Manuale di Programmazione in C
Esempi ed Esercizi

Angelo Ciaramella, Giulio Giunta

Università degli Studi di Napoli "Parthenope"
Dipartimento di Scienze Applicate

Indice

Elenco delle Tabelle

Elenco delle Figure

Introduzione

Il **C** è un linguaggio di programmazione di uso generale, caratterizzato dalla sinteticità, da un controllo del flusso e da strutture dati avanzate e da un vasto insieme di operatori. Questo manuale di programmazione si propone come strumento didattico integrativo per gli studenti che seguono il corso di Programmazione I e Laboratorio di Programmazione I, del I anno del corso di Laurea in Informatica. Il libro presenta, con una decisa connotazione pratica, i concetti teorici e metodologici introdotti nelle lezioni del corso, corredandoli di esempi e proponendo nuovi esercizi. Lo studente può inoltre focalizzare l'attenzione sugli esercizi identificati dal simbolo [✓] poichè sono quelli selezionati per la "prova all'impronta" durante la prova di verifica dell'esame finale del corso.

Napoli, 28 Settembre 2009

Angelo Ciaramella, Giulio Giunta

Gli autori desiderano ringraziare Paolo Borrelli e Claudia Botti per le loro "preziose critiche".

Capitolo 1

Introduzione al linguaggio C

Lo scopo di questo Capitolo è introdurre alcune nozioni fondamentali del linguaggio C come i tipi di dati, le variabili, le costanti e i puntatori. Vengono illustrate le fasi principali per la scrittura, compilazione ed esecuzione di un programma C usando un ambiente di sviluppo integrato.

1.1 Ambiente di sviluppo integrato

Il nostro Integrated Development Environment (IDE), in italiano ambiente integrato di sviluppo, per scrivere i nostri programmi è **Code::Blocks** che è un prodotto software libero, *open source* e multipiattaforma. Può essere scaricato dalla pagina web *www.-codeblocks.org/*. L'ambiente è scritto in C++ e, usando un'architettura basata su *plugin*, le sue capacità e caratteristiche sono estese proprio dai plugin installati. Attualmente, **Code::Blocks** è orientato verso il C/C++. Il layout principale di questo IDE è visualizzato in Figura 1.1. **Code::Blocks** non si basa su un singolo compilatore ma permette di scegliere diversi compilatori come è mostrato in Figura 1.2. In questo testo faremo riferimento al compilatore **GNU GCC**. **GCC**, GNU Compiler Collection, fa parte del progetto GNU (www.gnu.org), che fu lanciato nel 1984 da Richard Stallman con lo scopo di sviluppare applicazioni su sistemi operativi di tipo Unix (Linux) che fossero completamente "*free*" software.

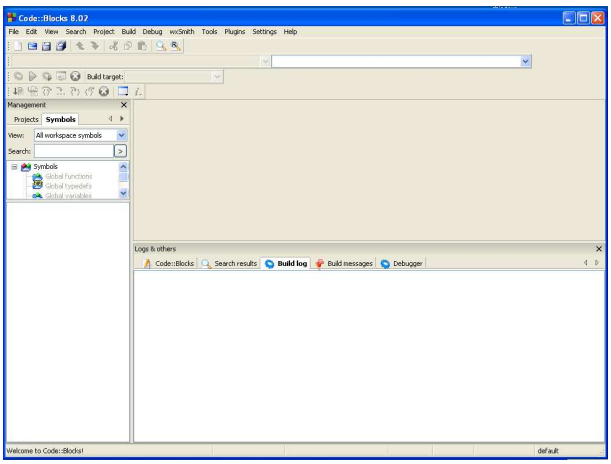

Figura 1.1: Layout principale di **Code::Blocks**.

1.2 Linguaggio di programmazione C

Per imparare un linguaggio di programmazione, il modo più rapido ed efficace è quello di utilizzarlo. Per questo motivo iniziamo analizzando il seguente programma C

```
#include <stdio.h>


 void main()
 {
  printf("questo e' il mio primo programma C \n");
 }
```

La prima linea del programma

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

richiede al compilatore di includere le informazioni relative alla libreria standard di input/output. Come vedremo nel seguito, le librerie possono essere già definite in C oppure possiamo costruirne noi altre. Lo scopo di una libreria è quella di contenere le *funzioni* [1] che possono essere utilizzate nei programmi scritti dagli utenti.

La successiva riga identifica l'inizio del programma principale. Il **main**, infatti, è la funzione principale che ci permette di eseguire il nostro programma. La parola **void** viene inserita per identificare una funzione che non restituisce valori. Può, comunque, ritornare

[1] Nel testo usiamo il termine "**funzione**" per tradurre il termine inglese "**function C**"

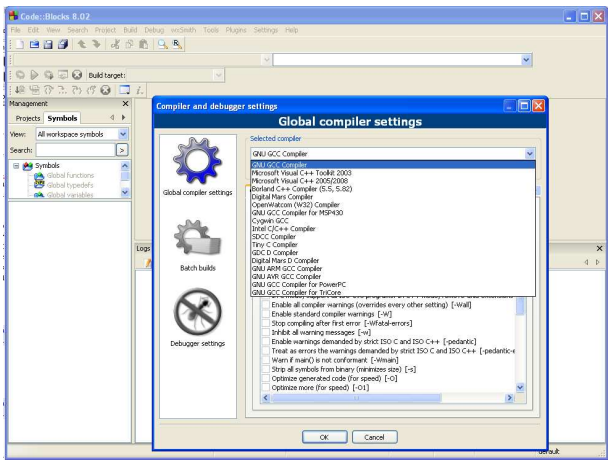

Figura 1.2: Selezione del compilatore in **Code::Blocks**.

anche un valore intero, come vedremo successivamente. Il corpo del **main** è racchiuso tra parentesi graffe. Il linguaggio C usa le parentesi graffe per identificare qualsiasi blocco di istruzioni da eseguire.

All'interno del **main** viene chiamata la funzione **printf**, contenuta nella libreria **stdio**, che permette di visualizzare una sequenza di caratteri o stringa ("questo e' il mio primo programma C" nel nostro caso); **n** indica la nuova linea e cioè dopo aver visualizzato la stringa il cursore si sposta alla linea successiva.

Dopo aver analizzato il programma C vogliamo "scriverlo" e fare "eseguire". Che cosa bisogna fare ?

Bisogna usare l'ambiente di sviluppo **Code::Blocks** e costruire un "progetto" (vedi Figura 1.3). Un **Wizard** ci permette di costruire automaticamente un nuovo progetto. Scegliamo un progetto del tipo *Console Application*, e scegliamo come titolo *programma_1*. Successivamente scriviamo il nostro programma C come in Figura 1.4. A questo punto, mediante il primo tasto della Figura 1.5 possiamo **Compilare/Linkare** il programma e con il secondo eseguirlo. Il risultato è visualizzato in Figura 1.6.

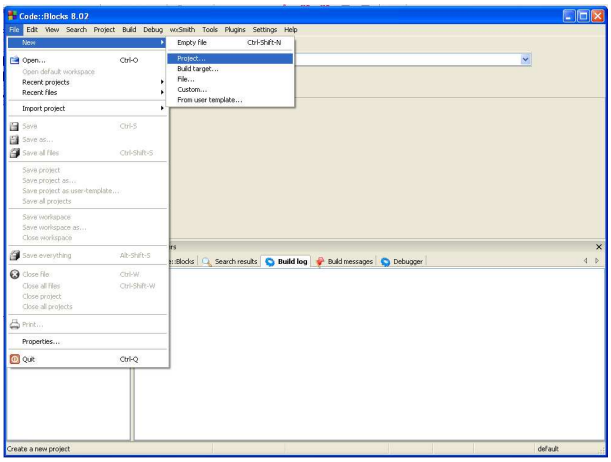

Figura 1.3: Creazione di un nuovo progetto in **Code::Blocks**.

1.2.1 Esercizi

Esercizio 1

Scrivere un programma che visualizzi la seguente stringa "**Hello World!!!**". Scriverlo e compilarlo con **Code::Blocks** o **Visual C++**. Visualizzare il risultato.

Esercizio 2

Scrivere un programma che visualizzi la seguenti frasi "**Questa è la prima riga**" e "**Questa è la seconda riga**" su due righe differenti. Scriverlo e compilarlo con **Code::Blocks** o **Visual C++**. Visualizzare il risultato.

