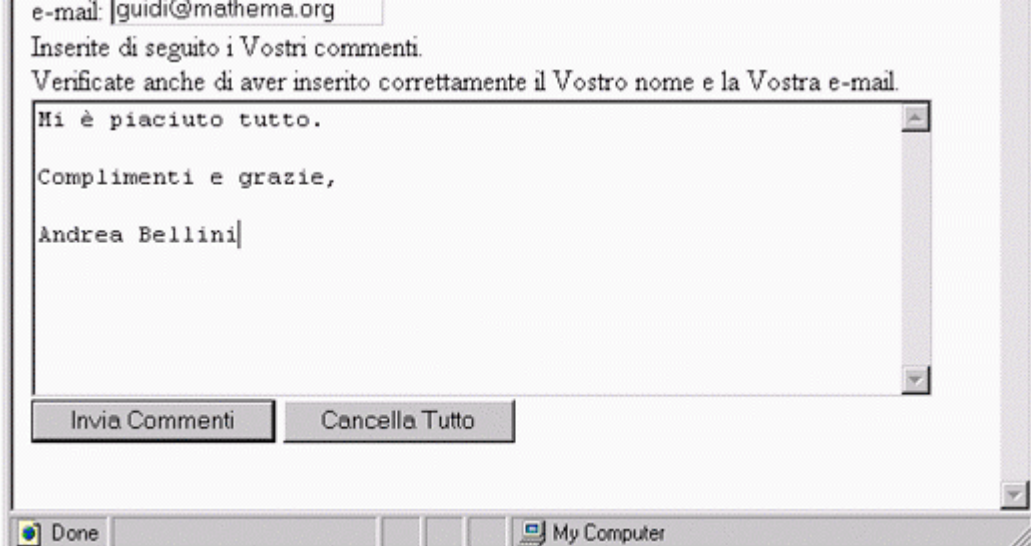


Figura 16.7 Esempio di FORM con i campi riempiti

Premendo il pulsante “Invia Commenti” si attiva il CGI che acquisisce i dati immessi, effettua delle elaborazioni e ritorna una pagina HTML per confermare l’avvenuta elaborazione. Il codice HTML che ha permesso di generare questa pagina è quello del Listato 16.3.

```
<html>
<head>
<title>Sono Graditi i Vostri Commenti</title>
</head>
```

```
<textarea name="comments" rows="10" cols="60">
</textarea>
<br>
<input type="submit" value="Invia Commenti">
<input type="reset" value="Cancella Tutto">
</form>
```

Si osservi la prima direttiva:

```
<form action="http://localhost/comments.exe" method="POST">
```

In essa si inizia un FORM e come azione associata al FORM, ovvero come CGI da innescare in seguito alla selezione del bottone “Invia Commenti”, lanceremo il CGI identificato da

```
http://localhost/comments.exe
```

facendo uso del metodo di passaggio di parametri detto “POST”. Esistono due metodi per inviare i parametri a un CGI.

**GET** i parametri sono passati al CGI per mezzo della variabile di ambiente QUERY\_STRING, e in questa variabile di solito non si possono memorizzare più di un centinaio di byte di dati;

**POST** i parametri vengono trasmessi al CGI per mezzo dello standard input del CGI senza alcuna limitazione di dimensione. In questo caso il CGI può conoscere quanti dati deve leggere da STDIN andando a leggere il contenuto di un'altra variabile di memoria CONTENT\_LENGTH.



name=Andrea+Bellini

Si noti come lo spazio venga simbolicamente rappresentato con il simbolo '+'. Il successivo campo relativo all'indirizzo e-mail:

e-mail: <input type="text" name="email">

ha il medesimo significato del precedente, ma con un diverso nome simbolico, "email". A questo punto la stringa di parametri che sarà passata al CGI si allunga:

name=Andrea+Bellini&email=guidi@mathema.com

Si osservi come le coppie

<nome=valore>

vengano separate per mezzo del simbolo "&". Il terzo campo:

```
<textarea name="comments" rows="10" cols="60">
</textarea>
```

non è concettualmente diverso dai primi due. Anche in questo caso sarà generata una coppia:

comments= <testo relativo ai commenti>

Gli indirizzi di tipo "mailto:" sono utilizzati per inviare e-mail. In questo caso, l'indirizzo è "mailto:guidi@mathema.com".

