# Rudimenti di programmazione in C
# Note per il corso di Informatica 1
# Corso di Laurea in Matematica

Agostino Dovier
Dip. di Matematica e Informatica, Univ. di Udine
Via delle Scienze 206, 33100 Udine (Italy).
dovier@dimi.uniud.it
10 GIUGNO 2004

# Indice


Introduzione 5

Capitolo 1. La nozione di algoritmo 7

Capitolo 2. Introduzione per esempi al linguaggio **C** 9
1. Input-output 9
2. Costrutto condizionale 12
3. Il ciclo `for` 12
4. Il ciclo `while` 13
5. Vettori 14
6. Identificatori e Parole riservate 15
7. Esercizi/esempi 15

Capitolo 3. Tipi 17
1. Tipi semplici built-in 17
2. Tipi semplici definiti dall'utente 23
3. Tipi strutturati 24

Capitolo 4. Funzioni 29
1. Sintassi 29
2. Parametri 31
3. Il tipo di una funzione 34
4. Variabili locali e globali 35
5. Le istruzioni `switch` e `do while` 36

Capitolo 5. La ricorsione 37
1. Motivazioni 37
2. Esempi 38
3. Ottimizzazione di funzioni ricorsive 39

Capitolo 6. Esercizi 41
1. Aritmetica su interi grandi a piacere 41
2. Calcolo e stampa (primitiva) di funzioni 43
3. esercizi grafici 45
4. Ordinamento di vettori 45
5. Visite di grafi 49

Capitolo 7. Alcuni approfondimenti 51
1. Il Preprocessore **C** 51
2. La libreria standard 52
3. I file 53

# Introduzione

Queste note non vogliono costituire né un manuale del linguaggio **C**, né un corso di programmazione. Per entrambi gli argomenti esistono centinaia di ottimi testi. Vanno considerate solo ed unicamente un aiuto per la sperimentazione in laboratorio della programmazione usando il linguaggio di programmazione **C**. Un ambiente di programmazione in **C** per il Sistema Operativo Windows, completo di editor e (eventualmente) debugger si può scaricare dal sito `http://cs-alb-pc3.massey.ac.nz/`. Ci riferiremo a tale ambiente come a JFE. Quanto scritto vuol essere tuttavia il più possibile indipendente dall'editor/compilatore **C** utilizzato per le sperimentazioni. Le note iniziano con una breve discussione circa la nozione di *algoritmo*.

Ringrazio Demis Ballis per l'attenta lettura e le osservazioni.

CAPITOLO 1

# La nozione di algoritmo

Discuteremo ora le caratteristiche che deve avere un algoritmo partendo da quella che è l'esperienza e l'idea intuitiva che tutti ne abbiamo. Un algoritmo viene descritto in un certo linguaggio, che può anche essere semplicemente l'italiano, così come usando i linguaggi nati appositamente per descrivere algoritmi quali i linguaggi di programmazione e i sistemi formali che vedremo nei prossimi capitoli. Facciamo qui riferimento all'esperienza personale e immaginiamo di guardare un algoritmo. Possiamo concordare che, indipendentemente dal problema che intende risolvere, ha delle caratteristiche comuni.

- **a:** Un algoritmo è di lunghezza finita.
- **b:** Esiste un agente di calcolo che porta avanti il calcolo eseguendo le istruzioni dell'algoritmo.
- **c:** L'agente di calcolo ha a disposizione una memoria dove vengono immagazzinati i risultati intermedi del calcolo.
- **d:** Il calcolo avviene per passi discreti.
- **e:** Il calcolo non è probabilistico.

I punti **a**–**c** hanno una ovvia interpretazione. Il punto **d** afferma che il calcolo non avviene mediante dispositivi analogici. Il punto **e** afferma che il calcolo non obbedisce a nessuna legge di probabilità. Associato al punto **a** parleremo di *espressioni simboliche* ovvero espressioni in un linguaggio di simboli che costituisce il linguaggio per decrivere un algoritmo.

Altre caratteristiche degli algoritmi sono:

- **f:** Non deve esserci alcun limite finito alla lunghezza dei dati di ingresso.
- **g:** Non deve esserci alcun limite alla quantità di memoria disponibile.

Mentre il punto **f** è ragionevole: ad esempio un algoritmo di somma deve poter funzionare per ogni possibile addendo, ovvero numero naturale, per il punto **g** è necessario un chiarimento. Per evidenziare la necessità di assumere una memoria illimitata facciano osservare che, limitandola, alcuni algoritmi noti per calcolare semplici funzioni non potrebbero funzionare. Ad esempio la funzione $f(x) = x^2$ non sarebbe calcolabile poiché lo spazio di memoria necessario per calcolare il quadrato di $x$ dipende da $x$, e per il punto **f**, esso deve essere illimitato.

Le seguenti osservazioni sono essenziali per comprendere la natura del calcolo e la sua complessità.

- **h:** Deve esserci un limite finito alla complessità delle istruzioni eseguibili dal dispositivo.
- **i:** Sono ammesse esecuzioni con un numero di passi finito ma illimitato.

Il punto **h**, in relazione al punto **a**, stabilisce la intrinseca finitezza del dispositivo di calcolo. Ad esempio, pensando ad un calcolatore, esso sarà in grado di calcolare solo indirizzando direttamente una parte finita della sua memoria (registri o

memoria RAM) stabilendo quindi un limite nella complessità delle istruzioni eseguibili. Questo è un punto chiave nell'analisi che vedremo della nozione di effettiva calcolabilità. Un dispositivo di calcolo può effettivamente tenere traccia solo di un numero finito di simboli, ovvero esso può reagire (eseguendo un istruzione) tenendo conto di un numero finito di simboli ricordati. Questo venne intuitivamente giustificato da Turing che per primo analizzò formalmente il concetto di effettiva calcolabilità, con l'intrinseca limitazione della memoria umana. Tuttavia, non c'è limite alla memoria ausiliaria, come visto nel punto **g**. Per quanto riguarda il punto **i**, osserviamo che non essendo limitabile il numero di passi richiesti per eseguire un generico algoritmo (si pensi alla moltiplicazione o alla funzione $x^2$ discussa nel punto **g**), non è possibile stabilire a priori un limite massimo sul numero di passi nell'esecuzione di un algoritmo. Uno studio approfondito su come legare il numero di passi alla lunghezza dei dati è argomento trattato nella teoria della *complessità* degli algoritmi.

CAPITOLO 2

# Introduzione per esempi al linguaggio C

In questo capitolo si vuol fornire una introduzione informale al linguaggio **C** mediante semplici esempi. Prenderemo confidenza, in particolare, con le nozioni di *struttura di un programma*, di *commento*, di *tipo di dato intero* e *vettore* (di interi), con le funzioni di libreria di input e output `scanf` e `printf`, con il costrutto condizionale `if ... else`, e con i cicli `for` e `while`.

## 1. Input-output

Un programma **C** consta di varie parti, ma due di queste sono sempre presenti: l'*intestazione* del programma (`main()`) ed il *corpo* del programma stesso, delimitato da due parentesi graffe (una aperta ed una chiusa) entro le quali sono presenti zero (anche se avrebbe poco senso) o più *istruzioni*. Un (semplicissimo) programma è il seguente:

```
main(){
   printf("Buongiorno a voi");
}
```

Una volta creato un nuovo *file* con un programma editor (nell'ambiente JFE, si selezioni la tendina "`File`" e quindi si scelga il comando "`New`") si copino le 3 righe sopra e si salvi il file. Il file deve essere prima *compilato* (in JFE, si scelga la tendina "`Compiler`" e si selezioni il comando "`Compile`"—oppure si usi l'abbreviazione `F9`) e, se la compilazione va a buon fine (ovvero non abbiamo commesso errori nel ricopiare le 3 righe sopra) possiamo *eseguirlo* (in JFE, si scelga ancora la tendina "`Compiler`" e quindi si scelga il comando "`Run`"—oppure si usi l'abbreviazione `Ctrl-F9`).

Nel programma scritto identifichiamo le due parti: intestazione e corpo. Come ci si può attendere, durante l'*esecuzione* del programma comparirà sul monitor il messaggio `Buongiorno a voi`. La funzione di libreria **C** `printf` dunque ha il compito di fornire un 'output' da parte del programma verso il programmatore.

NOTA 1. *Ad essere rigorosi, durante la compilazione potrebbero essere stampati alcuni messaggi di* warning. *Ciò è sconfornante viste le dimensioni del programma, ma per ora non preoccupiamoci. Warning non sono necessariamente errori. La versione che più piace al compilatore (in particolare allo standard ANSI C++) sarebbe comunque:*

```
#include <stdio.h>
int main(){
   printf("Buongiorno a voi");
   return 0;
}
```

*Qualora i messaggi di warning ci annoiassero, possiamo decidere di disabilitarli (in JFE, selezionate* `Compliler`*, quindi* `Options`*).*

Il seguente programma, simile al precedente:

```
main(){
  /* Programma che augura
     il Buongiorno */
  printf("Buongiorno a voi");
}
```

non presenta alcun tipo di differenza durante l'esecuzione. In **C** è possibile aggiungere al programma delle informazioni che vengono ignorate dal compilatore (e dunque non hanno effetto durante le esecuzioni). Queste informazioni sono dette *commenti* e vengono introdotte per facilitare la *leggibilità* del programma. Un commento inizia con la sequenza di simboli `/*` e termina con la sequenza (inversa) `*/`.

Se sostituiamo l'istruzione `printf(Buongiorno a voi);` con le due istruzioni seguenti:

```
printf("Buongiorno ");
printf("a voi");
```

non si ottiene ancora alcuna differenza durante l'esecuzione. Se invece la sostituissimo con:

```
printf("Buongiorno \n");
printf("a voi");
```

oppure con la singola istruzione

```
printf("Buongiorno \n  a voi");
```

ci accorgeremmo che l'output diventerebbe:

```
Buongiorno
a voi
```

L'istruzione `printf` permette di stampare su monitor una *stringa*, ovvero una sequenza di caratteri che vengono scritti tra doppi apici. La sequenza dei due caratteri `\n` viene interpretata come un comando per 'andare a capo'.

Analizziamo altre potenzialità della funzione `printf`. Il programma:

```
main(){
   int a;

   a = 2;
   printf("Il valore della variabile a e': a");
}
```

ci permette di introdurre il concetto di *variabile*. La prima riga del corpo del programma serve a *dichiarare* che nel programma stesso vorremo far uso della variabile `a`. Dice inoltre che la variabile assumerà (indicazione essenziale per il compilatore) valori di tipo *intero* (`int`). In ogni programma che utilizza variabili è presente una fase iniziale detta *dichiarazione delle variabili*.

La prima riga del programma vero e proprio costituisce una istruzione di *assegnamento*. L'esecuzione di tale istruzione fa sì che la variabile `a` assuma il valore (intero) `2`. L'esecuzione del programma produrrà come output:

```
Il valore della variabile a e': a
```

Tuttavia potremmo essere più interessati a scoprire il *valore* della variabile `a`, ovvero `2`. Per far ciò bisogna sostituire l'istruzione sopra con l'istruzione:

```
printf("Il valore della variabile a e': %d ", a);
```

che produrrà come output:

```
Il valore della variabile a e': 2
```

Prima di capire bene il perchè, effettuiamo altri due esperimenti. Sostituendo l'istruzione di stampa con:

```
printf("Stampiamo due volte il valore della var. a: %d, %d", a);
```

otterremo come output:

```
Stampiamo due volte il valore della var. a: 2 -545713792
```

Tale strano numero dipende dalla macchina. Mentre con

```
printf("Stampiamo due volte il valore della var. a: %d, %d ", a, a);
```

si ottiene:

```
Stampiamo due volte il valore della var. a: 2 2
```

Usando, infine, l'istruzione:

```
printf("Il doppio di %d e': %d",a,a*2);
```

si ottiene:

```
Il doppio di 2 e': 4
```

Oltre al comando per andare a capo, entro la stringa che si vuol stampare si può inserire anche la cosiddetta *direttiva di output* `%d`. Ogni qualvolta `%d` è presente nella *stringa di output* viene ordinatamente stampato il valore (`d` sta per *decimale*) della variabile o dell'espressione presente dopo la stringa suddetta. Se vi sono più occorrenze di tale direttiva, verranno stampate ordinatamente le variabili o le espressioni che compaiono, separate ciascuna da una virgola, a destra di tale stringa.