Esercizio 3

Scrivere un programma che visualizzi i seguenti versi

```
Mi domandate perchè in alto io viva?

Non rispondo, sorrido, il cuore in pace;

il fiume scorre, transitano i venti:

quest'è il mio mondo, diverso dal vostro.
```

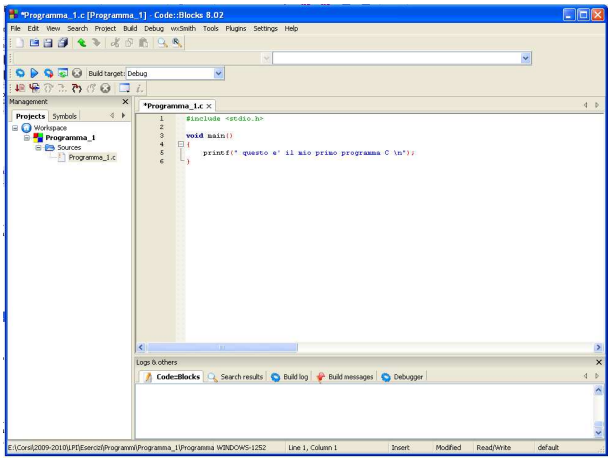

Figura 1.4: Scrittura del programma in **Code::Blocks**.

Figura 1.5: Compilazione ed esecuzione in **Code::Blocks**.

Scriverlo e compilarlo con **Code::Blocks** o **Visual C++**. Visualizzare il risultato (N.B.: considerare anche gli spazi e il rigo vuoto).

Esercizio 4

Scrivere un programma che visualizzi le seguenti frasi "**L'immagine del sole si disegna tranquilla dentro l'acqua**" e "**non un'onda s'increspa nella quiete**" separate da una virgola. Scriverlo e compilarlo con **Code::Blocks** o **Visual C++**. Visualizzare il risultato.

Esercizio 5

Scrivere un programma che visualizzi la seguente frase

```
Corrono nuvole insistenti: corrono!

Sgocciola pioggia regolare: sgocciola!...
```

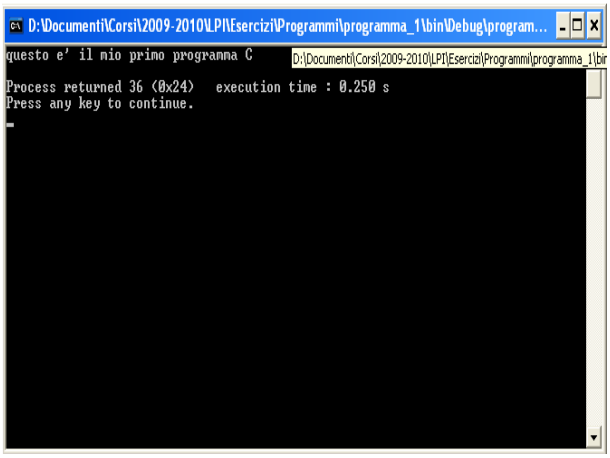

Figura 1.6: Esecuzione del programma con **Code::Blocks**.

usando una funzione **printf** per ogni parola o carattere. Scriverlo e compilarlo con **Code::Blocks** o **Visual C++**. Visualizzare il risultato (N.B.: considerare anche gli spazi e il rigo vuoto).

Esercizio 6

Scrivere, esattamente come mostrato, ed eseguire il seguente codice

```
#include <stdio.h>

void main()
 {
  printf("Il mio programma in C \n");
  printf("contiene qualche errore.")
 }
```

Verificare e correggere gli errori di compilazione.

Esercizio 7

Scrivere, esattamente come mostrato, ed eseguire il seguente codice

```
#include <stdio>

void main()
```

```
{
 printf("Il mio programma in C ");  printf(" funziona?");
}
```

Verificare e correggere gli errori di compilazione.

Esercizio 8

Scrivere , esattamente come mostrato, ed eseguire il seguente codice

```
#include <stdio.h>

void main()
  printf("Il mio programma in C ");  printf(" funziona?");
```

Verificare e correggere gli errori di compilazione.

Esercizio 9

Scrivere, esattamente come mostrato, ed eseguire il seguente codice

```
#include stdio.h

void main()
  printf("Il mio programma in C");
  printf("\t funziona?");
```

Verificare e correggere gli errori di compilazione.

Esercizio 10

Scrivere, esattamente come mostrato, ed eseguire il seguente codice

```
#include <stdio>

void main() {printf("\t Il mio programma in C") printf("\n
funziona?"); }
```

Verificare e correggere gli errori di compilazione.

char	un singolo byte, in grado di rappresentare uno qualsiasi dei caratteri del set locale
int	un intero, riflette l'ampiezza degli interi sulla macchina
float	floating-point in singola precisione
double	floating-point in doppia precisione

Tabella 1.1: Tipi di dati fondamentali.

Esercizio 11

Scrivere, esattamente come mostrato, ed eseguire il seguente codice

```
#include <stdio.h>


void() {printf("Il mio programma in C funziona?"); }
```

Verificare e correggere gli errori di compilazione.

1.3 Variabili e tipi in C

Le **variabili** e le **costanti** sono gli oggetti di base utilizzati in un programma. Le dichiarazioni elencano le variabili che dovranno essere usate e stabiliscono il loro tipo ed eventualmente anche il loro valore iniziale.

1.3.1 Nomi di variabili

Per i nomi delle variabili e delle costanti esistono alcune restrizioni. I nomi sono costituiti da lettere e da cifre; il primo carattere deve essere sempre una lettera. L'underscore ("_") talvolta consente di aumentare la leggibilità di nomi molto lunghi. Le lettere minuscole e maiuscole sono distinte e la prassi del C adotta lettere minuscole per i nomi delle variabili e maiuscole per le costanti. Le parole chiave come **if**, **else**, **int**, ..., sono riservate.

1.3.2 Tipi di dati e dimensioni

In C esiste un ristretto numero di tipi di dati fondamentali (vedi Tabella 1.1). In aggiunta

esistono alcuni qualificatori, che possono essere applicati a questi tipi; per esempio **short** e **long** sono associabili agli interi:

```
short int sh;
long int counter;
```

In queste dichiarazioni il termine **int** può essere omesso. Lo scopo è quello di fornire, ove necessario, delle lunghezze diverse per gli interi. Spesso, **short** indica un intero a 16 bit e **long** uno di 32 mentre **int** occupa 16 o 32 bit.
I qualificatori **signed** e **unsigned** possono essere associati sia ai **char** che agli **int**. I numeri **unsigned** sono sempre positivi o nulli. Ad esempio **unsigned char** (il **char** è 8 bit) assume valore tra 0 e 255 mentre **signed char** vanno da -128 a 127.
long double specifica variabili floating-point in doppia precisione.
Inoltre il comando **sizeof(tipo)** restituisce il numero di byte necessario per la rappresentazione di un valore del tipo.

1.3.3 Dichiarazioni

Tutte le variabili, prima di essere utilizzate, devono essere dichiarate. Una dichiarazione specifica il tipo e contiene una lista di una o più variabili di quel tipo che possono essere anche inizializzate. Ad esempio

```
int lower=0, upper=10, step=1;

char esc='\\', line[1000];

float eps = 1.0e-5;
```

Il qualificatore **const** permette di dichiarare delle costanti

```
const double e = 2.7182;

const char msg[] = "warning";
```

Una costante **long** è seguita da una **l** o da una **L** (ad esempio 123456789L). Le costanti prive di segno sono terminate con **u** o **U** mentre **ul** e **UL** indicano **unsigned long**. I suffissi **f** o **F** indicano una costante float, mentre **l** ed **L** indicano una costante **long double**.

\a	allarme (campanello)
\b	backspace
\f	salto pagina
\n	new line
\r	return
\t	tab orizzontale
\v	tab verticale
\\	backslash
\?	punto interrogativo
\'	apice singolo
\"	doppi apici
\ooo	numero ottale
\xhh	numero esadecimale

Tabella 1.2: Sequenze di escape.

Una costante carattere è un intero, scritto sotto forma di carattere racchiuso tra apici singoli come **'x'**.
Esistono anche delle sequenze di escape che sono elencate nella Tabella 1.2.
E' possibile definire delle costanti usando # **define** prima del main (parte del pre-processore). Ad esempio possiamo dichiarare

```
#define MAXLINE 1000
#define LEAP 1
```

1.4 Operatori aritmetici

Gli operatori aritmetici binari sono $+$, $-$, $*$, / e l'operatore modulo % (resto della divisione tra due interi [2]). E' buona norma usare come operandi degli operatori aritmetici, costanti e/o varibili dello stesso tipo. Il C ammette anche operandi di tipo diverso. Quando un

[2] la divisione tra due interi, detta divisione intera, produce come risultato un intero.

operatore ha operandi di tipo diverso, questi vengono convertiti in un tipo comune secondo un insieme di regole (dette *regole per le operazioni di tipo misto*).
Un esempio completo di programma che utilizza i concetti introdotti in questa unità è il seguente

```
#include <stdio.h>

/* calcolo circonferenza di un cerchio

 versione doppia precisione */

void main () {
 const double pi_greco = 3.14159265258416;
 double raggio, circon;

 raggio = 2.0; /* il Raggio è fissato */
 circon = 2.0 * pi_greco * raggio;
 printf ("circonferenza=%23.14lf\n",circon);
}
```