Mapedit. Il codice è riportato nel Listato 16.4 con molti commenti.

```
/* Indicare in quale directory inserire il file dei commenti */

#define COMMENT_FILE "c:\\http\\comments.txt"

#include <stdio.h>
#include <stdlib.h>

/* Variabili globali */

/* Numero massimo di campi gestiti nel form */
#define FIELDS_MAX 100

char *names[FIELDS_MAX];
char *values[FIELDS_MAX];

int fieldsTotal = 0;

/* Controlla che la richiesta provenga davvero da un modulo */
int VerifyForm();

/* Analizza i parametri passati, riempiendo gli array names[]
e values[] con informazione utile */
void ParseForm();
```

```
ParseForm();    /* OK, analizza il form. */

/* Usa l'informazione */

/* Trova l'indice di ogni campo nell'array */
for (i = 0; (i < fieldsTotal); i++) {
    if (!strcmp(names[i], "name")) {
        nameIndex = i;
    } else if (!strcmp(names[i], "email")) {
        emailIndex = i;
    } else if (!strcmp(names[i], "comments")) {
        commentsIndex = i;
    }
}

/* Se manca un campo, segnalalo */
if ((nameIndex == -1) || (emailIndex == -1) || (commentsIndex ==
-1)) {
    printf("<title></title>\n");
    printf("</head>\n");
    printf("<h1>Riempire tutti i campi</h1>\n");
    printf("Inserire nome, email e i commenti.\n");
    printf("Torna alla pagina precedente e riprova.\n");
    printf("</body></html>\n");
    return 0;
}
```

```

    if (strcmp(contentType, "application/x-www-form-urlencoded") !=
0) {
        bad = 1;
    }

    /* Controlla che si sia usato un metodo POST */
    requestMethod = getenv("REQUEST_METHOD");
    if (strcmp(requestMethod, "POST") != 0) {
        bad = 1;
    }

    return !bad;
}

/* Analizza l'informazione ricevuta e valorizza names e values[]*/

void ParseForm()
{
    char *contentLength = getenv("CONTENT_LENGTH");
    /* Numero di caratteri nei dati */
    int length;
    /* Prepara il buffer dove leggere i dati */
    char *buffer;
    /* Posizione corrente nel buffer mentre si cercano i separatori */
    char *p;

```

```

if (fieldsTotal == FIELDS_MAX) {
/* Basta, non ne posso accettare altri */
    return;
}

name = p;

/* Per prima cosa cerca il segno =. */
found = 0;
while (length) {
    if (*p == '=') {
        /* Termina il nome con un carattere null */
        *p = '\0';
        p++;
        found = 1;
        break;
    }
    p++;
    length--;
}
if (!found) {
/* Un vuoto o una voce troncata. È strano ma potrebbe accadere
*/
    break;
}

```

```

        free(names[fieldsTotal]);
        return;
    }

    /* Copia la stringa e risolve l'escape */
    UnescapeString(names[fieldsTotal], name);
    UnescapeString(values[fieldsTotal], value);
    fieldsTotal++;
    /* Continua con le altre coppie */
}
free(buffer); /* Libera il buffer. */
}

void FreeForm()
{
    int i;
    for (i=0; (i < fieldsTotal); i++) {
        free(names[i]);
        free(values[i]);
    }
}

void UnescapeString(char *dst, char *src)
{
    /* Cicla sui caratteri della stringa finché non trova il null */

```

```

        c = ascii;
    }
    *dst = c;
    src++;
    dst++;
}
*dst = '\\0';
}

```

## Listato 16.4 Un CGI per gestire il FORM dei commenti

Per linee generali il programma segue il seguente flusso:

- • invia (come sempre) l'intestazione della pagina di ritorno;
- • verifica la correttezza del FORM che gli è stato inviato (`VerifyForm`). In questo caso specifico si assicura che sia stato scelto un metodo POST per l'invio dei parametri, altrimenti risponde con una pagina che segnala l'errore;
- • analizza i parametri del FORM (`ParseForm`). In pratica con questa funzione si valorizzano due array:

```

char *names[FIELDS_MAX];
char *values[FIELDS_MAX];

```

Ogni elemento del primo array punta a un nome di parametro secondo l'ordine di trasmissione e, corrispondentemente, ogni elemento del secondo punta al valore associato a quel nome di parametro. All'interno della funzione `ParseForm` viene invocata la funzione `UnescapeString` che provvede a decodificare le eventuali sequenze di escape trasmesse nella sequenza di coppie `<nome=valore>`;

Ma mentre il mondo della Rete è in continua rapida evoluzione, il C prosegue la sua corsa spignata dopo essersi levato la soddisfazione di costituire la matrice indelebile da cui sono nati molti “nuovi” linguaggi.