Generalizziamo l'esempio mediante la possibilità di accettare valori dall'utente per le variabili durante l'esecuzione (al *run-time*).

```
main(){
    int a;

    printf("Dammi un numero: ");
    scanf("%d",&a);
    printf("Il doppio di %d e' %d ", a, 2*a);
}
```

Durante l'esecuzione viene stampato:

```
Dammi un numero:
```

L'esecuzione del programma continua quando l'utente fornisce una valore di input. Supponiamo di inserire il numero `17`; l'output sarà:

```
Il doppio di 17 e' 34
```

Si osservi che la sequenza di caratteri `%d` è presente anche nell'istruzione di `scanf` e con lo stesso significato. Si noti pure la presenza (indispensabile, il perchè lo capiremo nel Capitolo 2) del simbolo `&` prima della variabile `a` in accettazione.

Qualora si desideri stampare i caratteri %, ", e \ che sono dei caratteri speciali usati il primo per iniziare una direttiva, il secondo per aprire e chiudere una stringa,

il terzo per inserire caratteri speciali in una stringa, si deve scrivere nella stringa di output, rispettivamente, %%, \', e \\.

## 2. Costrutto condizionale

Gli esempi della sezione precedente sono tutti contraddistinti dal fatto che il *flusso d'esecuzione* del programma procede sequenzialmente dalla prima all'ultima istruzione del programma, senza saltarne alcuna. Il costrutto `if ... else`, argomento di questa sezione permette al flusso di esecuzione di diramarsi in più strade, a seconda del risultato della valutazione di condizioni al run-time.

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

    printf("Dammi un numero: ");
    scanf("%d",&a);
    printf("Dammi un altro numero: ");
    scanf("%d",&b);
    if (a > b)
         printf("Il piu' grande e' %d ", a);
    else printf("Il piu' grande e' %d ", b);
}
```

A seconda della coppia di valori che forniamo in input durante l'esecuzione, esattamente una tra le due istruzioni di stampa sarà eseguita. Il test che permettere di discernere tra le due è il test `(a > b)`.

Estendiamo ora il programma cercando di catturare bene il caso in cui i due valori di ingresso sono uguali:

```
if (a > b)
     printf("Il piu' grande e' %d ", a);
else if (b > a)
          printf("Il piu' grande e' %d ", b);
     else printf("Sono uguali");
```

Si osservi come la possibilità di *annidare* costrutti condizionali permetta di scrivere programmi in grado di prendere un numero arbitrario di strade, anche se il costrutto preso da solo sembra permettere di gestire solo due possibilità.

I caratteri usati per *rientrare a destra* in ogni riga sono ininfluenti per il calcolatore. Come consiglio generale, tuttavia, si consiglia di cercare di evidenziare la struttura dei vari `if ... else` per far comprendere la struttura del programma al lettore umano.

## 3. Il ciclo `for`

Siamo dunque in grado di selezionare le istruzioni da eseguire a seconda del valore assunto dalle variabili durante l'esecuzione. I programmi così ottenuti sono tuttavia ancora *unidirezionali* ovvero la loro esecuzione procede dall'inizio verso la fine senza possibilità di *ritornare indietro*.

Il costrutto che incontreremo in questa sezione permette di ovviare in parte a questa lacuna, consentendo la *ripetizione* (controllata) di una o più istruzioni.

L'effetto del seguente programma:

```
main(){
     int a,i;

     printf("Dammi un numero: ");
     scanf("%d",&a);
     for (i = 1; i <= a; i = i + 1)
       printf("Ciao \n");
}
```

è quello di stampare in output `ciao` un certo numero (che decidiamo noi assegnando il valore ad `a`) di volte. Il significato di

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

è infatti quello di assegnare il valore iniziale `1` alla variabile `a`. Si verifica quindi se il valore di `i` è minore o uguale a quello di `a`. In caso affermativo viene eseguita l'istruzione seguente (`printf`). A questo punto la variabile `i` viene aggiornata (`i = i + 1`) e si ripete tutto ciò. Alla luce di ciò è interessante chiedersi (esercizio) quale sia l'effetto della sostituzione del ciclo `for` sopra con il seguente:

```
for (i = 1; i <= a; i = i)
  printf("Ciao \n");
```

L'istruzione di incremento di uno è estremamente utilizzata in programmazione, tanto da indurre i progettisti del **C** (e di altri linguaggi) ad inserire una sua abbreviazione:

`i = i + 1` è equivalente all'abbreviazione `i++`

## 4. Il ciclo `while`

Argomento di questa sezione è il costrutto `while` il quale permette di iterare una (o più) istruzioni fino a che il test diventa falso. L'effetto del programma:

```
main(){
  int i;

  i = 0;
  while (i < 5){
      printf("Ciao \n");
      i = i + 1;
  }
}
```

è quello di stampare 5 volte `ciao`. Visto così sembrerebbe simile al costrutto `for`. In effetti, l'istruzione `for`

```
for (i = 1; i < 5; i = i + 1)
   printf("aaa");
```

può essere equivalentemente rimpiazzata da:

```
i = 1;
while (i < 5){
    printf("aaa");
    i = i + 1;
}
```

Tuttavia, diversamente dal `for`, il ciclo `while` non è basato su un'idea di una variabile che dev'essere aggiornata ad ogni ciclo finché raggiunge un dato valore. Il `while` consta di una *condizione* e di un *corpo*. Vi è una condizione che stabilisce se ripetere il ciclo che può coinvolgere una o più variabili. Queste possono essere aggiornate (o meno) nel corpo del while in modo esplicito (come nel terzo parametro del `for`) o implicito (mediante effetto di altre procedure/funzioni eventualmente presenti nel corpo del while).

Il costrutto `while` è fondamentale in un linguaggio di programmazione imperativo; il teorema di Böhm-Jacopini assicura che *assegnamento* e ciclo `while` sono sufficienti a scrivere qualunque programma. Abbiamo già visto come si riesca a simulare il `for`. Mostriamo come si possa simulare il costrutto condizionale. La condizione del `while` può essere più complessa di quelle viste finora (saremo più precisi nella sezione 1.2). Ad esempio è possibile combinare in *congiunzione* (**and** logico) due condizioni semplici (p. es., `i < 1` e `a >= 5`) mediante &&.

Il frammento di codice

```
if (a > b)
    printf("aaa");
else printf("bbb");
```

è simulato da (si assuma che la variabile `i` non occorra altrove nel programma, altrimenti si scelga un'altra variabile):

```
i = 0;
while (a > b && i == 0){
    printf("aaa");
    i = 1;
}
while (i == 0){
  printf("bbb");
  i = 1;
}
```

## 5. Vettori

Concludiamo la panoramica informale dei costrutti del linguaggio **C** con l'introduzione del tipo di dato *vettore* che permette di memorizzare ed accedere in modo compatto ed uniforme ad un numero arbitrariamente grande di dati omogenei (nella sezione 3.1 si darà una definizione più rigorosa di questo importante concetto). Si consideri il seguente programma:

```
main(){
  int v[6],i,Tot;

  for (i = 1; i <= 5; i = i + 1){
     printf("Dammi il numero %d -esimo: ",i);
     scanf("%d",&v[i]);
  }
  Tot = 0;
  for (i = 1; i <= 5; i = i + 1)
     Tot = Tot + v[i];
  printf("La somma degli elementi dati e': %d\n",Tot);
```

```
}
```

Il primo ciclo `for` permette di incamerare 5 diversi dati. Si noti come l'istruzione sia unica ma resa *parametrica* dall'indice `i`. Allo stesso modo, il secondo ciclo `for` permette, in modo compatto, di effettuare la somma di tutti i dati raccolti.

## 6. Identificatori e Parole riservate

Il nome di un programma **C** deve rispettare le regole del sistema operativo su cui si lavora. Una sequenza di caratteri maiuscoli e minuscoli ed il carattere di sottolineatura "_" sono di solito sempre accettati. Spesso è richiesto il suffisso "`.c`".

Per quanto riguarda i nomi da attribuire alle variabili, alle costanti o alle varie funzioni che si vogliono utilizzare, la regola è che devono *iniziare* con una lettera (maiuscola e minuscola) e poi possono contenere lettere o numeri eventualmente inframezzati dal carattere di sottolineatura "_" o dal carattere "$". Tali nomi vengono comunemente chiamati *identificatori*. Un identificatore tuttavia non può coincidere con le *parole riservate* del linguaggio (non possiamo ad esempio definire la variabile `while`). Alleghiamo l'elenco delle parole riservate del **C++** (ANSI/ISO Standard Version 2.2):

| | | | | | | |
|---|---|---|---|---|---|---|
| asm | auto | break | case | catch | char | class |
| const | continue | default | delete | do | double | else |
| enum | extern | float | for | friend | goto | if |
| inlineint | long | new | operator | private | protected | public |
| register | return | short | signed | sizeof | static | struct |
| switch | template | this | throw | try | typedef | union |
| unsigned | virtual | void | volatile | wchar_t | while | |

## 7. Esercizi/esempi

Concludiamo questo Capitolo con alcuni esercizi in tutto o in parte risolti (l'autore delle note si rende conto che le nozioni fin qui introdotte non possono aver fornito padronanza del linguaggio).

Esercizio 1. *Il prossimo programma illustra un tipico utilizzo del ciclo* `for` *per* aggiornare *(un certo numero di volte) il valore di una variabile. Quale sarà il valore stampato?*

```
main(){
     int a,i,Tot;

     Tot = 0;
     printf("Dammi un numero: ");
     scanf("%d",&a);
     for(i = 1; i <= a; i++)
          Tot = Tot + 1;
     printf("Risultato: %d\n",Tot);
}
```

Esercizio 2. *Scrivere un programma* **C** *che accetta in input un numero positivo n e verifica se tale numero sia o meno un numero primo. Si utilizzi l'operatore % che restituisce il modulo della divisione intera, il ciclo* `for` *e il costrutto condizionale* `if (n % i == 0) ....`

ESERCIZIO 3. *Scrivere un programma* **C** *che accetta in input una sequenza di lunghezza* non nota a priori *di numeri interi positivi (terminata da uno* `0`*) e ne restituisce in output la somma e la media.*

*Il seguente codice risolve il problema per la somma:*

```
main() {
     int Tot,n;

     Tot = 0;
     printf("Dammi un numero naturale: (0 per uscire)  ");
     scanf("%d",&n);
     while( n > 0){
          Tot = Tot + n;
          printf("Dammi un naturale: (0 per uscire)  ");
          scanf("%d",&n);
     }
     printf("La somma e' %d\n",Tot);
}
```

ESERCIZIO 4. *Si scriva un programma utilizzante due cicli* `for` *annidati che accetta un valore* $n$ *in input e stampa una "*$\times$*" di lato* $n \times n$*. Una possibile soluzione è la seguente:*

```
main(){
      int i,j,n;

      printf("Dammi un numero: ");
      scanf("%d",&n);
      for(i = 0; i <= n; i = i++){
          for(j = 0; j <= n; j = j++)
               if (i == j || i == n - j)
                    printf(" ");
               else printf("*");
          printf("\n");
       }
}
```

*La doppia barra verticale nella condizione dell'istruzione di* `if ... else` *va letta come* **or** *logico.*

CAPITOLO 3

# Tipi

Per *tipo di dato* si intende un insieme di valori ed un insieme di operazioni che possono essere applicate ad essi. I tipi possono essere suddivisi come:

- tipi predefiniti (built-in),
- tipi definiti dall'utente;

oppure caratterizzati dalla loro complessità strutturale:

- tipi semplici,
- tipi strutturati.

In questo capitolo saranno presentati diverse possibilità offerte dal linguaggio **C** per definire ed utilizzare tipi di dati.

## 1. Tipi semplici built-in

Il linguaggio **C** mette a disposizione quattro famiglie di tipi semplici predefiniti:

- per i numeri interi: `(signed, unsigned) (short,long) int,`
- per i booleani: `bool`
- per i numeri reali: `float, (long) double`
- per i caratteri: `(signed,unsigned) char`.

Ad esser precisi, nello standard **C** non vi sono i booleani ma si possono definire come tipo enumerativo. Nel **C** indicato da queste note tuttavia il tipo esiste ed esiste anche in Java ed in Pascal, pertanto si è preferito inserirli esplicitamente. Come vedremo, le varie sottofamiglie permettono di suggerire al compilatore dettagli implementativi al fine di fornire la possibilità di ottimizzare la memoria utilizzata, di migliorare l'efficienza in esecuzione e, osservando ciò ad un più alto livello, di ingrandire la classe di valori ammissibili del tipo.

**1.1. Tipi interi.** I valori assumibili dalle variabili di tipo intero sono un sottoinsieme dell'insieme $\mathbb{Z} = \{0, 1, -1, 2, -2, \dots\}$. Saremo più precisi sui valori esatti per ciascuna delle famiglie di tipi interi permessi.

Per tutti sono permesse in **C** le seguenti operazioni aritmetiche:

| | |
|---|---|
| `+, -, *` | Somma, sottrazione e moltiplicazione |
| `/` | Quoziente della divisione intera |
| `%` | Resto della divisione intera |

Inoltre le variabili di tipo intero possono essere impiegate come argomenti degli operatori di

| | |
|---|---|
| `==, !==` | Simbolo relazionale di uguaglianza e disuguaglianza |
| `<, >, <=, >=` | Simboli per le relazioni d'ordine |

Le seguenti sono le 6 famiglie di interi permesse in **C**:

- `short unsigned int`;

- `short int` (eq. `short signed int`);
- `unsigned int`;
- `int` (eq. `signed int`);
- `long unsigned int`;
- `long int` (eq. `long signed int`).

Il prefisso `signed` (o equivalentemente l'assenza di prefisso) prevede una rappresentazione del numero in complemento a due. `unsigned` suggerisce l'uso di tutti i bit per la rappresentazione del valore assoluto del numero (che sarà dunque solo positivo). `short`, nulla o `long` suggeriscono quanti bytes vadano usati per rappresentare il numero. Spetterà poi al compilatore decidere se sia vantaggioso o meno sfruttare tale suggerimento. Mediante il programma