Le uniche conversioni automatiche sono quelle che trasformano un operando di un tipo *più piccolo* in uno di tipo *più grande*, senza perdita di informazione, come ad esempio una conversione da intero a float. Espressioni che possono provocare una perdita di informazione, come l'assegnazione di un intero a uno short o quello di un float a un intero, producono un messaggio di *warning* ma non sono illegali.
Infine, in qualsiasi espressione è possibile forzare particolari conversioni, tramite un operatore unario detto **cast**. Il cast specifica un tipo racchiuso tra parentesi tonde. Per capire il funzionamento dell'operatore consideriamo il seguente codice

```
int n = 5;
double x;
x = (double)n;
```

che assegna a **x** il numero 5, rappresentato come dato di tipo **double**.
Un ruolo particolare nel linguaggio C è ricoperto dalle operazioni di **incremento** e **decremento** di una variabile. In C per effettuare queste operazioni è possibile usare rispettivamente gli operatori $++$ e $--$. L'operazione di incremento $\mathbf{i = i + 1}$ viene effettuata usando una delle due espressioni $\mathbf{i{+}{+}}$ oppure $\mathbf{{+}{+}i}$. Il significato delle due espressioni è

comunque differente. Mentre **i++** ha il significato di incremento della variabile **i** dopo il suo utilizzo, il significato dell'operazione **++i** è quello dell' incremento della variabile **i** prima del suo utilizzo. Per capire la differenza tra le due espressioni consideriamo il seguente esempio

```
int i=1;
printf("%d\n",i++);
printf("%d\n",i);
```

In questo caso il risultato delle istruzioni è 1 e 2. Invece, il risultato delle seguenti istruzioni

```
int i=1;
printf("%d\n",++i);
printf("%d\n",i);
```

è 2 e 2.
In modo analogo per l'operazione di decremento **i = i - 1** sono usate le espressioni **i**– e –**i**.
Inoltre in C è possibile esprimere alcune operazioni in maniera sintetica. Ad esempio l'espressione

```
i = i + 10;
```

può essere scritta come

```
i += 10;
```

Lo stesso può essere fatto per il prodotto e la divisione. Ad esempio la seguente espressione

```
x *= 20;
```

descrivere il seguente prodotto

```
x = x * 20;
```

1.5 I/O formattati

La funzione **printf** trasforma dei dati dalla rappresentazione interna a una rappresentazione in caratteri. Essa converte, formatta e visualizza sullo schermo i suoi argomenti, sotto il controllo di una stringa di riferimento, detta **stringa di formato**

```
printf("stringa di formato", arg1, arg2, ...);
```

Ogni specifica di conversione (detta anche **codice formato**) inizia con un % e termina con un carattere di conversione, che specifica il tipo di dato da visualizzare. Ad esempio

```
printf("valore di eta=%d\n",eta);
```

In Tabella 1.3 vengono elencati tutti i caratteri di conversione.
I codici di formato hanno la seguente forma

```
% - n.m caratterre di conversione
```

dove "%" è il carattere di formato, "−" indica l'allineamento a sinistra, "**n**" è l'ampiezza (numeri di caratteri) del campo di visualizzazione, "." è il separatore. Il significato di "**m**" dipende dal carattere di conversione: se è "s", allora il dato è una stringa ed è il massimo numero di caratteri da visualizzare; se è "d", allora il dato è un intero ed è il massimo numero di cifre; se è "f" oppure "e", allora il dato è un float oppure un double ed è il massimo numero di cifre da visualizzare dopo il punto frazionario.
Ad esempio

```
char c1,c2;
c1 = 'P'; c2 = 'O';
printf("%c %4c %-8c\n", c1, c2, c1);
```

Il risultato dell'esecuzione del codice è presentata in Figura 1.8.
Il seguente codice, invece, mostra l'effetto della **printf** nella visualizzazione di dati di tipo **float**

```
float x, x1;
double x2;

x = 54.32572F;
printf("%10.3f\n", x);
printf("%10.5f\n", x);

x1 = 12.345678F;
printf("%16.7e\n", x1);

x2 = 12.34567890123456;
printf("%22.15e\n%22.15f\n", x2,x2);
```

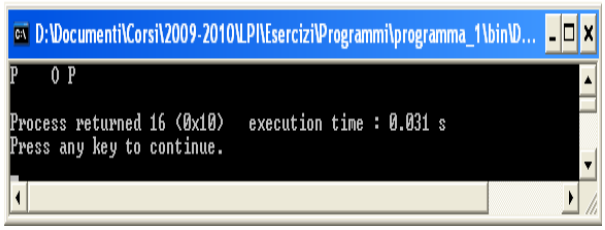

Figura 1.7: Esempio di utilizzo di **printf** con caratteri.

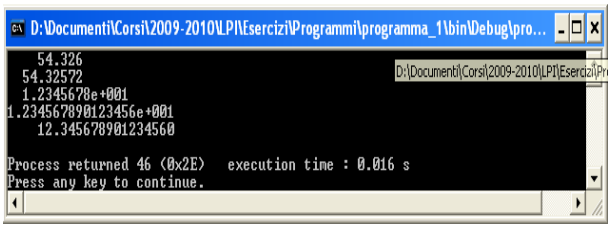

Figura 1.8: Esempio di **printf** con float.

La funzione **scanf** legge caratteri che vengono digitati sulla tastiera e li interpetra come dati, in base alla stringa di riferimento. Lo schema è il seguente

```
scanf("stringa di riferimento", &arg1, &arg2, ...);
```

Le conversioni sono simili a quelle della funzione **printf** riportate in Tabella 1.3.
Per capire come utilizzare l'input e l'output formattato riscriviamo il programma visto precedentemente

```
#include <stdio.h>


void main ()
{
 const float pi_greco = 3.1415926F;
 float raggio, circon;
 scanf("%f",&raggio);
 circon = 2.F*pi_greco*raggio;
 printf ("raggio=%f  circonferenza=%f\n", raggio,circon);
}
```

Per operazioni di input e di output di singoli caratteri è possibile usare le funzioni **getchar()** e **putchar()**, rispettivamente. Il seguente codice permette di leggere un carattere da tastiera e visualizzarlo

Carattere	Tipo argomento - Stampato come
d, **i**	**int** - numero decimale
o	**int** - numero ottale privo di segno
x, **X**	**int** - numero esadecimale privo di segno
u	**int** - numero decimale privo di segno
c	**int** - carattere singolo
s	**char *** - stringa
f	**double** - [-]m.dddddd
e, **E**	**double** - [-]m.dddddde±xx
g, **G**	**double** - usa %e o %E se l'esponente è minore di -4 o maggiore della precisione
p, **E**	**void *** - puntatore

Tabella 1.3: Caratteri di conversione.

```
char c ;
c = getchar();
putchar(c);
```

1.5.1 Esercizi

Esercizio 1

Scrivere un programma che dato un carattere scelto dall'utente visualizzi un rettangolo costruito usando lo stesso carattere. Ad esempio se l'utente sceglie il carattere "∗" si deve ottenere

```
************************
*                      *
*                      *
************************
```

Esercizio 2

Dichiarare ed inizializzare la seguente variabile $x = 12356332.34123222$. Visualizzarla usando un'appropriata stringa di riferimento in **printf**. Usare il carattere di conversione

f, per la visualizzazione come parte intera e parte frazionaria e il carattere di conversione **e** per la visualizzazione in notazione scientifica.

Esercizio 3

Scrivere un programma che calcoli il volume di un parallelepipedo rettangolo. La base, l'altezza e la profondità vengono scelte dall'utente e sono dati di input per il programma.

Esercizio 4

Date le seguenti operazioni, scrivere un programma che dichiari opportunamente le variabili e visualizzi i risultati delle operazioni

```
a = 6;
b = 'A';
c = 12331222;

d = 8.14245322;

e = ((c * d)^2)%10;

visualizzare a,b,c,d,e
```

Esercizio 5 [✓]

Individuare gli errori nel seguente programma

```
#include <stdio.h>

void main () {

  const unsigned char a = 255u;
  const short b = 512;

  const unsigned long c = 10000000ul;

  unsigned short x;

  unsigned short y;
```

```
  x = a-b;

  y = (b-a) * c;

  printf("x risulta = %u\n",x);
  printf("y risulta = %u\n",y);
}
```

Esercizio 6 [✓]

Individuare gli errori nel seguente programma

```
#include <stdio.h>

void main () {

  const unsigned char a = 255u;
  const short b = 512;

  const unsigned long c = 10000000ul;

  unsigned short x;

  unsigned short y;

  x = a-b;

  y = (b-a) * c;

  printf("x risulta = %u\n",x);
  printf("y risulta = %u\n",y);
}
```

Esercizio 7

Scrivere ed eseguire il seguente programma

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

```
void main()
 {
 const float a = 8.32453u;
 int b, c;

 scanf("%d",&c);

b = a/c;

 printf ("divisione=%f  \n", b);
}
```

Verificare e correggere gli errori.