## 16.9 Esercizi

1. 1. Scrivere un CGI che accetti in ingresso la data di nascita e calcoli il numero di anni e giorni trascorsi da quella data a oggi.
2. 2. Realizzare un CGI che legga lo stato attuale da un file di log e lo presenti in forma di pagine HTML.
3. 3. Scrivere tramite CGI un sistema che consenta:
  - • la registrazione
  - • la cancellazione
  - • la correzione

di un utente che intende abbonarsi a un generico servizio. [*Suggerimento*: Usare file tradizionali.]

4. 4. Scrivere un CGI che, basandosi su un metodo di generazione di numeri casuali, produca ogni volta una pagina con un colore diverso.

## Soluzione degli esercizi



## Capitolo 2

2

```
/* Determina il maggiore tra quattro valori */
#include <stdio.h>
main()
{
    int a, b, c, d;
    printf("\nDigita quattro valori interi distinti: ");
    scanf("%d", &a);
    scanf("%d", &b);
    scanf("%d", &c);
    scanf("%d", &d);

    if(a>b)
        if(a>c)
            if(a>d)
                printf("Il maggiore è: %d\n", a);
            else
                printf("Il maggiore è: %d\n", d);
        else
            if(c>d)
                printf("Il maggiore è: %d\n", c);
            else
                printf("Il maggiore è: %d\n", d);
}
```

```

    else
        printf("Il minore è: %d\n", c);
    }
    else {
        printf("Il maggiore è: %d\n", c);
        printf("Il minore è: %d\n", b);
    }
else
    if(b>c) {
        printf("Il maggiore è: %d\n", b);
        if(a<c)
            printf("Il minore è: %d\n", a);
        else
            printf("Il minore è: %d\n", c);
    }
    else {
        printf("Il maggiore è: %d\n", c);
        printf("Il minore è: %d\n", a);
    }
}

```

## 5

```

/* Verifica il valore delle espressioni date */
#include <stdio.h>
int a, b, c, ris;
main()
{
    a = 5;
    b = 35;
    c = 7;

```

```

printf("\n d) %d", (a < b) && (a < b));
printf("\n e) %d", (a + b) || (a > b));
printf("\n f) %d", ((a * c) - b) || (a > b));
printf("\n g) %d", ((a * c) != b) || (a > b));
printf("\n h) %d\n", (a > b) || (a < c) || (c == b));
}

```

I risultati visualizzati dal programma saranno i seguenti:

- a) 250
- b) 0
- c) 40
- d) 1
- e) 1
- f) 0
- g) 0
- h) 1

## 6

I risultati visualizzati dal programma saranno i seguenti:

- a) Vero (1)
- b) Falso (0)
- c) Vero (1)
- d) Vero (1)
- e) Falso (0)
- f) Vero (1)

## 7

```

        break;
    case 'b':
        printf("\n In esecuzione l'opzione b");
        break;
    case 'c':
        printf("\n In esecuzione l'opzione c");
        break;
    case 'd':
        printf("\n In esecuzione l'opzione d");
        break;
    case 'e':
        printf("\n In esecuzione l'opzione e");
        break;
    default:
        printf("\n Opzione inesistente");
        break;
}
}

```

## 10

```
printf("\n Il maggiore è: %d", (x=(a>b)?a:b)?c:x:c);
```

Dove x è una variabile int.