```
main() {
      short unsigned int sui;
      short int si;
      unsigned int ui;
      int i,count;
      long unsigned int lui;
      long int li;

      sui=1;si=1;ui=1;i=1;lui=1;li=1;
      for(count = 1; count < 32; count++){
            sui=sui*2;si=si*2;ui=ui*2;
            i=i*2;lui=lui*2;li=li*2;
            printf("1 %hud  2 %hd  3 %ud  4 %d  5 %lud  6 %ld \n",
                    sui,si,ui,i,lui,li);
      }
}
```

si vuole illustrare l'uso delle opportune direttive di stampa per il tipo intero e, mediante l'analisi dell'output, i valori ammissibili per ognuno dei dati tipi. In particolare, l'esperimento basato su JFE sul calcolatore di chi scrive ha evidenziato l'uso di 2Bytes per i tipi short e di 4 bytes per gli altri tipi. In particolare `int` e `long int` risultano implementati allo stesso modo.

NOTA 2. *Salvo particolari scopi, quali scrivere un codice che poi sarà utilizzato su dispositivi con poca memoria, si consiglia di utilizzare sempre il tipo* `int` *e la direttiva di stampa %d.*

**1.2. Il tipo Booleano.** Nelle più recenti versioni del **C** esiste il tipo Booleano `bool`. Qualora non vi sia, 'e sufficiente anteporre al programma in cui li si desidera utilizzare la riga:

```
typedef enum {false,true} bool;
```

Le variabili di tipo `bool` possono assumere solo due valori: `false` e `true` (o, equivalentemente 0 e 1).

Per il tipo `bool` sono permesse le operazioni, tipiche della logica booleana, ovvero

**and:** `&&`
**or:** `||`
**not:** `!`

Per stampare un booleano come 0 e 1 si usa la direttiva numerica `%d`. Per stampare invece `true` o `false` bisogna dirlo esplicitamente. Ad esempio, se `b` è una variabile Booleana:

```
if (b)
   printf("true");
else printf("false");
```

Si osservi l'uso diretto della variabile Booleana nella condizione dell'`if`. Poiché `false` e `true` sono sinonimi di 0 e 1 sono permesse (ma sconsigliate) anche operazioni aritmetiche tra Booleani. Il seguente programma vuole chiarire queste osservazioni:

```
main(){
    bool x;

/* Operazioni Booleane */
    x = false;
    printf("1. la var x vale %d\n",x);
    x = x || true;
    printf("2. la var x vale %d\n",x);
    x = !x;
    printf("3. la var x vale %d\n",x);
/* Operazioni Aritmetiche */
    x = 0;
    printf("4. la var x vale %d\n",x);
    x = x + 1;
    printf("5. la var x vale %d\n",x);
    x = x + 1;
    printf("6. la var x vale %d  (si noti che non e' 2) \n",x);
    x = -x;
    printf("7. la var x vale %d  (si noti che non e' -1) \n",x);
}
```

L'output risulta:

```
1. la var x vale 0
2. la var x vale 1
3. la var x vale 0
4. la var x vale 0
5. la var x vale 1
6. la var x vale 1  (si noti che non e' 2)
7. la var x vale 1  (si noti che non e' -1)
```

**1.3. Tipi reali.** Per utilizzare numeri *reali*[1] vi sono i tipi (built-in):

- `float`, e
- `(long) double`.

[1]Si ricorda che nel calcolatore non è possibile rappresentare numeri reali *irrazionali*, per i quali sarebbe necessario disporre di infinite cifre decimali (e dunque di memoria infinita), ma solo una loro approssimazione.

Similmente agli interi, la differenza tra questi tipi è nel numero di bytes destinati alla loro memorizzazione (e dunque alla loro *precisione*).

I numeri reali possono entrare nelle espressioni condizionali, inoltre le operazioni tra numeri reali (nelle varie possibilità) sono:

| | |
|---|---|
| `+, -, *` | Somma, sottrazione e moltiplicazione |
| / | Risultato (reale) della divisione |

Si osservi che non vi è più la possibilità di utilizzare il simbolo `%` (resto della divisione intera) che perde di significato nella divisione tra numeri reali. Inoltre il simbolo / ha cambiato il proprio significato, in quanto in questo contesto fornisce il valore esatto del rapporto tra i suoi argomenti.

Vi sono due possibilità per scrivere delle *costanti* reali in espressioni che andranno assegnate a variabili `float` e (`long`) `double`:

- mediante una rappresentazione estesa, come ad esempio:
  - `0.325` (o equivalentemente `.325`)
  - `-123.456`
- mediante la rappresentazione esponenziale:
  - `3.25 e -1`
  - `-1.2345 e 2`,

  ove tali numeri vanno intesi come $3.25 \times 10^{-1}$ e $-1.2345 \times 10^{2}$.

Il seguente programma mostra le direttive per la stampa di float e double, con eventuale scelta di cifre significative.

```
main(){
  float a;
  double b;

  a = 12345.67890123456789;
  b = a;
  printf("Stampo il valore di a per esteso %f \n",a);
  printf("e in notazione esponenziale a = %e\n", a);
  printf("Scelgo le cifre significative: (3) a = %.3f, (5) a = %.5f,\n",a,a);
  printf("(10) a = %.10f, (3-exp) a = %.3e \n", a, a);
  printf("Stampo un double (5) %.5f ", b );
}
```

Analizziamo con il seguente programma un altra caratteristica.

```
main(){
     float r;
     double d;
     int j;

     r = 1; d = 1;
     for (j = 1; j <= 12; j = j + 1){
         r  = r * 1e4; d  = d * 1e28;
         printf("%.20e \t %.20e \n", r , d);
     }
}
```

Il suo output è:

```
1.0000000000000000000e+004      9.9999999999999960000e+027
1.0000000000000000000e+008      9.9999999999999870000e+055
9.9999995904000000000e+011      9.9999999999999840000e+083
1.0000000275642240000e+016      9.9999999999999720000e+111
1.0000000204087730000e+020      9.9999999999999640000e+139
1.0000000138484279000e+024      9.9999999999999510000e+167
1.0000000622711310000e+028      9.9999999999999540000e+195
1.0000000331813535000e+032      9.9999999999999560000e+223
1.0000000409184788000e+036      9.9999999999999490000e+251
1.#INF0000000000000000e+000     9.9999999999999430000e+279
1.#INF0000000000000000e+000     9.9999999999999410000e+307
1.#INF0000000000000000e+000     1.#INF0000000000000000e+000
```

Osservando l'output si nota immediatamente l'imprecisione dei numeri reali, che emerge quando si richiede un certo numero di cifre significative. Partendo entrambe dal valore iniziale 1 ed essendo iterativamente moltiplicate per potenze di 10 le due variabili finiscono per assumere dei valori alquanto imprevedibili, già prima di superare i propri limiti massimi. Ciò significa che quando si utilizzano delle variabili `float` o `double` un test di uguaglianza tra variabili (ad esempio nelle condizioni dell'`if`) non ha spesso il significato voluto. Il sotterfugio comunemente utilizzato è quello di sostituire un test `u == v` con un test della forma: `(u <= v * 1.0001) && (v <= u * 1.0001)`.

NOTA 3. *Non abbiamo parlato dei* `long double`. *Il lettore interessato è invitato ad approfondire autonomamente le proprie conoscenze.*

**1.4. Il tipo carattere.** Il linguaggio **C** dispone di un tipo di dato built-in relativo ad un singolo carattere. Più precisamente viene data all'utente la possibilità di scegliere il tipo di codifica binaria del carattere:

| | |
|---|---|
| `signed char` | -128–127 |
| `(unsigned) char` | 0–255 |

In entrambi i casi, i valori dallo 0 al 127 sono riservati alla codifica ASCII dei caratteri. Similmente ai numeri, anche per i caratteri si possono utilizzare delle *costanti*. Esse vanno messe tra apici (singoli). Ad esempio, i seguenti sono assegnamenti leciti:

```
c = ' ';
c = '\n';
c = 'a';
```

Nel primo assegnamo il carattere *spazietto* nel secondo il carattere associato al comando di *nuova linea* (simile a quello che si ottiene digitando il tasto di invio) nel terzo il carattere 'a'.

Il programma seguente:

```
main(){
     char c;

     printf("Dammi un carattere: ");
     scanf("%c",&c);
     printf("Il carattere e' %c \n", c);
}
```

| | |
|---|---|
| hd | numero decimale (short) |
| d | numero decimale (int) |
| ld | numero decimale (long int) |
| f | numero reale (virgola mobile) per esteso |
| e | numero reale (virgola mobile) notaz. esponenziale |
| c | singolo carattere |
| s | stringa (si veda la Sezione 3.1.1). |
| u | numero decimale senza segno (positivo) |
| o | numero ottale senza segno |
| x | numero esadecimale senza segno |

FIGURA 1. Direttive di conversione

accetta in input un carattere e lo stampa in output. Con la seguente modifica, invece, il programma permette di stampare il carattere successivo (nel senso della codifica ASCII):

```
c = c + 1;
printf("Il carattere successivo e' %c \n", c);
```

Mentre con la modifica:

```
printf("Il suo valore ASCII e' %d \n", c);
```

è possibile conoscere il valore numerico (ASCII) per il carattere in questione.

Alcuni caratteri particolari sono i seguenti:

$\backslash n$ linea nuova
$\backslash t$ tab
$\backslash b$ spazio indietro
$\backslash r$ tasto 'return'

ESERCIZIO 5. *Si scriva un programma* **C** *che stampi la tabella dei valori ASCII. Si estenda tale programma per verificare cosa succeda quando si desidera stampare un carattere a cui sia associato un valore maggiore a 127.*

Si noti che i caratteri sono visti come numeri interi (con la loro codifica ASCII). Le operazioni ammesse per loro sono pertanto le stesse degli interi.

Come illustrato nell'Introduzione, nelle funzioni di libreria di input-output è possibile fornire delle direttive di conversione per interpretare correttamente un tipo di dato. Nella Figura 1 vi sono le varie possibilità. Per maggiori informazioni si invita a consultare un manuale di **C**, eventualmente il manuale in linea di JFE.

**1.5. Esercizi su rappresentazione e compatibilità di tipi built-in.** I tipi semplici built-in presentati costituiscono una gerarchia di tipi. E' infatti possibile, ad esempio, assegnare ad una variabile reale un valore intero, ma non viceversa. La gerarchia (relativa ai tipi più usati) è la seguente:

$$\texttt{char} < \texttt{int} < \texttt{long} < \texttt{float} < \texttt{double}$$

In generale una variabile di un dato tipo può assumere valori di un tipo più a sinistra, ma non viceversa. Tuttavia, non tutti i compilatori segnalano situazioni di errore (anche se spesso sono messe a disposizioni dell'utente delle opzioni affinchè il compilatore segnali situazioni indesiderate); deve essere cura del programmatore utilizzare in modo opportuno i tipi. Si consideri il seguente semplice programma

```
    main(){
       float f;
       int i;

       f = 10000000000;
       i = f;
       printf("i = %d \n", i);

       i = 1;
       f = 3*(i/3);
       printf("f = %.5f ",f);
}
```

Ci si attenderebbe il seguente output

```
i = 10000000000
f = 1.00000
```

Tuttavia, in modo inatteso la prima riga risulta:

```
i = -2147483648
```

E' successo che il valore $10^{10}$ supera il massimo valore positivo per variabili di tipo `int`. Durante la compilazione sono segnalati dei "Warning" (ovvero dei punti di possibile errore). Ma è solo durante l'esecuzione che l'errore emerge.

Molto più inattesa è la seconda riga di output:

```
f = 0.000000
```

Qui è successo qualcosa di ancora più subdolo. La divisione tra `i`, variabile `int` e `3`, una costante che viene assegnata al tipo più piccolo che la contiene (`int` o suoi sottotipi) è una divisione intera, che restituisce un intero ovvero 0. Anche se moltiplicato poi per 3, sempre 0 resta. Questo è un problema tipico del **C** e si risolve utilizzando il cosiddetto operatore di casting. Se `n` è un intero (costante o variabile intera), (`float`)`n` è lo stesso valore, ma rappresentato come `float`. Sostituendo l'assegnamento sopra con

$$\texttt{f} = 3 * ((\texttt{float})\texttt{i}/3)$$

si ottiene in stampa l'atteso valore 1.0000. Ogni tipo ha il proprio *operatore di casting* che permette di convertire i valori nel formato desiderato.

## 2. Tipi semplici definiti dall'utente

In **C** è permesso definire nuovi tipi semplici. In questo paragrafo si vuol descrivere brevemente questa possibilità che permette di programmare ad un più alto livello di astrazione.

**2.1. Ridefinizione.** E' possibile assegnare un nuovo *nome* ad un tipo preesistente. La sintassi della dichiarazione volta a ciò è:

`typedef` Tipo esistente Nuovo Tipo `;`

Come esempio si potrebbe pensare di italianizzare i nomi dei tipi:

```
typedef int interi;
typedef float reali;
typedef char caratteri;
```

A questo punto è possibile usare i nomi introdotti per dichiarare le variabili:

```
interi i,j,k;
reali r;
caratteri c;
```

**2.2. Enumerazione.** I tipi enumerativi permetono di descrivere ad un maggior grado di astrazione insiemi di oggetti ben caratterizzati. La sintassi è:

`typedef enum` { elenco } nuovo nome ;

Ad esempio:

```
typedef enum { lun, mar, mer, gio, ven, sab, dom } giorno_settimana ;
```

E' poi possibile dichiarare una variabile `g`

```
giorno_settimana g;
```

che potrà assumere come valori ammissibili solo `lun`, ..., `dom`.

Il compilatore associa internamente un numero progressivo (intero da 0 in poi) a tali oggetti. Ciò permette di utilizzare per questi tipi le operazioni proprie dei numeri interi.

## 3. Tipi strutturati

In **C** si possono definire tipi più complessi a partire da altri più semplici. Sono questi i tipi *strutturati*. In questo capitolo analizzeremo la possibilità di struttura offerta da `array`, `struct` e puntatore.

**3.1. Vettori.** I vettori sono stati introdotti informalmente nella Sezione 5. Se una variabile `f` viene definita come segue:

```
int f[20];
```

essa permette di manipolare in modo compatto 20 dati interi distinti: fornisce dunque una *struttura* a tali dati. Dal punto di vista prettamente matematico, `f` rappresenta una funzione dall'insieme $\{0, \ldots, 19\}$ all'insieme `int` dei numeri interi rappresentabili nel linguaggio: $\texttt{f} : \{0, \ldots, 19\} \longrightarrow \texttt{int}$. Per tale ragione i vettori sono detti mappe finite.

Con la tecnica di ridefinizione (Sez. 2.1), risulta possibile utilizzare un nome diverso per tale tipo. Ad esempio:

```
typedef int vettore[20];
vettore f;
```

Si noti che, come accade spesso in informatica, si inizia a contare dallo 0. L'assegnamento `u = v;` per due vettori non è ammesso, nemmeno se essi sono dello stesso tipo. Bisogna copiare elemento per elemento.

Per ragioni di leggibilità e di semplicità nella stesura del codice, è talvolta conveniente dichiarare un vettore avente un elemento di più, come emerge dal seguente esempio.

Supponiamo di voler memorizzare la temperatura a mezzogiorno nella cuccia del dobermann di un nostro amico per tutti i giorni del prossimo anno. Definiremo allora una variabile di tipo vettore:

```
float temp_1998[366];
```

che si può immaginare come una tabella monodimensionale:

| 0 | 1 | ... | 365 |
|---|---|---|---|
| | | | |

In tal modo potremo assegnare a `temp_1998[1]` il valore del primo gennaio, a `temp_1998[2]` quello del 2 di gennaio e così via. Sarebbe possibile assegnare un valore anche a `temp_1998[0]`. Tuttavia, tale valore va al di fuori dei nostri scopi. Alternativamente si sarebbe potuto dichiarare

```
float temp_1998[365];
```

ed iniziare ad assegnare i valori con il giorno `0`, anche se probabilmente è meno intuitivo usare i numeri in tal modo.

In entrambi i casi, tuttavia, è abbastanza difficile usare il programma quando ci si allontana dai primi giorni dell'anno. Ad esempio, per associare al 28 luglio il giorno 240 è necessario farsi un bel po' di calcoli a parte. Sarebbe estremamente più semplice poter identificare un giorno (come si fa usualmente) mediante giorno del mese e mese, ovvero mediante due valori. In **C** ciò è (facilmente) possibile mediante i vettori bidimensionali, altresì detti *matrici*. Possiamo dichiarare una nuova variabile nel seguente modo:

```
float temp_matr_1998[32][13];
```

La variabile `temp_matr_1998` si può vedere come una tabella:

| | 0 | 1 | ... | 31 |
|---|---|---|---|---|
| 0 | | | | |
| 1<br>⋮ | | | | |
| 12 | | | | |

In realtà molte celle non saranno utilizzate (oltre ai giorni 0, al mese 0, i vari 29, 30, 31 febbraio, il 31 aprile e così via). Si capisce però qui l'importanza di partire da 1 per semplicità! Il 28 luglio è ora accessibile semplicemente come: `temp_matr_1998[28][7]`.

Similmente, potremmo aver bisogno di più indici per accedere ad un dato. Ad esempio memorizzare una temperatura all'ora, per ogni giorno, per ogni mese e ogni anno. La dichiarazione potrebbe essere del tipo:

```
float temp[24][32][13][2100].
```

Vogliamo concludere questa sezione illustrando un modo rapido per inizializzare variabili di tipo vettore o matrice. Questo può essere di enorme comodità per testare programmi senza dover immettere i dati di volta in volta. L'idea è di assegnare direttamente i valori nel momento della dichiarazione. Ad esempio:

```
int matrice [6][6] =
       {{0,1,1,1,0,0},
        {0,0,1,1,1,0},
        {0,0,0,0,0,1},
        {1,1,0,0,1,0},
        {0,0,0,0,0,1},
        {0,0,0,0,0,1}};
bool vettore [6] =
```

```
        {0,1,1,0,1,0};
```

Si avrà, ad esempio, `matrice[0][0] = 0, matrice[0][1] = 0, matrice[1][0] = 0,matrice[1][1] = 0, vettore[0] = 0, vettore[1] = 1`.

Si osservi che se `u` e `v` sono vettori (o matrici) dello stesso tipo, non è ammesso l'assegnamento `u = v`; Qualora si voglia *copiare* un vettore in un altro, va fatto elemento per elemento, dentro un ciclo `for`.

ESERCIZIO 6. *Si scriva un programma in grado di accettare un certo numero di* temperature*, di memorizzarle in un vettore bi-dimensionale e di stampare la media tra le temperature inserite.*

3.1.1. *Stringhe.* Un importante caso particolare dei vettori sono le stringhe. Una stringa è un vettore di caratteri; il compilatore **C** permette di utilizzare le stringhe per l'input-output di testi.

Il seguente programma accetta una sequenza di al più 20 caratteri finchè non viene battuto il tasto di ritorno a capo. Si vuole poi mostrare i due modi per rendere visibile la variabile `s`: un carattere alla volta o come un'unica stringa.

```
main(){
   char s[20];
   int i;

   scanf("%s",s);
   for (i = 0; i < 20; i++)
       printf("s( %d ) = %c \n",i,s[i]);
   printf("%s \n",s);
}
```

Se `s` è una variabile di tipo stringa, l'assegnamento ad essa di una stringa avviene con la funzione `strcpy(s,--stringa--)`. Per i dettagli sull'utilizzo di questa e altre funzioni su stringhe si consiglia di consultare un manuale del linguaggio **C**.

ESERCIZIO 7. *Si scriva un programma in grado di accettare in input una stringa di al più 40 elementi e che fornisce in output la stringa ottenuta sostituendo le lettere minuscole con le corrispondenti maiuscole.*

**3.2. Strutture (record).** Un'altra possibilità offerta dal **C** per *strutturare* i dati è quella offerta dal costrutto `struct`. Il tipo ottenuto (chiamato *record* in altri linguaggi) è costituito da diversi *campi*. La definizione ha la forma:

```
typedef struct{
               tipo campo 1;
               ...
               tipo campo n;
             }
        nometipo;
```

Un esempio può essere quello della memorizzazione di una data:

```
main(){
   typedef struct{
              int giorno;
              int mese;
              int anno;
```

```
            } data;
        data d;

        printf("Dammi il giorno: ");
        scanf("%d",&d.giorno);
        printf("Dammi il mese: ");
        scanf("%d",&d.mese);
        printf("Dammi l'anno: ");
        scanf("%d",&d.anno);
    }
```

Il programma acquisisce i tre parametri. Similmente ai vettori, vi è un modo comune di accedere alle varie informazioni (campi). Diversa è la tecnica di accesso, non più mediante indici, bensì mediante i nomi dei campi.

ESERCIZIO 8. *Si scriva un programma* **C** *che definisca il tipo* `complex` *per i numeri complessi, ne accetti due in input e ne restituisca somma e prodotto.*

ESERCIZIO 9. *Si scriva un programma* **C** *in grado di accettare in input un vettore di 10 elementi ciascuno dei quali destinato a memorizzare un dato contente le informazioni* `cognome`*,* `nome` *ed* `anni` *di un dato individuo e che stampa infine in output tale vettore in ordine inverso da quello di inserimento.*

**3.3. Puntatori.** Una delle caratteristiche distintive del **C** rispetto ad altri linguaggi di programmazione strutturati è la gestione dei puntatori. Mediante i puntatori si possono definire le strutture dati fondamentali per la programmazione quali liste, code, pile e alberi. Il concetto di puntatore però può essere compreso anche in maniere isolata e indipendente da tale utilizzo.

Supponiamo di definire una variabile intera `a` e di assegnarle il valore `1`. La dichiarazione `int a` suggerisce al compilatore di assegnarle una certa area di memoria a partire da un certo indirizzo, supponiamo sia 100. Nella fase di assegnamento in tale area viene messa l'opportuna codifica del valore `1`.

Il comando `&` (puntatore) di **C** permette di conoscere l'indirizzo di memoria in cui una variabile viene memorizzata. Ad esempio, nel caso sopra `&a` avrebbe valore 100. Nel programma seguente:

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

    a = 1; b = 2;
    printf("La variabile a ha valore %d ed e' memorizzata
         nell'indirizzo  %u.\n",   a , &a);
    printf("La variabile b ha valore %d ed e' memorizzata
         nell'indirizzo  %u.\n",   b , &b);
}
```

L'output di tale programma (sul calcolatore di chi scrive) è il seguente:

```
La variabile a ha valore 1 ed e' memorizzata
         nell'indirizzo  39058916.
La variabile b ha valore 2 ed e' memorizzata
         nell'indirizzo  39058912.
```

Da cui si evince in particolare che il compilatore in questione alloca 4 parole di memoria per ogni variabile intera. Il valore restituito da `&a` è dunque un intero senza segno e può essere assegnato a variabili di quel tipo.

Nella fase dichiarativa del programma si possono definire variabili di tipo puntatore nel seguente modo:

```
<nome di tipo> *<nome di variabile>;
```

Tale dichiarazione dice che la variabile `nome di variabile` sarà destinata a contenere l'indirizzo di locazioni di memoria destinate alla memorizzazione di valori di tipo `tipo`. Tale dichiarazione ci permette di realizzare le strutture dati dinamiche.

Lo stesso operatore `*` può essere usato nel programma ma con significato diverso (come operatore di *indirezione*):

```
*(& a) = a
```

ovvero `*` applicato ad un numero intero positivo restituisce il contenuto di una locazione di memoria usata come suo argomento.

```
main(){
     int b;
     float a;
     double d;

     a = 1; b = 2; d = 3;
     printf("La variabile float a ha valore %f ed
              e' memorizzata nell'indirizzo %u.\n", *(&a), &a);
     printf("La variabile intera b ha valore %d ed
              e' memorizzata nell'indirizzo %u.\n", *(&b), &b);
     printf("La variabile double d ha valore %f ed
              e' memorizzata nell'indirizzo %u.\n", *(&d), &d);
}
```

Output:

```
La variabile float a ha valore 1.000000 ed
                  e' memorizzata nell'indirizzo 39058912.
La variabile intera b ha valore 2 ed
                  e' memorizzata nell'indirizzo 39058916.
La variabile double d ha valore 3.000000 ed
                  e' memorizzata nell'indirizzo 39058904.
```

CAPITOLO 4

# Funzioni

Un programma **C** è una particolare *funzione* dal nome `main`. In generale una funzione è un *blocco di codice* autocontenuto in grado di svolgere un particolare compito, ovvero in grado di fornire deterministicamente dei valori di output (o di modificare valori di dati memorizzati in variabili o files esterni) a seconda dei valori di input.