Esercizio 8

Ottimizzare e correggere il seguente programma

```
#include <stdio.h>

void main()
 {
 float a;
 int b, c, d;

 a = 3;

 scanf("%d",&b);
 scanf("%d",&c);

 d = (a/c) * b;

 printf ("b=%f  \n", b);
 printf ("c=%f  \n", c);
 printf ("operazione=%f  \n", d);

}
```

Esercizio 9 [✓]

Completare e correggere il seguente programma

```
#include <stdio.h>


void main()
 {
  const ... a = 4194937293;
  const short b = 32765;


  ... x;


  x = a * b ;


  printf("x risulta = %15.0 ... \n", x);
}
```

Esercizio 10 [✓]

Scrivere un programma che calcoli e stampi il risultato della seguente formula

$$\frac{x^2+3x}{0.5+x}+\frac{0.1x+1}{x^3+x}*5 \tag{1.1}$$

dove il coefficiente x è inserito dall'utente ed è il dato di input del programma.

1.6 Puntatori in C

Un puntatore è una variabile che contiene l'indirizzo di un'altra variabile. L'operatore & fornisce l'indirizzo di un oggetto e quindi ad esempio l'istruzione

```
p = & c;
```

assegna l'indirizzo di **c** alla variabile **p**. L'effetto di questa assegnazione viene sintetizzato dicendo che **p** punta a **c**.
L'operatore unario $*$ è l'operatore di **dereferenziazione** o **deriferimento**; quando viene applicato a un puntatore, esso accede all'oggetto puntato. L'esempio mostra come si dichiara un puntatore e come può essere usato

```
int  x=1, y=2; int *ip;
/* ip è un puntatore ad un intero */

ip= &x;                   /* ora ip punta a x */

y = *ip; /* ora y vale 1 */

*ip = 0;                   /* ora x vale 0 */
```

Inoltre possiamo dichiarare un puntatore a puntatore nel seguente modo

```
char **p;
```

In questo caso, ***p** identifica un indirizzo e ****p** un dato.

1.6.1 Esercizi

Esercizio 1

Determinare, senza usare il compilatore, il risultato del seguente programma C

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

int main() {

  int i = 3, j = 5, *p = &i, *r;

  printf("%d",*(r = &j) *= *p);

return 0; }
```

Esercizio 2

Determinare, senza usare il compilatore, il risultato del seguente programma C

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

int main() {
```

```
  int i = 3, j = 5, *p = &i,  *q = &j;

  printf("%d", 7**p / * q + 7);

return 0; }
```

Esercizio 3 [✓]

Determinare, senza usare il compilatore, il risultato del seguente programma C

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

int main() {

 short i = 9, j = 3, *a = &i,  *b, *c;

  b = &j;
  ++*a;
  *c = i**b;

  printf("%d",*c);

return 0; }
```

Esercizio 4

Determinare, senza usare il compilatore, il risultato del seguente programma C

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

int main() {

 short i = 10, j = 8, *a = &i, *c = &j, *b;

  double x = 2.2;

  b = a;
```

```
  i = 0;

  *c += (int)x**b ;

  printf("%d",*c);

return 0; }
```

Esercizio 5 [✓]

Individuare gli errori nel seguente codice C

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

int main() {

  int i = 3, j = 5, p = &i,  q = &j;

  printf("d", 2**p + 3/*q );

return 0; }
```

Capitolo 2

Costrutti di Controllo

Le istruzioni di controllo del flusso (*costrutti di controllo*) specificano l'ordine secondo il quale vengono eseguite sequenze di istruzioni. In questo Capitolo motriamo le più comuni istruzioni per il controllo del flusso in C.

2.1 IF-ELSE

L'istruzione **if-else** (*costrutto di selezione*) viene usata per esprimere una scelta tra due sequenze di istruzioni, in base ad una condizione (descritta da un predicato logico). La sua sintassi è

```
if (<predicato>) {
  <corpo del then>
}
 else
{
  <corpo dell'else>
}
```

Ogni parte delle istruzioni è un blocco ed è quindi racchiusa da parentesi graffe. Se nel blocco c'è una singola istruzione, le parentesi graffe si possono omettere. La parte associata all'**else** è opzionale, cioè è possibile avere una istruzione **if** senza **else** (**if semplificato**). I costrutti possono essere anche **annidati** [1]. Il seguente codice mostra un esempio di costrutto **if-else**

[1] **Costrutti annidati**, o **nidificati**, è un neologismo tecnico (dall'inglese **nested**) che indica la situazione in cui un costrutto appare all'interno di un altro costrutto della stessa tipologia.

```
if (n > 0)
   if (a > b)
      z = a;
else
   {z = b; n=n+1;}
```

2.2 SWITCH

L'istruzione **switch** è una struttura di scelta plurima. L'istruzione (a valore **int** o **char**) assume valore all'interno di un certo insieme di costanti ed esegue la sequenza di istruzioni associata a quel valore. Ad esempio nel seguente caso

```
switch (c) {
   case '0' : n=n+1; break;
   case '\t': a+=10; break;
   default: i++; break;
}
```

se la variabile **char c** ha valore '0', si esegue la sequenza dopo **case** '0', etc. L'istruzione **break** provoca l'uscita dal blocco.

2.3 Operatori logici

Per costruire il predicato delle istruzioni di controllo si ricorre agli operatori logici. Gli operatori logici relazionali, di uguaglianza e di connessione sono

- $>$ maggiore
- $>=$ maggiore o uguale
- $<$ minore
- $<=$ minore o uguale
- $==$ minore o uguale
- $!=$ minore o uguale

- ! negazione (not)
- && congiunzione (and)
- || disgiunzione (or)

2.4 CICLI

Per **ciclo** (o **iterazione** o **ripetizione**) si intende il processo in cui viene ripetuto ciclicamente l'esecuzione di una sequenza di istruzioni. In C le istruzioni **while**, **for** e **do-while** permettono di ripetere azioni. La sintassi del ciclo **while** è la seguente

```
while (<predicato>) {
  <corpo del ciclo>
}
```

Ad esempio il seguente programma permette di visualizzare i numeri da 1 a 10

```
#include <stdio.h>

void main() {
 int i=1;
 while(i<=10) {
    printf("%3d",i);
    i++;
 }
}
```

L'istruzione **for** viene utilizzata per ripetere l'esecuzione di una sequenza di istruzioni. La sintassi è la seguente

```
for (expr1; expr2; expr3) {
  <corpo del ciclo>
}
```

Il seguente programma permette di visualizzare i numeri da 1 a 10 usando il ciclo **for**

```
#include <stdio.h>

void main() {
```

```
 int i;
 for(i=1; i<=10 ; i++)
  {
   printf("%3d",i);
  }
}
```

Infine l'istruzione **do-while** può essere vista come una variante dell'istruzione **while**. La sintassi è la seguente

```
do {
  <corpo del ciclo>

} while (<predicato>);
```

In questo caso il programma che visualizza i numeri da 1 a 10 risulta

```
#include <stdio.h> void main() {
 int i=1;
 do{
    printf("%3d",i);
    i++;
 } while(i<=10);
}
```

2.5 Esercizi

Esercizio 1

Fornire espressioni logiche equivalenti alle seguenti senza utilizzare le negazioni

```
!(a > b)

!(a <= b && c <= d)

!(a + 1 == b + 1)

!(a < 1 || b < 2 && c < 3)
```

Esercizio 2

Dato il seguente codice dire che cosa viene visualizzato

```
int i = 7, j = 7;

if (i == 1)
   if (j == 2)
      printf("%d\n", i = i + j);
else
   printf("%d \n", i = i + j);
printf("%d\n",i);
```

Esercizio 3

Dato il seguente codice sostituire il ciclo **for** con un ciclo **do-while**

```
int i,j; double x,s;

for(i=0;i<20;i++) {
 scanf("%f",&x);
 s +=x;
}

if (s != 10) printf("Risultato");
```

Esercizio 4 [✓]

Dato il seguente codice sostituire il ciclo **while** con un ciclo **do-while** (EOF = Ctrl-Z)

```
char c;

int lower, total;

while(( c = getchar()) != EOF){
  if (c >= 'a' && c <= 'z')
    ++lower;
  total++;
}
```

Esercizio 5

Dire che cosa fa il seguente codice