## Capitolo 3

```

printf("MAGGIORE MINORE E MEDIA\n");
min = INT_MAX;
max = INT_MIN;
media = 0;
i = 1;

do {
    printf("\nLunghezza della sequenza: ");
    scanf("%d", &n);
}
while(n<1);

for(i=1; i<=n; i++) {
    printf("Valore int.: \t");
    scanf("%d", &numero);
    if(numero>max)
        max = numero;
    else
        if(numero<min)
            min = numero;
    media = media+numero;
}

printf("Maggiore: %d\n", max);
printf("Minore: %d\n", min);
printf("Media: %d\n", media/n);
}

```

```
        desidera l'utente con caratteri e dimensioni
        scelti a tempo di esecuzione */
#include <stdio.h>
main()
{
    int i, j, y, linee, colonne, volte;
    char cornice, interno;

    do {
        printf("\nNumero di linee: ");
        scanf("%d", &linee);
    }
    while(linee<1);

    do {
        printf("\nNumero di colonne: ");
        scanf("%d", &colonne);
    }
    while(colonne<1);

    printf("\nCarattere della cornice: ");
    scanf("%ls", &cornice);
    printf("\nCarattere dell'interno: ");
    scanf("%ls", &interno);

    do {
        printf("\nNumero di visualizzazioni: ");
        scanf("%d", &volte);
    }
```

```

    if(voti[i]>max)
        max = voti[i];
    else
        if(voti[i]<min)
            min = voti[i];
    media = media+voti[i];
}

```

### 13.

Devono essere definite le dimensioni della matrice.

```

#define N 10
#define P 10
#define M 10

int mat1[N][P];
int mat2[P][M];
int pmat[N][M];

```

Si devono richiede all'utente le reali dimensioni e si deve controllare che il loro valore non superi le dimensione delle matrici. I valori da richiede sono soltanto tre in quanto le colonne della prima matrice devono essere in numero uguale alle righe della seconda.

```

/* Richiesta delle dimensioni */
do {
    printf("Numero di linee I matrice: ");
    scanf("%d", &n);
}

```

```

main()
{
    int i, j;

    printf("\n \n CARICAMENTO DEI VOTI \n \n");
    for(i=0; i<n-1; i++)
        for(j=0; j<m-1; j++) {
            printf("Ins. studente %d prova %d: ", i+1, j+1);
            scanf("%f", &voti[i][j]);
        };

    /* Calcolo medie per studente */
    for(i=0; i<n-1; i++) {
        voti[i][m-1] = 0;
        for(j = 0; j < m-1; j++)
            voti[i][m-1] = voti[i][m-1] + voti[i][j];
        voti[i][m-1] = voti[i][m-1] / (m-1);
    }

    /* Calcolo medie per prova */
    for(j=0; j<m; j++) {
        voti[n-1][j] = 0;
        for(i=0; i<n-1; i++)
            voti[n-1][j] = voti[n-1][j] + voti[i][j];
        voti[n-1][j] = voti[n-1][j]/(n-1);
    }

    printf("\n \n VISUALIZZAZIONE DELLA MATRICE \n ");
    for(i=0; i<n; i++) {
        printf("%d\t", voti[i][0]);
        for(j=1; j<m; j++)
            printf("%f\t", voti[i][j]);
        printf("\n");
    }
}

```



## Capitolo 5

### 1.

Qualsiasi soluzione si adotti tra quelle proposte nel testo, l'operatore relazionale dell'`if`, che controlla lo scambio di valori tra gli elementi deve essere cambiato da `>` a `<`.

```
if(vet[i]<vet[i+1])
```

Non ci sono altre modifiche da effettuare.

### 2.

```
/* Ricerca di un valore in una matrice */  
#include <stdio.h>
```

```
#define N 10  
#define M 10  
char alfa[N][M];
```

```
main()  
{  
    int n, m, i, j, k;  
    char ric;
```

```
/* Richiesta delle dimensioni */  
do {
```

```

/ ... ricerca del carattere all'interno della matrice ...
k = 0;
for(i=0; i<n; i++)
    for(j=0; j<m; j++) {
        if(alfa[i][j]==ric) {
            printf("%c in linea %d colonna %d\n", ric, i+1, j+1);
            k = 1;
        }
    };
if(k==0) printf("%c non presente nella matrice", ric);
}

```

### Esempio di esecuzione

Carattere da ricercare: a

#### VISUALIZZAZIONE DELLA MATRICE

t	b	a
m	d	g
a	k	k
d	a	m
v	f	g

```

a in linea 1 colonna 3
a in linea 3 colonna 1
a in linea 4 colonna 2