Sebbene, in linea di principio, sia possibile scrivere qualunque programma senza ausilio di altre funzioni, in programmazione è molto comodo avere la possibilità di *assegnare* un nome ad un blocco di codice con un particolare scopo e di *richiamare* tale blocco semplicemente usando il nome assegnatogli.

## 1. Sintassi

Iniziamo col dare una definizione sufficientemente rigorosa della sintassi di un programma **C**:

Programma ::=
⟨ Dich. globali ⟩ (*opt.*)
⟨ funzione `main` ⟩
⟨ Altre funzioni ⟩ (*opt.*)

La sintassi di una funzione generica è la seguente:

⟨ Tipo funzione⟩⟨ nome funzione ⟩(⟨ lista parametri ⟩)
⟨ Blocco ⟩

Quella di `main` è semplicemente:

`main`()
⟨ Blocco ⟩

La sintassi per il Blocco (per ora parziale) è la seguente:

{
⟨ Dichiarazioni locali⟩
⟨ Corpo della funzione⟩
}

Il *tipo* della funzione indica il tipo del valore che ci si aspetta come output della funzione stessa. Ad esempio la funzione che calcola la radice quadrata avrà come tipo `float` oppure `double`. I tipi possibili sono tutti i tipi semplici (built-in e enumerazioni) nonchè i puntatori. Saremo più precisi nella sezione 3. Inoltre `void` è un particolare tipo che viene utilizzato quando non ci si aspetta output dalla funzione. Questo particolare tipo di funzione viene anche detto *procedura*.

La *lista dei parametri* (vuota per `main`) verrà discussa in dettaglio nella Sezione 2. Le *variabili locali e globali* saranno discusse in dettaglio nella Sezione 4.

Il Corpo della funzione consiste informalmente di una sequenza di istruzioni o di un blocco. Definendo bene cosa sia una istruzione basta definire il Corpo come segue:

$$\text{Corpo della Funzione} \quad ::= \quad \langle \text{ Istruzione } \rangle$$

Mentre la sintassi per l'istruzione è la seguente

Istruzione ::=
⟨ Istruzione ⟩⟨ Istruzione ⟩|
⟨ Blocco ⟩|
⟨ Var ⟩ = ⟨ Expr ⟩; |
`if`⟨ Expr ⟩⟨ Istruzione ⟩|
`if`⟨ Expr ⟩⟨ Istruzione ⟩`else`⟨ Istruzione ⟩|
`for`(⟨ Var ⟩ = ⟨ Expr ⟩; ⟨ Expr ⟩; ⟨ Var ⟩ = ⟨ Expr ⟩)⟨ Istruzione ⟩|
`while`⟨ Expr ⟩`do`⟨ Istruzione ⟩|
...

Si noti che alla fine di ogni istruzione di assegnamento va messo il ';'.

NOTA 4. *Erroneamente si è portati a pensare che il ';' sia necessario a separare due istruzioni consecutive. La grammatica sopra non dice questo. Il seguente codice è infatti sintatticamente corretto anche se tra le due istruzioni* `while` *non vi sono ';'.*

```
main(){
    int a;

    a = 5;
    while (a > 0) {
        a = a - 1;
    }
    while (a < 5) {
        a = a + 1;
    }
}
```

Qualora il tipo della funzione non sia `void`, alla fine del corpo della funzione vi deve essere l'istruzione

```
return <espressione>;
```

(ove il tipo dell'espressione deve essere il tipo della funzione) che indica quale deve essere il valore restituito dalla funzione stessa come risultato.

NOTA 5. *Implicitamente il tipo di* `main` *è un intero. Pertanto sarebbe corretto (anche se privo di utilità nei programmi presentati in queste note) ultimare il programma con una istruzione del tipo* `return n`*, con* `n` *numero intero (si veda anche la nota 1).*

Una funzione può *chiamare* al suo interno qualunque funzione, anche sé stessa. Se chiama un'altra funzione, questa deve essere prima dichiarata (come tipo e tipo dei parametri) nella fase di dichiarazione.

Una funzione può anche chiamare sè stessa (ricorsione diretta). In tal caso non serve autodichiararsi. Un esempio estremo di ricorsione è costituito dal seguente programma:

```
main(){
    main();
}
```

Durante l'esecuzione il programma richiama sé stesso infinitamente (bisogna interrompere forzatamente l'esecuzione). Nel Capitolo 5 impareremo a sfruttare alcune delle enormi potenzialità fornite dalla ricorsione.

Analizziamo il seguente esempio:

```
main(){
   /* Dichiarazione */
   void primo(), secondo();
   /* Corpo */
   printf("Sono in main\n");
   primo();
   printf("Sono di nuovo in main\n");
   secondo();
   printf("Sono di nuovo in main\n");
}
/* Funzioni */
void primo(){
   printf("Sono in primo\n");
}
void secondo(){
   void primo;
   /* Corpo secondo */
   printf("Sono in secondo\n");
   primo();
   printf("Sono di nuovo in secondo\n");
}
```

L'output prodotto illustra come il flusso d'esecuzione sia passato tra le varie funzioni:

```
Sono in main
Sono in primo
Sono di nuovo in main
Sono in secondo
Sono in primo
Sono di nuovo in secondo
Sono di nuovo in main
```

## 2. Parametri

Ad una funzione è comodo, naturale ed a volte necessario passare degli argomenti, dei parametri. Ad esempio, utilizzando la funzione `sqrt` che calcola la radice quadrata, è indispensabile fornire come argomento il valore di cui vogliamo conoscere la radice quadrata. Chiameremo infatti tale funzione con l'istruzione:

```
sqrt(argomento);
```

**2.1. Parametri formali e parametri attuali.** Quando si definisce una funzione si deve definire con precisione l'elenco dei suoi parametri. Ad esempio:

```
float sigma(int x , float y , int [] z)
```

Le variabili `x`, `y` e `z` costituiscono la lista dei *parametri formali* della funzione `sigma`.

Supponiamo ora che nel programma principale (funzione `main`) vi sia la chiamata

```
sigma( a * 2, b , v)
```

Le tre *espressioni* `a * 2`, `b` e `v` costituiscono i tre *parametri attuali* della chiamata di funzione. All'atto della chiamata i tre parametri attuali vengono *valutati*. In quel momento vi è la comunicazione tra i parametri attuali e quelli formali. Questa comunicazione (passaggio dei parametri) avviene in generale con due modalità.

**2.2. Passaggio per valore e per indirizzo.** Quando si utilizzano funzioni di un certo tipo (per esempio funzioni matematiche quali `sqrt`) è preferibile che l'esecuzione della funzione non abbia effetto sulle variabili occorrenti nei parametri attuali. In altri termini che la funzione restituisca un risultato senza andare ad alterare valori presenti nel programma/funzione chiamante.

In altri casi (ad esempio la funzione di lettura di un valore `scanf` oppure funzioni per leggere/modificare dei vettori) non si richiede alla funzione un risultato (o almeno non solo quello) ma di modificare una o più variabili che passiamo come parametri attuali. Ancora in altri casi, la dimensione dei dati passati è tale da rendere poco efficiente sia in termini di tempo la copiatura dei valori dei parametri attuali in nuove variabili. In questi ultimi due casi è naturale che parametri formali ed attuali condividano l'indirizzo in cui l'informazione viene unicamente memorizzata.

In **C** il passaggio di parametri è fondamentalmente per *valore*. Tuttavia è facile, e vedremo come, simulare il passaggio per *indirizzo*.

Riferendoci all'esempio della sottosezione precedente, nel momento della chiamata viene calcolato il valore dei primi due parametri attuali, ovvero delle espressioni `a * 2` e `b`. Tale *valore* viene dunque assegnato ai parametri formali corrispondenti della funzione, ovvero ai parametri formali `x` e `y`. L'esecuzione della funzione `sigma` non può modificare il valore delle variabili presenti nei parametri attuali (`a` e `b` nel nostro esempio). Inoltre, al termine dell'esecuzione della funzione `sigma` i valori dei parametri formali `x` e `y` vengono perduti.

Diversa è la situazione del terzo parametro. Il parametro attuale è la variabile `v`. Quello formale la variabile `z`. Entrambe le variabili sono dei vettori. All'atto del passaggio dei parametri la variabile `z` assume il valore dell'indirizzo dovre il vettore `v` è memorizzato. In altri termini è copiato l'*indirizzo* del vettore. In questo caso il passaggio parametri avviene per *indirizzo*. In questo caso (nel caso dei vettori) ciò accade implicitamente in quanto la variabile che memorizza un vettore in realtà memorizza semplicemente la locazione di inizio del vettore. Ciò consente ad esempio di scrivere delle funzioni di input-output di vettori, al fine di trasformare il programma:

```
main(){
    int i, v[10];
    for (i = 0; i < 5; i++){
```

```
        printf("Elemento %d -esimo: ",i);
        scanf("%d", &v[i]);
    }
    for (i = 0; i < 5; i++)
        printf("Elemento %d -esimo: %d \n",i,w[i]);
}
```

nel programma equivalente:

```
main(){
/* Dichiarazione */
    void leggi_vettore( int[] ),
    scrivi_vettore( int[] );
    int v[10];
/* Corpo */
    leggi_vettore( v , 5);
    scrivi_vettore( v , 5);
}
/* Funzione di lettura */
void leggi_vettore( int w[], int n ){
    int i;
    for (i = 0; i < n; i++){
        printf("Elemento %d -esimo: ",i);
        scanf("%d", &w[i]);
   }
}
/* Funzione di scrittura */
void scrivi_vettore( int w[], int n ){
    int i;
    for (i = 0; i < n; i++)
        printf("Elemento %d -esimo: %d \n",i,w[i]);
}
```

In questo esempio vi sono dunque tre funzioni: quella principale (`main`), e le due funzioni di lettura e scrittura di un vettore. Si osservi come tra i parametri sia presente anche un parametro che dice a quale porzione del vettore siamo interessati.

Per mezzo del costrutto di puntatore `&` (si veda il Capitolo 3, Sezione 3.3) il passaggio di parametri può essere esplicitamente per indirizzo. Ciò si ottiene passando ai parametri formali l'indirizzo di una variabile usata come parametro attuale. In questo modo all'interno della funzione si può modificare il contenuto di tale indirizzo, modificando, di fatto, il valore di una variabile `a` occorrente come `&a` nel parametro formale.

Come esempio, l'esecuzione del programma seguente:

```
main(){
    int valore(int), indirizzo(int);
    int a;

    a = 1;
    printf("Risultato valore( %d ) = %d \n", a, valore(a));
    printf("a vale: %d \n",a);
```

```
    printf("Risultato indirizzo( %d ) = %d \n", a, indirizzo(&a));
    printf("a vale: %d \n",a);
}

int valore(int x){
    x = 2 + x;
    return x;
}

int indirizzo(int *x){
   *x = 2 + *x;
    return *x;
}
```

produce come output

```
Risultato valore( 1 ) = 3
a vale: 1
Risultato indirizzo( 2 ) = 3
a vale: 3
```

### 3. Il tipo di una funzione

Una funzione ha associato un suo tipo (di output). Tale tipo può essere

- `void` (in tal caso le funzioni sono dette procedure)
- un tipo semplice built-in, ovvero `int` con tutte le sue varianti, `float`, `double`, ma anche `char` e `bool`.
- un puntatore.

Quest'ultima possibilità viene sfruttata per restitituire come output un vettore. I vettori in **C** sono infatti dei puntatori all'inizio del vettore stesso. Per far ciò tuttavia è bene definire il vettore che sarà restituito come `static`. `static` è una direttiva che dice di allocare la stessa memoria per tale variabile ogni volta che la funzione viene richiamata. Il seguente esempio illustra un utilizzo del puntatore per restituire un vettore.

```
#include <stdio.h>
#define DIM 5

main(){
    int * vettore();
    int i;
    for(i=0;i<DIM;i++)
       printf("Risultato valore( %d ) = %d \n", i, vettore()[i]);
}

int * vettore(){
   static int v [DIM];
   int i;
   for(i=0;i<DIM;i++)
      v[i] = DIM-i;
   return v;
```

```
}
```

## 4. Variabili locali e globali

Si consideri il seguente esempio:

```
main(){
     void proc();
     int a;
     a = 5;
     proc();
     printf("a vale: %d \n", a);
}
void proc(){
     int a;
     a = 6;
}
```

La sua esecuzione produce come output `a vale: 5`. L'assegnamento `a = 6` presente nella funzione `proc()` ha l'effetto di assegnare il valore 6 ad una variabile *locale* alla funzione stessa, dichiarata all'inizio della funzione. Tale assegnamento non ha nessun effetto sulla variabile (in questo caso omonima) `a` dichiarata all'inizio del programma principale.