```
int i=0,a=0;


do{
   a+=i;
   scanf("%d", &i);
} while (i>0);
```

Esercizio 6

Scrivere un programma che data una sequenza di numeri inseriti dall'utente visualizzi quanti di loro sono negativi e quanti sono positivi.

Esercizio 7

Scrivere un programma che, data una sequenza di caratteri, visualizzi quanti corrispondono a delle vocali.

Esercizio 8

Scrivere un programma che simuli un *salvadanaio*. L'utente può inserire e prelevare soldi. Visualizzare *salvadanaio vuoto* se non ci sono soldi nel salvadanaio. Il numero di operazioni di inserimento e prelievo sono decise dall'utente.

Esercizio 9 [✓]

Scrivere un programma in cui l'utente deve indovinare, in due fasi, con un massimo di 10 tentativi, un codice segreto. Il codice segreto è un intero definito dal programmatore con valore tra 0 e 100. Se l'utente indovina il codice nella prima fase, allora nella seconda fase l'utente deve indovinare un secondo codice segreto composto da una vocale (controllare che l'utente non inserisca una consonante).

Esercizio 10 [✓]

Scrivere un programma per simulare l'inserimento di un **PIN** per il telefonino. Nella prima fase viene chiesto all'utente di inserire un codice di lunghezza 5. Nella seconda fase l'utente inserisce il codice e ha al massimo 3 tentativi per indovinarlo.

Capitolo 3

Funzioni e Procedure

Le funzioni consentono di organizzare programmi complessi in moduli più semplici, sfruttabili anche singolarmente per la risoluzione di problemi diversi. Se strutturate nel modo corretto, le funzioni rendono invisibili al resto del programma i dettagli implementativi che non è necessario conoscere in modo esplicito.

3.1 Funzioni

Un programma è costituito da uno o più file, ognuno dei quali contiene zero o più funzioni, di cui una, e solo una, di nome **main**.
Come l'equivalente concetto matematico, la funzione in programmazione ha un nome, un insieme di dati di input e un output. Una definizione di funzione in C ha la seguente forma generale

```
<tipo output> <nome function>(<tipo> <parametro>, <tipo> <parametro>, ... ) {
<parte dichiarativa>
<parte esecutiva>
}
```

Ad esempio consideriamo il programma che calcola la lunghezza della circonferenza di un cerchio dato il suo raggio. Scegliamo di costruire una funzione **circon** che restituisce il calcolo di una circonferenza dato la lunghezza del raggio. Il programma completo risulta

```
#include <stdio.h>
float circon(float raggio);


void main () {
  float raggio, circonferenza;
```

```
  printf ("Inserire il raggio : ");
  scanf("%f",&raggio);
  circonferenza = circon(raggio);
  printf ("circonferenza=%f\n",circonferenza);
}

float circon(float raggio) {
  const float pi_greco = 3.1415926F;
  float risultato;
  risultato = 2.0F*pi_greco*raggio;
  return risultato;
}
```

Per utilizzare una funzione è necessario dichiarare un prototipo prima del **main**, scrivere una chiamata alla funzione all'interno del **main** e poi scrivere la descrizione della funzione in questo caso posta dopo il **main**.
Generalmente per rendere più leggibile un programma è utile suddividerlo su più file. Facendo riferimento all'esempio precedente, è possibile costruire un file **main.c** contenente il **main**, un file **circon.c** contenente la descrizione della funzione e un file header **mio_header.h** contenente il suo prototipo. Il file header è utile anche per la dichiarazione di variabili globali.
Il nostro programma viene quindi suddiviso in 3 file. Il file **main.c** risulta

```
#include "mio_header.h"

void main () {
  float raggio, circonferenza;
  printf ("Inserire il raggio : ");
  scanf("%f",&raggio);
  circonferenza = circon(raggio);
  printf ("circonferenza=%f\n",circonferenza);
}
```

Il file **mio_header.h**

```
#include <stdio.h>
float circon(float raggio);
```

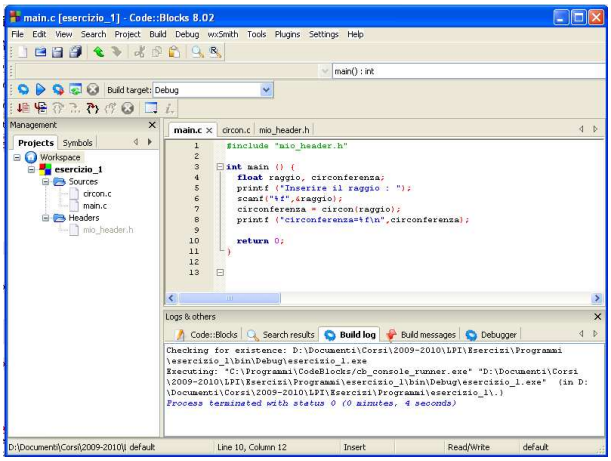

Figura 3.1: Esempio di progetto.

e infine il file **circon.c**

```
float circon(float raggio) {
  const float pi_greco = 3.1415926F;
  float risultato;
  risultato = 2.0F*pi_greco*raggio;
  return risultato;
}
```

Per poter compilare e linkare i diversi file basta creare un progetto contenente tutti e tre i file. In Figura 3.1 viene presentato lo schema di questo progetto.

3.2 Procedure

In C una procedura viene realizzata sempre mediante una funzione che restituisce più valori. Per poter implementare una procedura, il passaggio dei parametri di output alla funzione deve avvenire per **indirizzo**. Il passaggio dei parametri che abbiamo visto precedentemente è detto per **valore**. Nel passaggio per valore, la funzione considera il parametro (formale) e il corrispondente argomento di chiamata come due entità separate e assegna al parametro il valore del corrispondente argomento, cioè crea una copia esatta dell'argomento in input e lavora su questa. Se invece l'argomento viene passato per indirizzo, una eventuale modifica all'interno del corpo della funzione del parametro modifica

anche il valore dell'argomento corrispondente, cioè si ripercuote anche all'interno della funzione chiamante.
Ritornando al nostro esempio sulla funzione **circon** allora questa può essere definita come procedura nel seguente modo

```
void circ(float r, float *c) {
 const float pi_greco = 3.1415926F;
 *c = 2.0F*pi_greco*r;
}
```

La chiamata all'interno del **main** risulta invece

```
#include<stdio.h>

void circ(float ,float *);

void main () {
 float raggio, circonferenza;
 printf("inserire il raggio: ");
 scanf("%f",&raggio);
 circ(raggio, &circonferenza);
 printf("circonferenza =%f\n",circonferenza);
}
```

3.3 Classi di memorizzazione

Tutte le variabili e funzioni del C hanno due attributi: il *tipo* e la *classe di memorizzazione*. Le quattro classi di memorizzazione disponibili sono: **auto**, **extern**, **register** e **static**.

3.3.1 Classe auto

Le variabili dichiarate all'interno del corpo di una funzione sono per default automatiche (**classe auto**). Esplicitamente potremmo dichiarare

```
auto int a, b, c;
auto float f;
```

3.3.2 Classe esterna

Quando una variabile viene dichiarata al di fuori di una funzione, a essa viene assegnata memoria permanente e la sua classe di memorizzazione è **extern**. Un metodo per trasferire le informazioni tra blocchi e le funzioni consiste nell'impiego di variabili esterne. Le variabili **extern** sono anche dette **variabili globali**, perchè sono variabili (cioè utilizzabili) da tutte le funzioni che costituiscono il programma.

3.3.3 Classe registro

La classe di memorizzazione **register** richiede al compilatore che la variabile corrispondente sia memorizzata in registri di memoria ad alta velocità. L'obiettivo è l'aumento della velocità d'esecuzione del programma. Un esempio di dichiarazione è la seguente

```
register int i;
```

3.3.4 Classe statica

La dichiarazione statica ha due utilizzi distinti. Il primo permette di avere una variabile locale che mantiene il valore precedente al rientro in un blocco, a differenza delle variabili automatiche, che prendono il valore all'uscita e devono essere nuovamente iniziailizzate. Come esempio dell'impiego di **static** per mantenere il valore, viene presentata uno schema di funzione che si comporta in modo diverso a seconda di quante volte è già stata chiamata

```
void f(void) {

 static int cnt = 0;

 ++cnt;

 if(cnt % 2 == 0)
    ...
 else
    ...
}
```

Alla prima chiamata della funzione, la variabile **cnt** viene inizializzata a zero e all'uscita dalla funzione il valore di **cnt** viene mantenuto in memoria. Alla successiva chiamata

della funzione, **cnt** non viene più inizializzata, ma mantiene invece il valore che aveva alla precedente chiamata della funzione.
Il secondo utilizzo è legato alle dichiarazioni esterne. Le variabili statiche esterne sono variabili esterne con visibilità ristretta. La loro visibilità è il resto del file sorgente in cui sono dichiarate.

3.4 Il main

Il **main** è la funzione principale che permette l'esecuzione del programma. Ogni programma deve contenere una (e una sola) funzione main. Esso può essere usato sia come procedura

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

che come una funzione

```
int main () { ...


return 0; }
```

Esso può accettare anche due argomenti di input

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


return 0; }
```

dove **argc** è il numero degli argomenti sulla linea di comando passati al programma e **argv** è un vettore di stringhe di caratteri che contengono gli argomenti (cioè le parole digitate sulla linea di comando al momento dell'esecuzione del programma). Noi non faremo uso di questa possibilità.