```

```

char prima[160] = "Analisi, requisiti";

main()
{
char seconda[80];
int i, x;

for(i=0; ((seconda[i]=getchar())!='\n') && (i<80); i++)
    ;
seconda[i]='\0';
if((x=(strcmp(prima, seconda, 5)))==0)
    printf("Sono uguali\n");
else
    if(x>0)
        printf("la prima è maggiore della seconda\n");
    else
        printf("la seconda è maggiore della prima\n");
}

```

## Capitolo 7

### 1

```

double pot(double base, int esp)
{
double po;

```

```
while( r-- ) putchar(' ');  
printf("%s\n\n", messaggio);  
}
```

### 3

```
#include <stdio.h>  
char buf[128];  
void min_man(void);  
  
main()  
{  
    printf("\nInserisci stringa: ");  
    scanf("%s", buf);  
  
    min_man();  
    printf("%s\n", buf);  
}  
  
void min_man(void)  
{  
    int i;  
  
    for ( i = 0; buf[i] != '\0'; i++)  
        if (buf[i] >= 'a' && buf[i] <= 'z')  
            buf[i] = buf[i] - 'a' + 'A';  
}
```

```

        case '2':
        case '3':
        case '4':
        case '5':
        case '6':
        case '7':
        case '8':
        case '9':
            numeri++;
            break;
        default:
            alfa++;
            break;
    }
}

```

## 5

È sufficiente aggiungere alla fine della funzione `immissione` la chiamata alla procedura, già presente nel programma, che effettua l'ordinamento, passandole il numero di elementi che la compongono.

```

int immissione2()
{
    ...
    ordinamento( n );
    return( n );
}

```

```

        NULL
    };
char **p = vet;

main()
{
    while(*p != NULL)
        printf("%s", *p++);
}

```

### 3

```

char *str_in_str(char *s, char *t)
{
    char *v;

    while(*s != '\0') {
        if(*s == *t)
            for(v = t; *s == *v;) {
                if(++v == '\0') return(s-(v-t)+1);
                if(*s++ == '\0') return(NULL);
            }
        else
            s++;
    }
    return(NULL);
}

```

## 5

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

main()
{
    char *s;
    int  n;
    printf("Inserire dimensione del buffer : ");
    scanf("%d", &n);
    s = (char*) malloc(n+1);
    ...
}
```

## 6

La funzione viene così dichiarata:

```
void immissione( int *, int * );
```

Il primo parametro è il puntatore alla variabile `n` di gestione\_sequenza dove immissione memorizzerà, mediante il puntatore `pn`, la lunghezza della sequenza.

```
void immissione( int *pn, int *vet )
{
    int i, n;
```

```

scanf("%f", &b);
printf("Inser. esponente: \t");
scanf("%d", &e);
printf("Potenza: %lf\n", potenza(b, e));
}

/* Funzione per il calcolo di base elevato a esp con esp>=0 */
double potenza(float base, int esp)
{
double pot = 1;
if(esp==0) return(1); /* caso esponente uguale a zero */
if(esp>0) { /* calcolo della potenza */
do
pot = pot*base; /* base*base*base.... esp volte */
while(--esp>0);
}
return(pot);
}

```

## 5

```

/* Funzione il calcolo di base elevato a esp
   esp può essere un intero qualsiasi */
double potenza(float base, int esp)
{
int s = 1;
double pot = 1; /* inizializzazione di pot a 1
                 caso di esponente uguale a zero */
if(esp<0) { /* l'esponente è negativo ? */

```



7

```
/* Massimo comun denominatore di due interi positivi */
#include <stdio.h>
int mcd(int, int);

main()
{
    int t, k;
    printf("\n\n Calcolo del massimo comun divisore \n\n");
    printf("Inser. t: \t");
    scanf("%d", &t);
    printf("Inser. k: \t");
    scanf("%d", &k);
    printf("Massimo comun divisore: %d\n", mcd(t, k));
}

/* Funzione ricorsiva per il calcolo del massimo comun divisore */
int mcd(int t, int k)  {
    if(k==0)
        return(t);
    else
        if(k>t)
            return(mcd(k, t));
        else
            return(mcd(k, t%k));
}
```