In **C** è possibile poi dichiarare variabili comuni a tutte le funzioni (compresa `main`). Tali variabili sono dette globali.

```
/* Variabili globali */
int a;

/* funzione main */
main(){
    void proc();
    a = 5;
    proc();
    printf("a vale: %d \n", a);
}
/* funzione proc */
void proc(){
    a = 6;
}
```

In questo caso l'output del programma diventa: `a vale 6`. Mancando una dichiarazione di `a` all'interno di `proc` si va a vedere al livello più esterno (in questo caso ancora sopra a `main`) di trovare una variabile di tale nome. Proprio quelle sarà modificata.

In questo paragrafo si è voluta dare solo un'idea di uso di variabili locali globali e di regole di visibilità (scope). Vista l'enorme importanza dell'argomento, si consiglia, una volta impratichitisi con il linguaggio, di leggere qualche testo avanzato per approfondire questi concetti.

## 5. Le istruzioni `switch` e `do while`

Concludiamo questo capitolo introducendo due ulteriori istruzioni. Spieghiamo informalmente la sintassi del costrutto `switch` con il seguente esempio. Supponiamo di aver letto un carattere `C`:

```
switch (C){
  case 'A': case 'E': case 'I':
       printf("Il carattere e' una delle tre prime vocali");
           break;
  case 'O': case 'U':
       printf("Il carattere e' una delle ultime due vocali");
           break;
  case 'B': case 'C': case 'D':
       printf("Il carattere e' una delle tre prime consonanti");
           break;
  default:
       printf("Il carattere e' una delle ultime 13 consonanti");
}
```

Il significato è evidente, così come è evidente che tale blocco potrebbe essere rimpiazzato da una opportuna sequenza di `if ... else`. La linea `default` può essere omessa, qualora non necessaria.

Tale costrutto può essere d'ausilio, ad esempio, alle fasi di input-output dei dati di tipo enumerativo (cf. Sezione 2.2).

La sintassi dell'istruzione `do-while` è la seguente:

`do`
⟨ istruz ⟩
`while` ⟨ condizione ⟩

Se `I` è l'istruzione e $\alpha$ la condizione, la sua semantica è del tutto equivalente al frammento di programma:

`I`;
`while` $\alpha$
`I`;

Sebbene palesemente inessenziale, tale costrutto permette di scrivere un ciclo in cui l'istruzione deve essere eseguita almeno una volta in modo più compatto rispetto al `while`.

CAPITOLO 5

# La ricorsione

## 1. Motivazioni

Supponiamo di voler scrivere una funzione che restituisca il fattoriale di un numero intero `n` passatole come parametro (per semplicità assumiamo che `n` sia positivo). La funzione potrebbe essere la seguente:

```
int fatt(int n){
   int i;

   for (i = 1; i < n; i++)
      n = n * i;
   return n;
}
```

Tale definizione, se pur semplice, non è la più naturale possibile. Per comprenderla, è necessario conoscere concetti quali variabili locali, sintassi e semantica del ciclo `for`, aggiornamento di variabili.

La definizione che viene data in matematica e che non richiede la conoscenza di nessun concetto oltre alla moltiplicazione ed alla sottrazione, è la seguente:

$$\texttt{fatt}(n) = \begin{cases} 1 & \text{se } n = 1 \\ n * \texttt{fatt}(n-1) & \text{se } n > 1 \end{cases}$$

Tale definizione viene detta *ricorsiva*, in quanto per definire `fatt` calcolata su un dato valore $n$ ci serve di conoscere il calcolo di `fatt` sul valore $n - 1$. In matematica, fisica, biologia e affini, ed in programmazione vi sono un gran numero di concetti che per essere descritti in modo opportuno necessitano di definizioni ricorsive.

Il **C** dà la possibilità di definire funzioni e procedure seguendo questa tecnica. La funzione che calcola il fattoriale può pertanto essere descritta nel seguente modo:

```
int fatt( int n ){
   if (n <= 1)
      return 1;
   else return n * fatt( n - 1 );
}
```

che rispecchia appieno la definizione ricorsiva. L'esecuzione del programma seguente non cambierà se richiamiamo la prima o la seconda delle definizioni sopra.

```
main(){
   int n, fatt();

   printf("Dammi un numero: ");
   scanf("%d",&n);
```

```
   printf("Il suo fattoriale e' %d \n", fatt( n ));
}
```

## 2. Esempi

**2.1. Pari e dispari.** Si vuole stabilire, utilizzando funzioni ricorsive, se un dato intero non negativo $n$ sia pari o dispari. Si può usare la seguente definizione mutuamente ricorsiva (si assuma che il tipo boolean sia dichiarato globalmente, così come le due funzioni `pari` e `dispari`):

```
boolean pari (int n){
    if (n == 0)
        return true;
    else return dispari( n - 1 );
}
boolean dispari (int n){
    if (n == 0)
        return false;
    else return pari( n - 1 );
}
```

Si segua a mano la computazione di `pari` o `dispari` per qualche valore di $n$.

**2.2. Fibonacci.** Leonardo Pisano (1180–1250; figlio di Bonaccio, da cui il soprannome *Fibonacci*) nel suo *Liber Abaci* propone, tra le altre cose il seguente problema:

> *Quante coppie di conigli verranno prodotte in un anno, a partire da un'unica coppia, se ogni mese ciascuna coppia dà alla luce una nuova coppia che diventa produttiva a partire dal secondo mese?*

Si osserva che tale problema dà luogo alla successione: 1,1,2,3,5,8,13,21,..., dalla quale si evince la seguente definizione ricorsiva:

$$\texttt{fibo}(n) = \begin{cases} 1 & \text{se } n = 0 \text{ oppure } n = 1 \\ \texttt{fibo}(n-1) + \texttt{fibo}(n-2) & \text{se } n > 1 \end{cases}$$

ESERCIZIO 10. *Si scriva una funzione* **C** *in grado di restituire il numero* $\texttt{fibo}(n)$.[1] *Si cerchi quindi di scoprire il numero di chiamate ricorsive necessarie a calcolare tale numero (in funzione di* $n$*). Si studi inoltre il rapporto di crescita* $\texttt{fibo}(n)/\texttt{fatt}(n)$.

**2.3. La torre di Hanoi.** Un altro classico problema (capostipite di una famiglia di problemi e giochi risolti al calcolatore) è quello della risoluzione del gioco della *torre di Hanoi*. Si tratta di spostare $n$ dischetti concentrici dal piolo $a$ al piolo $b$ utilizzando all'occorrenza il piolo $c$ e facendo sì che non succeda mai che un disco più grande stia al di sopra di uno più piccolo (si assuma che la situazione iniziale soddisfi tale condizione).

L'idea della soluzione ricorsiva è la seguente: per spostare $n$ dischetti dal piolo $a$ a quello $b$ usando $c$ posso

- prima spostare $n-1$ dischetti dal piolo $a$ a quello $c$ usando $b$;

[1] Si rimanda il lettore interessato a qualunque testo di analisi matematica per scoprire la relazione tra tale valore e la parte aurea del segmento, che ha reso così famosa questa funzione.

- poi spostare l'ultimo dischetto (il più grande) di $a$ in $b$,
- quindi sopstare gli $n-1$ dischetti dal piolo $c$ a quello $b$ usando $a$.

Il problema è stato dunque ridotto alla soluzione di due problemi analoghi ma di una dimensione minore e in un semplice spostamento.

ESERCIZIO 11. *Si scriva un programma* **C** *utilizzante una procedua ricorsiva che risolve il problema della torre di Hanoi stampando la sequenza delle mosse da compiere (le stampe saranno del tipo:* `muovi il disco più in alto del piolo` $x$ `sopra la pila di dischi del piolo` $y$*).*

## 3. Ottimizzazione di funzioni ricorsive

Nell'esercizio 10 abbiamo calcolato il numero di chiamate di funzione necessarie a calcolare il valore `fibo`$(n)$. Tale numero appariva essere molto alto in confronto al numero $n$. La ragione di ciò è legata al fatto che, ad esempio, per calcolare `fibo`$(4)$, abbiamo bisogno di calcolare `fibo`$(3)$ e `fibo`$(2)$; per calcolare `fibo`$(2)$ ci servono `fibo`$(1)$ e `fibo`$(0)$ che portano subito al risultato. Manca però ancora il calcolo di `fibo`$(3)$ per il quale ci serve `fibo`$(2)$ e `fibo`$(1)$. `fibo`$(1)$ porta immediatamente al risultato, mentre `fibo`$(2)$ va calcolata ancora utilizzando `fibo`$(1)$ e `fibo`$(0)$. In definitiva:

- `fibo`$(0)$ viene calcolata 2 volte.
- `fibo`$(1)$ viene calcolata 3 volte.
- `fibo`$(2)$ viene calcolata 2 volte.
- `fibo`$(3)$ viene calcolata 1 volta.

Il fenomeno assume proporzioni allarmanti per valori di $n$ piuttosto grandi: si potrebbe pensare di migliorare l'esecuzione cercando di *ricordare* i valori calcolati, utilizzando un vettore visto come variabile globale, nel seguente modo:

```
int store[100];
main(){
    int fibo();
    int i;

    for (i = 0; i < 100; i = i + 1)
        store[i] = 0;
    printf("Dammi un valore: ");
    scanf("%d", &i);
    printf("Fibonacci(%d) = %d \n", i , fibo(i));
}
int fibo( int n ){
    if (store[n] == 0){
       if( n <= 1)
        store[n] = 1;
        else  store[n] = fibo(n - 1) + fibo(n - 2);
    }
    return store[n];
}
```

Si osservi come ora, qualunque sia $i < n$, `fibo`$(i)$ viene chiamato al più una volta, preservando tuttavia la natura ricorsiva della definizione.

CAPITOLO 6

# Esercizi

In questo capitolo si vogliono presentare esempi di stesura funzioni e programmi con le strutture dati e le potenzialità fin qui analizzate.

### 1. Aritmetica su interi grandi a piacere

Come visto nei capitoli precedenti, ogni tipo di dato built-in ammette un certo insieme di valori ammissibili. Qualora si necessiti di operare con numeri maggiori (o minori) risulta necessario utilizzare una diversa rappresentazione di tali numeri e reimplementarsi gli algoritmi per il calcoli su essi. Nel caso degli interi, una tecnica è quella di considerare vettori di dimensione `DIM` fissata. Ciascun elemento del vettore conterrà una ed una sola cifra del numero in questione. Nel seguito si mostra come implementare le operazioni di somma e moltiplicazione in questo contesto. Gli algoritmi sono quelli imparati alle scuole elementari.

```
#include <stdio.h>
#define DIM 5
typedef enum {false,true} bool;

main(){
/* Assegno i vettori qui per evitare input-output lunghgi */
    int v[DIM] = {0,0,0,1,5};
    int w[DIM] = {0,0,1,3,2};
    int z[DIM];
    void somma(int [], int [], int []),
         prodotto(int [], int [], int []),
         inverti(int []), stampa(int []);

    inverti(v);
    inverti(w);
    stampa(v);
    printf(" +\n");
    stampa(w);
    printf(" =\n");
    somma(v,w,z);
    stampa(z);
    printf(" \n");

    stampa(v);
    printf(" *\n");
    stampa(w);
```

```
    printf(" =\n");
    prodotto(v,w,z);
    stampa(z);
    printf(" \n");
}

void stampa(int v []){
    int i;
    for(i = DIM-1; i >= 0; i--)
       printf("%d ",v[i]);
}

void inverti(int v []){
    int i,temp;
    for(i = 0; i <= DIM/2; i++){
        temp = v[i];
        v[i] = v[DIM-1-i];
        v[DIM-1-i] = temp;
    }
}

void somma(int x [], int y [], int z []) {
   int i,c;
   c = 0;
   for(i = 0; i < DIM; i++){
       z[i] = (x[i] + y[i] + c) % 10;
       c = (x[i] + y[i] + c) / 10;
    }
    if (c==1)
       printf("Overfull\n");
}

void prodotto(int x [], int y [], int z []) {
   int i,j,c,temp;
   bool ovf;
   ovf = false;
   for(i = 0; i < DIM; i++)
      z[i] = 0;
   for(i = 0; i < DIM; i++){
      c = 0;
      for(j = 0; j < DIM; j++)
           if (i + j < DIM) {
               temp = x[j]*y[i] + z[i+j] + c;
               z[i+j] = temp % 10;
               c = temp / 10;
           }
           else if (x[j]*y[i] + c > 0)
              ovf = true;
```

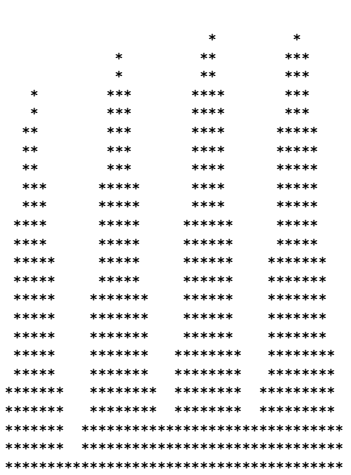
```
       }
    if (ovf)
      printf("Overfull\n");
}
```

## 2. Calcolo e stampa (primitiva) di funzioni

Il seguente programma illustra come calcolare in alcuni punti una data funzione matematica (definita nella procedura `funzione`). I risultati sono immagazzinati nel vettore `v`. Si mostra anche come costruire una rudimentale interfaccia grafica di output per diagrammare la funzione.