3.5 Numeri casuali

Per la generazione di numeri pseudocasuali in C viene usata la funzione **rand()**. Essa genera solo numeri interi (tipo int) che appartengono all'intervallo $[0, \mathbf{RAND\_MAX}]$

e che sono uniformemente distribuiti in tale intervallo. **RAND_MAX** è definito nella libreria **stdlib.h**. Un semplice utilizzo è il seguente

```
int nc;
nc = rand();
```

Per generare a ogni esecuzione del programma una diversa sequenza di numeri pseudocasuali è necessario usare la funzione **srand()**

```
srand(nuovo_seed);
```

Ad esempio per una generazione automatica di un diverso seme si può usare la seguente chiamata a **srand**

```
srand((unsigned int)time(0));
```

dove **time** è una funzione della libreria **time.h**, che restituisce il valore dell'ora riportato dall'orologio interno del calcolatore.
Per avere dei numeri interi casuali uniformemente distribuiti in un qualsiasi intervallo $[A, B]$ viene usata l'operazione di modulo

```
int nc;


nc = A + rand()%(B + 1 - A);
```

Per generare numeri pseudocasuali di tipo reale uniformemente distribuiti nell'intervallo $[a, b]$ usiamo invece la seguente relazione

```
nc_f = a+(b-a) * (float)rand()/(float)RAND_MAX;
```

3.6 Esercizi

Esercizio 1 [✓]

Dato il seguente programma costruire due funzioni: una che restituisca la distanza tra due punti e l'altra per la traslazione di un punto. Suddividere il programma su più file: file per il main, il file header ed un file con le funzioni.

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

```
int main () {
  float x1,y1,x2,y2,d,t1,t2,a=2,b=10;
  printf ("Inserire le coordinate del primo punto (float) : ");
  scanf("%f%f",&x1,&y1);
  printf ("Inserire le coordinate del secondo punto (float) : ");
  scanf("%f%f",&x2,&y2);
  t1 = x1 + a;
  t1 = y1 + b;
  d = sqrt(x1*x2+y1*y2);
  printf ("la distanza dei punti è:%f\n",d);
  return 0;
}
```

Esercizio 2 [✓]

Qual è il risultato dell'esecuzione del seguente programma

```
#include <stdio.h>

void f(void);

int main () {
  int i;
  for(i=0;i<=10;i++) f();
  return 0;
}

void f(void) {
 static int cnt = 1;
 ++cnt;
 if (cnt % 2 == 0)
      printf("cnt:%d\n",cnt);
 else
      printf("cnt:%d\n",cnt);
}
```

Esercizio 3 [✓]

Qual è il risultato dell'esecuzione del seguente programma supponendo che l'utente inserisca ogni volta 0

```
#include <stdio.h>

int f(int);

int main () {
  int i;
  int a, s = 0;
  for(i=0;i<3;i++)
  {
    printf("Dammi il %d numero\n", i+1);
    scanf("%d", &a);
    s += f(a);
  }
  printf("Il risultato e': %d", s);
  return 0;
}

int f(int a) { static int cnt = 1;
 ++cnt;
 if (cnt % 2 == 0) return a+cnt;
}
```

Esercizio 4 [✓]

Riscrivere il seguente programma utilizzando una procedura

```
#include <stdio.h>

int f(int);

int main () {
  int i;
  int a, s = 0;
  for(i=0;i<3;i++)
  {
```

```
    printf("Dammi il %d numero\n", i+1);
    scanf("%d", &a);
    s += f(a);
  }
  printf("Il risultato e': %d", s);
  return 0;
}

int f(int a)
{
static int cnt = 1;
 ++cnt;
 if (cnt % 2 == 0) return a+cnt;
}
```

Esercizio 5

Riscrivere la seguente funzione sotto forma di procedura

```
void f(int *a, int *b, int *c) {
 if (*a <= *b) *c +=*a;
  else if (*a**b != 10) *c = 0;
}
```

Esercizio 6

Scrivere una funzione che permetta di calcolare il minimo e il massimo di tre numeri.

Esercizio 7 [✓]

Scrivere una funzione che calcoli la seguente formula

$$(x^3 + 3x + 5)/(8x + 1) \tag{3.1}$$

Scrivere successivamente un programma che calcoli i valori della funzione per $x = 0$, 2, 4, 6, ..., 20.

Esercizio 8 [✓]

Scrivere una procedura che dati due valori x e y calcoli i valori delle seguenti funzioni matematiche

$$(x^3 + 3x + 5)/(8x + 1) \tag{3.2}$$

$$(y^4)/(1 + y) \tag{3.3}$$

Scrivere successivamente un programma che calcoli e visualizzi il risultato delle funzioni precedenti nei punti $(0, 0)$, $(1, 1)$, $(3, 3)$, $(5, 5)$, ..., $(9, 9)$.

Esercizio 9 [✓]

Descrivere il significato della seguente procedura e trasformarla in funzione

```
void f(int m, int n, int *p)
{
 int i; long p = 1;
 for(i = 1; i < = n; ++i) p *= m; }
```

Scrivere successivamente un programma che calcoli e visualizzi i suoi risultati in una tabella, usando i seguenti valori di **m** ed **n** : $0-0$, $0-1$, $0-2$, $1-0$, $1-1$, $1-2$, $2-0$, $2-1$, $2-2$.

Esercizio 10 [✓]

Scrivere un programma che simuli un *salvadanaio*. L'utente può inserire e prelevare soldi. Visualizzare *salvadanaio vuoto* se non ci sono soldi. Il numero di operazioni di inserimento e prelievo sono decise dall'utente. L'inserimento e il prelievo devono avvenire con la chiamata a due procedure differenti.

Esercizio 11

Scrivere due funzioni che calcolino il massimo e il minimo di un insieme di valori inseriti dall'utente. Scrivere successivamente un programma che verifichi se la loro differenza è minore di un certo valore inserito.

Esercizio 12

Simulare per dieci volte il lancio di un dado e visualizzare la somma dei risultati di tutti i lanci.

Esercizio 13 [✓]

Due giocatori si sfidano lanciando un "dado truccato". Il dado ha dei valori interi nell'intervallo $[5, 15]$. A ogni turno vince il giocatore che ottiene un punteggio maggiore. In caso di parità il punto viene assegnato a entrambi. Simulare 10 sfide e visualizzare il giocatore che vince più volte.

Esercizio 14 [✓]

Scrivere una procedura che dato una ascissa x calcoli la seguente funzione matematica

$$(x^3 + 3x + 5)/(8x + 1) \tag{3.4}$$

Scrivere successivamente un programma che calcoli e visualizzi il risultato della funzione in 20 ascisse casuali scelte nell'intervallo $[0, 1]$.

Capitolo 4

Strutture dati

In C esistono due tipi di dati strutturati, il tipo array e il tipo struttura (struct). Il tipo array viene usato per rappresentare un insieme di dati omogenei, il tipo struttura viene utilizzato per rappresentare un insieme di dati eterogenei.

4.1 Array

In molti problemi devono essere gestiti insiemi di dati omogenei. Ad esempio, per gestire 100 numeri interi abbiamo bisogno di 100 variabili. Se usiamo un struttura di tipo array possiamo immaginare di trattarli come una unica identità. In realtà i 100 valori sono memorizzati in voci di memoria successive e possiamo fare riferimento a loro tramite un indice (o più indici), oppure mediante in indirizzo. Per dichiarare un array monodimensionale la sintassi è la seguente

```
<tipo> <nome_array>[<size>];
```

Ad esempio un array di 100 interi è dichiarato nel seguente modo

```
int A[100];   // lo spazio va da A[0] fino A[99]
```

Per inizializzare un array possiamo usare direttamente la seguente espressione

```
int a[]={22,-4,9,11,-6};
```

Per inserire elementi nell'array, invece, possiamo considerare un ciclo **for**

```
for(i=0; i<n; i++)
   scanf("%d",&a[i]);
```

Lo stesso può avvenire per la visualizzazione

```
for(i=0; i<n; i++)
   printf("%d",a[i]);
```

Un array bidimensionale (matrice) viene dichiarato secondo il seguente criterio

```
<tipo> <nome_array>[<righe>][<col>];
```

Ad esempio possiamo considerare le seguenti dichiarazioni

```
float matrice[5][5];
int tabella[15][10];
char pagina[40][80];
```

In questo caso l'inizializzazione può avvenire nel seguente modo

```
int matrice[2][3]= {{21,16,14},
                    {12,22,30}};
```

Per l'inserimento dei dati all'interno di una matrice di m righe e n colonne vengono usati due cicli for

```
for(i=0; i< m; i++)
  for(j=0; j < n; j++)
    scanf("%d", &matrice[i][j]);
```

In modo analogo possiamo costruire dei cicli annidati per visualizzare i valori della matrice.