```

printf("Digitare il nome del file: ");
gets(nomefile);
fp = fopen(nomefile, "w");
numwrite = fwrite(buffer, sizeof(char), 80, fp);
printf("%d caratteri scritti sul file %s\n", numwrite, nomefile);
printf("Controllare visualizzando il file con un editor\n");
printf("del sistema operativo in uso (esempio type, vi o
wordpad)\n");
}

```

### 3

Come in Procedura Anagrafica, basta sostituire la struttura

struct per

con le strutture

```

struct dat
{int giorno, mese, anno};

```

```

typedef struct dat Data;
struct student {
    char *cog-stud;
    char *nom_stud;
    char *ind_stud;

```

```
}
```

**5**

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

```
main()
```

```
{
```

```
int i;
```

```
FILE *fp;
```

```
char sequenza[80];
```

```
/* Apre il file. Il simbolo di root \ si ottiene con il  
doppio slash */
```

```
fp = fopen("c:\\autoexec.bat", "r");
```

```
printf("Le prime 10 sequenze di caratteri sono:\n");
```

```
for(i = 0; i < 10; i++) {
```

```
    if (fscanf(fp, "%s", sequenza) == EOF){
```

```
        printf("Fine file!\n");
```

```
        break;
```

```
    }
```

```
else
```

```
    printf("Sequenza %d = \"%s\"\n", i, sequenza);
```

```
}
```

```
}
```

EOF è una costante di libreria che rappresenta la End Of File, cioè la fine del file.

```

punt_lista = crea_lista2();
visualizza_lista(punt_lista);

/* chiamata prima versione di conta pari */
conta_pari(punt_lista, &pari, &dispari);
printf("\nPari: %d    Dispari: %d", pari, dispari);

/* chiamate seconda versione di conta pari */
printf("\nPari: %d", conta_pari2(punt_lista, &dispari));
printf("    Dispari: %d\n", dispari);
}

void conta_pari(struct elemento *p, int *ppari, int *pdispari)
{
    *ppari = *pdispari = 0;

    while(p!=NULL) {
        if(p->inf % 2 == 0)
            (*ppari)++;
        else
            (*pdispari)++;
        p = p->pun;
    }
}

conta_pari2(struct elemento *p, int *pdispari)
{
    int pari = 0;
    *pdispari = 0;

```

```
main()
{
    struct elemento *punt_lista;

    punt_lista = crea_lista2();
    visualizza_lista(punt_lista);

    punt_lista = elimina_pari(punt_lista);
    visualizza_lista(punt_lista);
}

struct elemento *elimina_pari(struct elemento *p)
{
    struct elemento *paus;
    int logica = 1;

    while(p!=NULL && logica)
        if(p->inf % 2 == 0)
            p = p->pun;
        else
            logica = 0;

    paus = p;
    while(paus->pun != NULL)
        if(paus->pun->inf % 2 == 0)
            paus->pun = paus->pun->pun;
        else
            paus = paus->pun;
```

```

    punt_negativi = aggiungi(punt_negativi, x);
}
while(x.inf!=0);

visualizza_lista(punt_positivi);
visualizza_lista(punt_negativi);
}

struct elemento *aggiungi(struct elemento *p, struct elemento x)
{
    struct elemento *paus;

    if(p==NULL) {
        p = (struct elemento *)malloc(sizeof(struct elemento));
        p->inf = x.inf;
        p->pun = NULL;
    }
    else {
        paus = (struct elemento *)malloc(sizeof(struct elemento));
        paus->inf = x.inf;
        paus->pun = p;
        p = paus;
    }
    return(p);
}

```

```

        if(val < p->inf)
            p->alb_sin = crea_nodo2(p->alb_sin, val);
        else
            ++p->occorrenze;
    }
    return(p);
}

```

### 3

```

void simmetrico(struct nodo *p)
{
    if(p!=NULL) {
        simmetrico( p->alb_sin );
        printf("\n%d  %d", p->inf, p->occorrenze);
        simmetrico( p->alb_des );
    }
}

```

### 4

```

void differito(struct nodo *p)
{
    struct nodo *paus = p;
    paus = paus->p_arco;

    printf("(");
}

```

```
printf("\nImmettere l'etichetta di un nodo: ");  
scanf("%c", &invio);  
scanf("%c", &etichetta);  
giro( etichetta );  
}
```