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

#define DIMX 80
#define DIMY 25

main(){
  void calcolafunzione(int [], int), stampavettore(int[], int),
       matrixplot(int [], char [DIMX][DIMY], int),
       stampamatrice(char [DIMX][DIMY],int,int);
  int v[DIMX],n;
  char m[DIMX][DIMY];

  printf("Dammi il valore massimo:  ");
  scanf("%d",&n);
  if (n < DIMX) {
        calcolafunzione(v,n);
        stampavettore(v,n);
        matrixplot(v,m,n);
        stampamatrice(m,n,DIMY);
  }
  else printf("valore troppo grande");
}

void stampavettore(int v [],int n){
  int i;

  for (i=0; i<n; i++){
     printf("V[%d] = %d\t",i,v[i]);
     if ((i+1) % 4 == 0)
         printf("\n");
     }
  printf("\n");
}

void calcolafunzione(int v [], int n){
  int funzione(int), i;
  for (i=0; i<n; i++)
```

```
     v[i] = funzione(i);
}

int funzione(int  x){
     return (int)ceil( log(x+1) + 5 * sin(0.6 * x));
}

void matrixplot(int v[], char m [DIMX][DIMY], int n){
   int max(int [],int),min(int [],int),
       i,j,minimo;
   float unit;

   minimo = min(v,n);
   unit =  (float)DIMY / (max(v,n) - minimo);
   for(i = 0; i < n; i++)
     for(j = 0; j < DIMY; j++)
        if(((v[i]-minimo) * unit ) < j)
            m[i][j] = ' ';
         else m[i][j] = '*';
}

void stampamatrice(char mat [DIMX][DIMY], int x, int y){
    int i,j;
   for(j = y-1; j >= 0; j = j - 1){
     for(i = 0; i < x; i++)
        printf("%c",mat[i][j]);
     printf("\n");
   }
}

int max(int v [], int n){
    int i,M;
    M = v[0];
    for(i = 1; i < n; i++)
       if( v[i] > M)
          M = v[i];
    return M;
}

int min(int v [], int n){
    int i,M;
    M = v[0];
    for(i = 1; i < n; i++)
       if( v[i] < M)
          M = v[i];
    return M;
}
```



ESERCIZIO 12. *Si scriva un programma in grado di accettare la misura di un diametro e del valore di una superficie ed in grado di rispondere se tale superficie possa essere considerata l'area di un cerchio avente tale diametro.*

ESERCIZIO 13. *Si scriva un programma che accetti come input i valori a, b, e c relativi alla generica equazione di secondo grado* $ax^2 + bx + c = 0$*, e che fornisca in uscita le radici (soluzioni) dell'equazione, qualore queste esistano. (E' necessario aggiungere prima della riga* `main` *la riga:*

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

*in cui si dichiara di voler usare il file* `math.h` *indispensabile per poter usufruire della dunzione di libreria* `sqrt()` *(square root).)*

### 3. esercizi grafici

Si scriva una procedura che stampa il contenuto di una matrice di caratteri. Si scrivano poi le funzioni **C** per assegnare a matrici di dimensione fissata dall'utente i valori che permettono di ottenere come output le forme in Figura 1. Per i punti *c* (quadrati ricorsivamente annidati) e *d* (spirale) si consiglia di usare la ricorsione.

### 4. Ordinamento di vettori

Vediamo come esempio il problema dell'ordinamento di un vettore. Il seguente programma utilizza sia la procedura di ordinamento quadratica basata sull'idea del *bubble sort* che la procedura di ordinamento di complessità $n \log n$ mergesort. La variabile globale scambi permette di avere una stima del numero di operazioni compiute in entrambi i casi. La procedura `leggi_vettore` non richiede qui un vettore in input ma ne inserisce uno completamente disordinato (per esercizio definetene anche una che inserisce un vettore a caso). Si provi il programma con diversi valori per la costante `DIM` (si commenti/decommenti la stampa del vettore secondo la sua lunghezza).

```
#include <stdio.h>
#define DIM 1000
int scambi;

/* Programma principale */
```

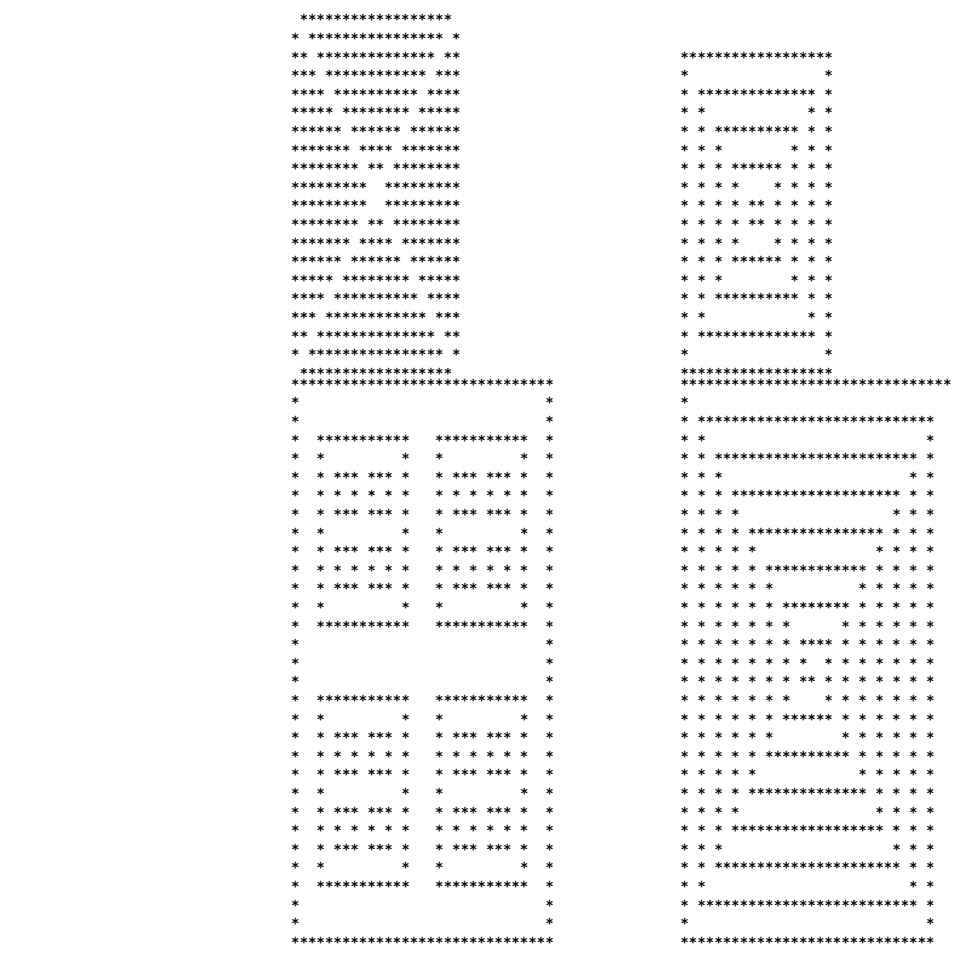


FIGURA 1. Figure a–b, c–d

```
main(){
  void leggi_vettore(int []);
  void ordina(int []);
  void stampa_vettore(int []);
  void mergesort(int [], int, int);
  int  vettore [DIM];

  scambi = 0;
  leggi_vettore(vettore);
  ordina(vettore);
/*   stampa_vettore(vettore); */
  printf("\n\t Scambi: %d\n",scambi);

  scambi = 0;
  leggi_vettore(vettore);
  mergesort(vettore,0,DIM-1);
```

```
/*   stampa_vettore(vettore); */
  printf("\n\t Numero Accessi ai vettori: %d",scambi);
}

/* Procedura (bubble sort) per ordinare un vettore dato */
void ordina(int v[]){
      void scambia(int[], int, int);
     int i,j;

     for(i=0; i<DIM; i++)
       for(j=i+1; j<DIM; j++)
          if (v[j] < v[i])
               scambia(v,i,j);
}

/* Procedura di scambio di due elementi */
void scambia(int v[], int i, int j){
     int t;
     t = v[i];
     v[i] = v[j];
     v[j] = t;
     scambi++;
}

/* Procedura fittizia che lo inserisce in ordine inverso */
void leggi_vettore(int v []){
    for(int i = 0; i < DIM; i++)
       v[i]= 2*DIM -i;
}

void stampa_vettore(int v []){
    for(int i = 0; i < DIM; i++)
       printf("%d\t",v[i]);
}

/* Procedura ricorsiva mergesort */
void mergesort(int v[], int b, int e){
    void unisci(int[],int,int,int);
    if (b < e){
        mergesort(v,b,(b+e)/2);
        mergesort(v,(b+e)/2+1,e);
        unisci(v,b,(b+e)/2,e);
    }
}

/* Procedura di unione (merge) */
void unisci(int v[], int b, int m, int e){
   int i,j,k;
```

```
   int w[DIM];

   i=b;
   j=m+1;
   k=i;
    /* Entrambi i sottovettori sono ancora da analizzare */
   while((i<=m) && (j<=e)){
       scambi++;
       if(v[i] < v[j]){
           w[k] = v[i];
           i++;
        }
        else{
            w[k] = v[j];
            j++;
        }
        k++;
    }
    /* E' finito prima il secondo pezzo */
   while(i<=m){
       scambi++;
       w[k] = v[i];
        i++;
        k++;
    }
    /* E' finito prima il primo pezzo */
    while(j <= e){
       scambi++;
       w[k] = v[j];
        j++;
        k++;
    }
    for(i = b; i <= e; i++)
        v[i] = w[i];
}
```

Esercizio 14. *Si implementi questo programma e si produca una sua variante che utilizza una variabile globale o locale alla procedura di ordinamento che conta il numero di scambi effettuati.*

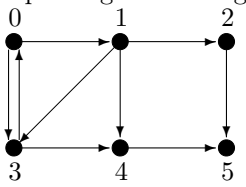
Esercizio 15. *Si definisca una variante di* `leggi_vettore` *che assegna i valori al vettore senza richiederli all'utente, in modo da generare vettori lunghi (basta aumentare il valore di* `DIM`*) e disordinati.*

Esercizio 16. *Si utilizzi la funzione* `rand()` *della libreria* `stlib.h` *per generare i valori casualmente.*

Esercizio 17. *Si implementi la procedura ricorsiva* `mergesort` *vista a lezione e si effettuino dei tests pratici utilizzando le procedure degli esercizi precedenti per mostrare la sua efficienza rispetto alla procedura* `ordina`.

## 5. Visite di grafi

Un grafo diretto è una coppia $G = \langle N, E \rangle$ dove $N$ è un insieme di nodi (per semplicità possiamo assumere $N = \{0, \ldots, k\}$ per un opportuno $k$ ed $E \subseteq N \times N$ è una relazione binaria. Un esempio di grafo è il seguente:




Un grafo si rappresenta mediante una matrice Booleana, ove `grafo[i][j]` è `true` se e solo se esiste un arco dal nodo `i` al nodo `j`. Il seguente programma vuole stabilire se esiste un cammino nel grafo tra due nodi accettati come input. Il grafo è, per comodità, una variabile globale. Vi è anche una variabile globale `visitato` che permette di evitare di rivisitare lo stesso nodo due volte, al fine di non entrare in ciclo.