4.1.1 Array e puntatori

In C, la relazione esistente tra puntatori e vettori è molto stretta. Qualsiasi operazione effettuabile indicizzando un vettore può essere eseguita anche tramite puntatori. In generale il programma che utilizza i puntatori è più efficiente.
Supponiamo di avere un array di 10 elementi

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

Se **pa** è un puntatore a un intero dichiarato come

```
int *pa;
```

allora l'assegnazione

```
pa = &a[0];
```

oppure

```
pa = a;
```

fa in modo che **pa** punti all'elemento di indice zero di **a** e cioè **pa** contiene l'indirizzo base di **a**.
Per riferirsi agli elementi successivi, ad esempio $\mathbf{a}[i]$, possiamo considerare

```
*(pa + i)
```

Inoltre abbiamo che $*(\mathbf{a}+i)$ coincide con $\mathbf{a}[i]$. Infatti se consideriamo l'indirizzo di $\mathbf{a}[i]$, cioè $\&\mathbf{a}[i]$, abbiamo he questo coincide con $(\mathbf{a}+i)$, in quanto $(\mathbf{a}+i)$ denota l'indirizzo che si ottiene sommando i all'indirizzo base di a.
Il nome di un array è un puntatore all' indirizzo base dell'array. Il passaggio di un array come parametro di una function è per riferimento (per indirizzo) e non per valore. Se si usa un array come argomento (nella chiamata alla function) basta inserire il nome dell'array. L'array come parametro (nell'intestazione di function) deve essere un nome di array seguito da [ ], oppure, nel caso di array identificati da puntatori, deve avere un puntatore (al tipo dell'array).
Ad esempio possiamo avere

```
int a[10]={10,20,30,40,50,60,70,80,90,100};

f(a,10)

void f(int a[], int n) {
  int i;
  for (i=0;i<n;i++)
      printf("%4d",a[i]);
}
```

oppure

```
void f(int *a, int n) {
  int i;
  for (i=0;i<n;i++)
      printf("%4d",*(a + i));    /* in modo equivalente printf("%4d", a[i]); */
}
```

Anche gli array bidimensionali possono essere rappresentati tramite puntatori. Ad esempio consideriamo la seguente matrice A

```
float A[n_rig][n_col]
```

L'elemento $\mathbf{A}[i][j]$ può essere rappresentato tramite puntatori nel seguente modo

```
*(&A[0][0]+n_col*i+j)
oppure
*(A + n_col*i + j)
```

Per il passaggio di una matrice a una funzione deve essere esplicitamente specificato il numero delle colonne di un parametro array, affinchè il compilatore possa generare la mappa di memorizzazione corretta. Non è necessario conoscere il numero di righe. Ad esempio possiamo considerare la seguente funzione di visualizzazione

```
 void  visualizza_a2DI(int a[][100], int n,int m)
{
 int i,j;
 for (i=0;i<n;i++)
   {
     for (j=0;j<m;j++)
         printf("%5d", a[i][j]);
     printf("\n");
   }
}
```

Facendo uso dei puntatori la funzione diventa

```
void visualizza_a2DIp(int *pa, int n, int m) {
 int i,j;
 for (i=0;i<n;i++)
   {
     for (j=0;j<m;j++)
         printf("%5d", *(pa+m*i+j));
     printf("\n");
   }
}
```

4.1.2 Aritmetica degli indirizzi

L'integrazione fra puntatori, array e aritmetica degli indirizzi è uno dei punti forza del linguaggio C. Per allocare memoria dinamicamente in C viene usata la funzione **malloc()** della libreria **stdlib.h**. L'oggetto che ritorna **malloc** è un puntatore a **void**. Possiamo forzare esplicitamente questo puntatore a un tipo desiderato usando un **cast**. La sintassi risulta

```
void *malloc(size_t size)
```

che restituisce un puntatore allo spazio per un oggetto di ampiezza pari a **size** e lo spazio non è inizializzato.
Ad esempio

```
double *p;
int n = 10;
p = (double *) malloc(sizeof(double) * n);
```

La funzione **malloc** ritorna **NULL** se lo spazio richiesto non è disponibile. La memoria ottenuta tramite la chiamata **malloc** può essere liberata usando la funzione **free**.

```
free(p);
```

Della libreria **stdlib.h** è possibile usare anche la funzione **calloc()**

```
void *calloc(size_t nobj, size_t size)
```

che restituisce un puntatore allo spazio per un array di **nobj** oggetti di ampiezza **size**. Lo spazio è inizializzato ad una serie di zeri.

4.1.3 Esercizi

Esercizio 1

Descrivere il comportamento della seguente funzione

```
int f(double *s, int n, int m)
{
 int i, a=0;

  for(i = 0; i < m; i++) a += *(s + m + i);
```

```
 return a;
}
```

Esercizio 2

Correggere e ottimizzare la seguente funzione

```
int f(double *a[10][], int n, int m)
{
 int i;

 int s=0;

 for(i = 0; i < m; i++) s *=*(a + m*j);

 return a;
}
```

Esercizio 3 [✓]

Scrivere una funzione che prende in input due array di dimensioni n ed m, rispettivamente, e restituisce un array composto dalla concatenazione dei due array in modo tale che nelle posizioni dispari sono contenuti i valori del primo array e nelle posizioni pari quelle del secondo.

Esercizio 4 [✓]

Scrivere una funzione che prende in input una matrice e restituisca in output la matrice trasposta (ovvero la matrice in cui la j-esima colonna è la j-esima riga della matrice data, per $j = 1, 2, \ldots, n$).

Esercizio 5 [✓]

Scrivere una funzione che prende in input due matrici e restituisce come output una matrice che corrisponde alla somma delle due matrici. Usare la notazione a puntatori.

Esercizio 6 [✓]

Scrivere una funzione che prende in input la dimensione di una matrice e restituisce in output il puntatore alla matrice allocata dinamicamente.

Esercizio 7 [✓]

Sviluppare una function C che, dati come parametri di input un array 2D di **float**, il numero delle righe, il numero delle colonne, un **int p** e un **int q** (con **int p** minore di **int q**), determina e restituisce come parametro di output la somma degli elementi che hanno gli indici di riga e di colonna compresi tra **int p** e **int q**.

Esercizio 8 [✓]

Sviluppare una function C che, dati come parametri di input un array 2D di **int**, il numero delle righe, il numero delle colonne e un **int k**, determina e restituisce come parametro di output la somma degli elementi della **k**-sima riga.

Esercizio 9 [✓]

Sviluppare una function C che, dati come parametri di input un array 2D di **int**, il numero delle righe, il numero delle colonne e un **int k**, determina e restituisce come parametro di output la somma degli elementi della **k**-sima colonna.

4.2 Tipi derivati

In C si può dare un nome (identificatore) a un tipo esistente. Il nome è utilizzabile come un nuovo tipo e il nuovo tipo può essere utilizzato per dichiarare variabili o funzioni. La sintassi è la sguente

```
typedef <tipo_esistente> <nuovo_tipo>;
```

Come esempio possiamo considerare il seguente codice

```
typedef int Integer;
typedef char Character;
```

```
Integer i,j;
Character c, parola[5];
Character int2char(Integer);/*prototipo*/
```

Esiste inoltre il comando al preprocessore # **define** che permette di dichiarare delle costanti. Il precompilatore sostituisce, nel testo del programma sorgente, tutte le occorrenze di <identificatore> con <valore>. La sintassi è la seguente

```
#define <identificatore> <valore>
```

Come esempio possiamo considerare

```
#define PI_GRECO 3.1415926F
#define TRUE 1
#define FALSE 0
```

E' possibile inoltre definire delle macro

```
#define <nome_macro(x)> <istruzioni_macro>
```

come ad esempio

```
#define QUADRATO(x) ((x)*(x))
```

Infine in C è possibile creare un nuovo tipo enumerando tutti i suoi oggetti (tipo enumerativo)

```
enum <etichetta> {<elenco_oggetti>};
```

come ad esempio

```
enum giorno {Lu,Ma,Me,Gi,Ve,Sa,Do};
enum seme {picche,fiori,quadri,cuori};

enum giorno festa, riunione;
enum seme carta[40];
enum giorno chegiornoe(int);/*prototipo*/
```

4.2.1 Esercizi

Esercizio 1

Definire una macro che calcoli la funzione x^3.

Esercizio 2

Definire una macro che calcoli la funzione $3x^2 + 2$ e implementare una funzione che la richiami.

Esercizio 3

Dichiarare una struttura opportuna che identifichi i giorni della settimana e che per ogni giorno associ i valori dell'umidità: *bassa*, *media*, *alta*.