```
void giro(char e)  
{  
    int i = 0;  
    printf("\n");  
    while(i<n && s[i].inf!=e) i++;  
    if (i==n) return;  
    vai(i);  
}
```

```
void vai(int i)  
{  
    struct successore *p;
```

```
    p = s[i].pun;  
    while(p!=NULL) {  
        if(gia_visitati[p->inf]==0) {  
            gia_visitati[p->inf] = 1;  
            printf(" %c", s[p->inf].inf);  
            vai( p->inf );  
        }  
        p = p->pun;  
    }  
}
```



int  
long  
register  
return  
short  
signed  
sizeof  
static  
struct  
switch  
typedef  
union  
unsigned  
void  
volatile  
while

## Appendice C: File header

Ogni funzione è associata ad uno o più *file header* che devono essere inclusi ogni volta che si fa uso di quella funzione. Tali file contengono: le dichiarazioni delle funzioni correlate, delle macro, dei tipi dati e la definizione di costanti necessari all'esecuzione di un insieme di funzioni di libreria. La definizione di queste ultime dipende poi dall'implementazione del compilatore in uso. Di seguito riportiamo l'elenco dei file header standard, di molti dei quali abbiamo parlato estesamente nel testo.

	contenere la differenza, con segno, tra due puntatori e <code>size_t</code> il tipo (intero privo di segno) prodotto dalla funzione <code>sizeof</code> .
<code>&lt;stdio.h&gt;</code>	Input e Output. Funzioni quali <code>printf</code> e <code>scanf</code> .
<code>&lt;stdlib.h&gt;</code>	Utilità generale. Per esempio le funzioni per la conversione dei numeri, come <code>atof</code> , che trasforma una stringa in un <code>double</code> , o <code>rand</code> che ritorna un numero pseudo casuale.
<code>&lt;string.h&gt;</code>	Gestione di stringhe. Funzioni quali <code>strcpy</code> , che consente di copiare una stringa su un'altra e <code>strcat</code> che concatena due stringhe.
<code>&lt;time.h&gt;</code>	Gestione della data e dell'ora. Per esempio la funzione <code>time</code> che ritorna l'ora corrente.

Lo standard ANSI garantisce che inclusioni multiple di uno stesso file header standard non portano alcun effetto negativo e che l'ordine di inclusione è influente. Dunque un header può essere incluso in qualsiasi ordine e un qualsiasi numero di volte; come sappiamo, deve comunque essere incluso prima che venga utilizzata una qualsiasi entità in esso definita.

13	Ɔ	45	—	77	М	109	м	141	ì	173	ı	205	=	237	Ÿ
14	Ɔ	46	.	78	Н	110	н	142	ǎ	174	«	206	≡	238	˙
15	✱	47	/	79	О	111	о	143	ǎ	175	»	207	⊙	239	/
16	►	48	0	80	Р	112	р	144	É	176	▤	208	δ	240	˙
17	◄	49	1	81	Q	113	q	145	æ	177	▥	209	Đ	241	±
18	⬆	50	2	82	R	114	r	146	fl	178	▧	210	Ê	242	=
19	!!	51	3	83	S	115	s	147	ô	179		211	Ë	243	¾
20	¶	52	4	84	T	116	t	148	ö	180	†	212	È	244	¶
21	§	53	5	85	U	117	u	149	ò	181	Á	213	ı	245	§
22	—	54	6	86	V	118	v	150	û	182	Â	214	Í	246	÷
23	‡	55	7	87	W	119	w	151	ù	183	À	215	Î	247	˘
24	↑	56	8	88	X	120	x	152	ÿ	184	©	216	Ï	248	°
25	↓	57	9	89	Y	121	y	153	ö	185	¶	217	J	249	ˆ
26	→	58	:	90	Z	122	z	154	ü	186		218	Г	250	˙
27	←	59	;	91	[	123	{	155	ø	187	¶	219	■	251	ı
28	└	60	<	92	\	124	ı	156	£	188	¶	220	■	252	ž
29	•	61	=	93	]	125	}	157	Ø	189	¢	221	ı	253	ž
30	▲	62	>	94	^	126	˜	158	×	190	¥	222	ì	254	■
31	▼	63	?	95	_	127	△	159	f	191	l	223	■	255	