```
#include <stdio.h>
#define DIM 6

bool grafo [DIM][DIM] =
       {{0,1,0,1,0,0},
        {0,0,1,1,1,0},
        {0,0,0,0,0,1},
        {1,0,0,0,1,0},
        {0,0,0,0,0,1},
        {0,0,0,0,0,1}};
bool visitato [DIM] =
        {0,0,0,0,0,0};

/* funzione main */
main(){
     bool cammino(int,int);
    int s,t;

    printf("Dammi il nodo sorgente ");
    scanf("%d",&s);
    printf("Ed il nodo destinazione ");
    scanf("%d",&t);
    if (cammino(s,t))
       printf("Si, sono connessi");
    else printf("Non c'e' cammino");
}

/* funzione cammino  */
bool cammino(int s, int t){
    int i;
    bool path;
```

```
    path = false;
    if (!visitato[s]){
       visitato[s] = true;
       if (s == t)
          path = true;
       else {
          i = 0;
          while ((!path) && (i < DIM) ){
              if (grafo[s][i])
                  path = path || cammino(i,t);
              i = i + 1;
          }
       }
    } /* if  !visitato[s] */
    return path;
}
```

Si osservi l'uso di variabili Booleane e l'aggiornamento con l'**or** nella chiamata ricorsiva.

Si modifichi il programma in modo tale da indicare, qualora esista, il cammino da seguire per raggiungere il nodo `t`.

CAPITOLO 7

# Alcuni approfondimenti

## 1. Il Preprocessore C

Quando si compila un programma **C**, viene effettuato un passo preliminare, che precede la compilazione vera e propria: immediatamente prima del compilatore entra in azione il *preprocessore*. Un programma **C** può contenere alcune istruzioni rivolte al preprocessore; vediamo le più semplici.

**1.1. Le macro.** La `#define` serve per definire costanti, funzioni, ecc. ecc. Il tutto viene *espanso* dal preprocessore **C**: ad ogni occorrenza del primo parametro della `#define` viene sostituito il secondo parametro. Esempi:

```
#define PIGRECO 3.14
#define DIM 10
...
int vettore[DIM]; /* MOLTO MEGLIO di int vettore[10]; */
```

In questo modo, a `PIGRECO` viene sostituito 3.14 e a `DIM` 10.

È possibile effettuare definizioni di funzioni, tipi, ecc. ecc.

**1.2. Inclusione di file.** La `#include` serve per includere altri file, "come se fossero lì". Di solito si usa per includere file contenenti `#define` di costanti, o prototipi di funzioni di libreria (così che quando poi il compilatore li trova può verificare il controllo dei tipi).

Esempi:

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

Il "`.h`" sta per "header file". Lo header file `<stdio.h>` contiene i prototipi delle funzioni `printf`, `scanf`, ... e le `#define` di `EOF`, `NULL`, ... Ad esempio, in `<stdio.h>` compare una riga del tipo:

```
#define EOF     (-1)
```

che definisce `EOF`.

Le parentesi angolari (`<>`) indicano che la libreria va cercata nelle directory di sistema; le virgolette (`‘‘`) che la libreria va cercata nella directory in cui di trova il file sorgente **C** contenente la `#include`.

**1.3. L'inclusione condizionale.** Oltre a `#define` e `#include`, vi sono molte altre istruzioni al preprocessore. Vediamo ancora l'inclusione condizionale. È possibile scrivere codice del tipo:

```
#if ...
#define ...
#else
#define ...
```

```
#endif
```

Una condizione che viene spesso controllata nei `#if` è se un certo oggetto è definito o meno. Per fare ciò vi sono due abbreviazioni, `#ifdef` e `#ifndef`, vere se l'oggetto specificato è (rispetivamente, non è) definito. Ad esempio, la definizione di `EOF` in `<stdio.h>` è in realtà:

```
#ifndef EOF
#define EOF     (-1)
#endif
```

in modo da evitare di definirlo due volte se `<stdio.h>` è incluso due volte.

## 2. La libreria standard

Una *libreria* è un insieme di funzioni (tipi, costanti, ecc. ecc.) predefinite e di uso frequente. Non fa parte del linguaggio vero e proprio, ma dell'*ambiente*. L'ambiente **C** mette a disposizione un insieme di librerie *standard*: sono le stesse (stesse funzioni, con stessi nomi) in *ogni* implementazione del linguaggio. Per diventare un buon programmatore **C** (e per risparmiare fatica inutile) bisogna essere in grado di utilizzare al meglio le librerie. Ne vediamo molto brevemente alcune parti (si consultino altri testi per ulteriori dettagli).

Le funzioni contenute nelle librerie sono già compilate, e vengono fuse insieme al programma sorgente durante la fase di *link*, subito dopo la compilazione vera e propria. Sembra quindi che il compilatore possa disinteressarsi delle librerie, ma non è così. Infatti, il compilatore deve effettuare il controllo dei tipi, e ha quindi bisogno, ad esempio, dei prototipi delle funzioni definite nelle librerie. Per questo motivo, quando si usa una libreria, bisogna includerne (con una `#include`) il corrispondente *header file*. È per questo motivo che dovete inserire ad esempio una `#include <stdio.h>` quando nel programma avete una `printf` (date un'occhiata alla nota 1).

**2.1.** `<stdio.h>`**.** È la libreria dedicata all'ingresso/uscita (input/output). Vi troviamo, fra le varie cose:

- le funzioni per lavorare su file;
- le funzioni per l'I/O formattato:
  - `int fprintf (FILE *flusso, const char *formato, ...);`
  - `int printf (const char *formato, ...)` (= `fprintf(stdout, const char *formato, ...)`);
  - `int fscanf (FILE *flusso, cont char *formato, ...);`
  - `int scanf (const char *formato, ...)` (= `fscanf(stdin, const char *formato, ...)`);
  - ... ce ne sono molte altre, si vedano i manuali.

**2.2.** `<math.h>`**.** Libreria contenente le funzioni matematiche (`x` e `y` sono `double`, e tutte le funzioni restituiscono `double`):

- `sin(x), cos(x), tan(x), exp(x), log(x), log10(x), pow(x,y), sqrt(x), fabs(x), ....`

**2.3.** `<string.h>`**.** Libreria contenente le funzioni sulle stringhe:

- `char *strcpy(char *to, const char *from);`
- `char *strcat(char *to, const char *from);`

- `int *strcmp(const char *to, const char *from);`
- `size_t strlen(const char *);`
- ...

**2.4.** `<stdlib.h>`**.** Libreria contenente funzioni per conversioni, allocazione di memoria, ecc. ecc.

- `int rand(void);`
- `void srand(unsigned int seme);`
- `void *malloc(size_t dimensione);`
- `void free(void *p);`
- `void exit(int stato);`
- ...

**2.5.** `<assert.h>`**.** Librerie contenenti funzioni per diagnostica.

- `void assert(int espressione)` (se espressione vale zero, stampa un messaggio del tipo "`Assertion failed: espressione, file <nome>, line N`" e poi termina il programma.)

**2.6.** `<ctype.h>`**.** Libreria contenente funzioni per controlli sulla classe dei caratteri. (L'argomento di ogni funzione è un `unsigned char` oppure `EOF`; ogni funzione ritorna un `int`: 0 per FALSE, un valore non nullo per TRUE)

- `isdigit(c);`
- `isspace(c);`
- `isupper(c);`
- `islower(c);`
- ...
- `int tolower(int c).`

## 3. I file

**3.1. L'apertura.** Come sapete già, un file ha un suo nome nel sistema operativo. L'*apertura* di un file all'interno di un programma serve per "prenotare" il file e per associarvi il nome di una variabile del programma, variabile che le funzioni successive useranno per fare riferimento al file (si dice che si associa al file fisico un *flusso*, o *stream*). La funzione **C** per aprire un file è la `fopen`; la sua intestazione (la si ritrova, a parte i nomi dei parametri, nel file `<stdio.h>`) è:

```
FILE *fopen(const char *nomefile, const char *modo);
```

dove `nomefile` indica il cammino (*pathname*) del file nel sistema operativo, `modo` può assumere uno dei valori seguenti:

- "`r`": (read) solo lettura;
- "`w`": (write) solo (ri)scrittura (i contenuti precedenti vengono persi);
- "`a`": (append) aggiunta: scrittura alla fine del file;
- ... altri che non vediamo.

e la `fopen` restituisce un puntatore ad un file (`fp`), che verrà usato nelle funzioni successive. Il tipo `FILE` è definito come una `struct` in `<stdio.h>`.

Se la `fopen` fallisce (ad esempio, se si prova ad aprire in lettura un file che non esiste), la `fopen` restituisce il valore `NULL`.

**3.2. La chiusura.** La chiusura libera il file fisico e libera anche la variabile interna al programma (di solito il numero di file contemporaneamente aperti in un programma è limitato). La funzione **C** è:

```
int fclose(FILE *fp)
```

**3.3. La lettura.** Per leggere da un file di testo, si usa una funzione simile alla ben nota `scanf` (infatti, l'input è a tutti gli effetti un file particolare, denominato *standard input*, o `stdin`):

```
int fscanf (FILE *fp, const char *formato, ...)
```

I puntini `...` indicano un numero non specificato di parametri. Il file puntato da `fp` deve essere stato aperto in precedenza.

**3.4. La scrittura.** Anche la scrittura su un file è analoga alla scrittura sul file *standard output* (`stdout`):

```
int fprintf (FILE *fp, const char *formato, ...)
```

**3.5. La gestione degli errori.** Nella gestione nei files si usano spesso le seguenti funzioni:

- `int feof(FILE *fp)`: restituisce un valore non nullo se è stata raggiunta fine file nell'operazione precedente;
- `int ferror(FILE *fp)`: restituisce un valore non nullo se c'è stato errore nell'operazione precedente.

**3.6. Esempi.** L'uso tipico delle funzioni precedenti all'interno di un programma è il seguente:

```
...
fp = fopen(<pathname>, "r"); /* Apertura */
...
fscanf(fp,...);  /* Lettura */
...
fclose(fp); /* Chiusura */
...
```

La lettura (o la scrittura) può eventualmente essere ripetuta, usando `feof` per verificare quando si è letto l'ultimo elemento.

ESERCIZIO 18. *Creare un file, scriverci 2 righe lette da input e chiuderlo. Come faresti per verificare che il file contiene effettivamente le due linee inserite?*

**3.7. Lettura e scrittura di caratteri.** Introduciamo le due principali istruzioni:

- `int fgetc(FILE *fp)`: legge un carattere da file.
- `int fputc(int c, FILE *fp)`: scrive un carattere su file. Attenzione all'ordine dei parametri!

ESEMPIO 7.1. *Stampa di un file (carattere per carattere) su stdout.*

```
/*Programma per stampare un file carattere per carattere. Versione 1*/
#include <stdio.h>

void main(){
  FILE *fp;
```

```
  char c, *nomefile;

  printf("Inserisci il nome del file da visualizzare: ");
  scanf("%s",nomefile);
  fp = fopen(nomefile,"r"); /* Apertura */
  if (ferror(fp))
    printf("ERRORE! Il file %s non puo' essere aperto.\n", nomefile);
  else {
      c = fgetc(fp);  /* Lettura primo carattere */
      while (!feof(fp)){
         putchar(c); /* Stampa carattere */
         c = fgetc(fp); /* Lettura carattere */
      }
  }
  fclose(fp); /* Chiusura */
}
```

*Ma nessun programmatore* **C** *lo scriverebbe così! Invece:*

```
/*Programma per stampare un file carattere per carattere. Versione 2*/
#include <stdio.h>
#include <stddef.h> /* contiene la def. di NULL */

void main(){
  FILE *fp;
  char c, *nomefile;

  printf("Inserisci il nome del file da visualizzare: ");
  scanf("%s",nomefile);
  if ((fp = fopen(nomefile,"r")) == NULL) /* Apertura */
    printf("ERRORE! Il file %s non puo' essere aperto.\n", nomefile);
  else
    while ((c = fgetc(fp)) != EOF) /* Lettura carattere */
       putchar(c); /* Stampa carattere */
  fclose(fp); /* Chiusura */
}
```

**3.8. Cenni ai file ad accesso diretto.** Con le funzioni viste in precedenza è molto scomodo effettuare *modifiche* su un file: se volessimo modificare un lunghissimo file in un unico punto dovremmo aprirlo in lettura, leggerlo interamente salvando il suo contenuto in memoria centrale, effettuare la modifica in memoria centrale, chiudere il file in lettura, riaprirlo in scrittura e riscriverne il contenuto. Il **C** mette a disposizione anche la possibilità di posizionarsi su un file e modificare solo il punto in cui ci si trova; noi non vediamo questa possibilità, si consulti un testo più approfondito.

Esercizio 19. *Scrivete un programma che legge da standard input Autore e Titolo di 10 libri e li memorizzazione in un file (memorizzare Autore e Titolo a righe alterne). Provare ad aprire il file con un editor per controllarne il contenuto.*

ESERCIZIO 20. *Scrivete un programma che legge il file con i 10 libri, lo stampa in un formato opportuno, lo memorizza in un vettore, ordina il vettore per autore, lo stampa, e salva sul file iniziale con il nuovo ordinamento.*