4.3 Stringhe

In C il tipo stringa non è direttamente definito. Il tipo stringa in C è realizzato come un array monodimensionale di elementi di tipo **char**. La fine della sequenza di caratteri della stringa è specificata esplicitamente dal carattere di fine stringa: '\ 0'. Il size dell'array deve tenere conto anche dello spazio necessario per memorizzare il carattere di fine stringa. Alcuni esempi di dichiarazioni sono i seguenti

```
char nome_Corso_Laurea[12];
char testo[1000];
char pagina_libro[1600];
char gene[250];
char genoma[100000000];
```

Alcuni esempi di inizializzazioni sono

```
char nome_Corso_Laurea[12]="Informatica";
char *nome_Corso_Laurea="Informatica";
```

4.3.1 I/0 delle stringhe

Per visualizzare una stringa usiamo la **printf** nei seguenti modi

```
char nome_Corso_Laurea[12]="Informatica";
printf("%s",nome_Corso_Laurea);
```

oppure

```
char *nome_Corso_Laurea="Informatica";
printf("%s",nome_Corso_Laurea);
```

Nella libreria **stdio** ci sono inoltre due funzioni per la lettura e scrittura di stringhe e sono **gets** e **puts**, rispettivamente. La funzione **gets()** ha un parametro di tipo puntatore a **char** (ovvero array di **char**) e restituisce un puntatore a **char**

```
char stringa[100];
gets(stringa);


char stringa[100];
stringa = gets();
```

La funzione **puts()**, invece, ha un parametro di tipo puntatore a char (ovvero array di **char**).

```
puts(stringa);
```

4.3.2 Libreria string.h

Nella libreria standard del C ci sono diverse funzioni per la manipolazione di stringhe. Ad esempio la funzione **strcat()** concatena una copia della stringa (puntata da) $str2$ alla stringa (puntata da) $str1$ e chiude la nuova sequenza di caratteri di $str1$ con il carattere nullo (terminatore di stringa). In dettaglio la chiamata alla funzione è la seguente

```
char *strcat(char *str1, char *str2)
```

La function restituisce $str1$ anche via return. La funzione **strchr()**, invece, restituisce un puntatore all'indirizzo del primo carattere della stringa (puntata da) str che risulta uguale al valore della variabile chiave se non viene trovata alcuna corrispondenza, la function restituisce un puntatore nullo. La sintassi è la seguente

```
char *strchr(char *str, char chiave)
```

La funzione **strcmp()** confronta, secondo le regole lessicografiche, le due stringhe (puntate da) $str1$, $str2$ e restituisce un intero il cui valore è minore di zero se $str1$ è minore di $str2$, zero se $str1$ è uguale a $str2$, maggiore di zero se $str1$ è maggiore di $str2$.

```
int strcmp(char *str1, char *str2)
```

La funzione **strncmp()** confronta, secondo le regole lessicografiche due sottostringhe, cioè due porzioni di due sottostringhe. Più precisamente **strncmp()** confronta i primi (al più) n caratteri delle due stringhe (puntate da) $str1$, $str2$ e restituisce un intero come nel caso di **strcmp()**

```
int strncmp(char *str1, char *str2, int n)
```

La funzione **strcpy()** copia la stringa (puntata da) *str*2 in *str*1

```
char *strcpy(char *str1, char *str2)
```

La funzione **strncpy()** copia una sottostringa di una stringa. Più precisamente copia i primi (al più) n caratteri della stringa (puntata da) *str*2 al posto dei primi n caratteri di *str*1.

```
char *strncpy(char *str1,char *str2,int n)
```

La funzione **strlen()**

```
unsigned int strlen(char *str)
```

restituisce la lunghezza della stringa (puntata da) *str*. La stringa deve essere chiusa dal carattere nullo (terminatore), che non viene conteggiato ai fini della determinazione della lunghezza.
Le seguenti funzioni

```
char *strlwr(char *str)
```

```
char *strupr(char *str)
```

convertono in minuscolo e in maiuscolo la stringa (puntata da) *str*.
La funzione **strncat()** opera su sottostringhe e concatena al più n caratteri della stringa (puntata da) *str*2 alla stringa *str*1 e chiude la sequenza di caratteri di *str*1 con il carattere nullo (terminatore).
Infine, **strstr()**

```
char *strstr(char *str1, char *str2)
```

restituisce un puntatore all'indirizzo del primo carattere della sottostringa della stringa (puntata da) *str*1 che risulta uguale alla stringa *str*2.

4.3.3 Esercizi

Esercizio 1

Descrivere il comportamento della seguente funzione e fare un esempio di esecuzione

```
int f(char *s)
{
 int n;

  for(n = 0; *s != '\0'; s++)
    n++

 return n;
}
```

Esercizio 2 [✓]

Modificare il seguente codice in modo tale che il risultato della visualizzazione sia zero

```
#include <string.h>
 #include <stdio.h>
 void main()
 {
    char *p;

    char a[] = "Musica";

    p = strchr("La mia Musica preferita non ha prezzo",'M');

    printf("%d\n", strncmp(p,a,7));

}
```

Esercizio 3 [✓]

Modificare il seguente codice in modo tale che il risultato della visualizzazione sia zero

```
#include <string.h>
 #include <stdio.h>
 void main()
 {
    char *p, b[10];

    char a[] = "PASS";
```

```
    p = strchr("inserisci la password",'l');

    strncpy(b,&p[3],strlen(p)-7);

    printf("%d\n", strcmp(a,strupr(b)));

}
```

Esercizio 4 [✓]

Modificare il seguente codice in modo tale che il risultato della visualizzazione sia la parola "**Comprimere**"

```
#include <string.h>
#include <stdio.h>
void main()
{
   char *p1, *p2;

   p1 = strchr(" Comp ",' ');

   p2 = strchr("rare attare rimere",'r');

   ...

}
```

Esercizio 5 [✓]

Scrivere un programma che simuli un sistema per determinare un codice segreto. Il programmatore definisce un codice di 5 lettere dell'alfabeto e in maniera casuale simula dei codici fino a che non trova quello esatto. Visualizzare il numero di prove effettuate.

4.4 Struct

Il tipo strutturato *record* viene realizzato in C mediante il meccanismo delle strutture. Il meccanismo delle strutture consente di dichiarare specifici tipi derivati e cioè aggregati

di variabili di tipo non omogeneo. In C la dichiarazione di una struttura (tipo **struct**) avviene nel seguente modo

```
struct <etichetta> {
    <tipo> <variabile>;
    <tipo> <variabile>;
    ...
    <tipo> <variabile>;
 };
```

Ad esempio una dichiarazione di struct è la seguente

```
struct punto_2D {
    double ascissa;
    double ordinata;
 };
```

che restituisce il tipo **struct punto_2D**, costituito dai campi ascissa (di tipo **double**) e ordinata (di tipo **double**), oppure

```
struct data {
    int giorno;
    char mese[10];
    int anno;
 };
```

che restituisce il tipo **struct data**, costituito dai campi giorno (di tipo **int**), mese (di tipo **array** di **char**), e anno (di tipo **int**).
Dobbiamo sottolineare comunque che la dichiarazione di una **struct** non alloca memoria. La dichiarazione di una variabile struttura avviene nel seguente modo

```
struct punto_2D vertice,estremo_sin;


struct data data_nascita,partenza;
```

Inoltre le variabili possono essere dichiarate insieme alla struttura

```
struct {
   int giorno;
   char mese[10];
   int anno;
 } data_nascita,partenza;
```

La dichiarazione di variabili struttura alloca memoria. Per la dichiarazione di un nuovo tipo di dato viene utilizzato il **typedef** ad esempio il seguente codice dichiarazione un nuovo tipo **Data**

```
typedef struct data Data;


Data data_nascita, partenza;
```

In C quando si usano i puntatori a variabili di tipo struct è possibile usare un operatore speciale ->. Ad esempio se consideriamo la seguente dichiarazione

```
struct punto_2D * ppunto,
                mio_punto={10.3,20.1};
 Carta * pcarta,mia_carta={1,cuori};
 ppunto = &mio_punto;
 pcarta = &mia_carta;
```

allora, per l'accesso ai campi attraverso puntatori si può scrivere

```
(*ppunto).ascissa = 100.2;
(*ppunto).ordinata = 200.1;
```

oppure

```
ppunto -> ascissa = 100.2;
ppunto -> ordinata = 200.1;
```

4.4.1 Esercizi

Esercizio 1

Individuare e correggere gli errori nel seguente programma

```
#include <string.h>
#include <stdio.h>
void main()
{


struct punto_2D {
   double ascissa;
   double ordinata;
```

```
};

struct punto_2D ppunto, mio_punto={0.3,0.1};
ppunto = mio_punto;
printf("%f",ppunto -> ascissa);
}
```

Esercizio 2 [✓]

Individuare e correggere gli errori nel seguente programma in modo che il risultato sia 18 e 30

```
#include <string.h>
#include <stdio.h>
struct studente{
   char *nome;
   char *cognome;
   short libretto[20];
 };

typedef struct studente Studente;

void main()
 {
Studente *a, b={"Mario","Rossi",{18,30}};
a = b;
printf("%d\n",a->libretto[0]);
printf("%d\n",a->libretto[1]);
}
```

Esercizio 3 [✓]

Sviluppare una function C che, dato come parametro di input un array di tipo **struct struttura** { **double a; float b** } e il suo size, restituisca come parametro di output un array composto solo dagli elementi di indice pari.

Esercizio 4 [✓]

Sviluppare una function C che, dati come parametri di input un array di tipo **struct struttura** { **double a; double b; double c**} e il suo size, determini e restituisca come parametri di output gli indici degli elementi uguali.

Esercizio 5 [✓]

Scrivere un programma per la gestione di un magazzino di prodotti elettronici. Il magazzino contiene al massimo 5 prodotti identificati da un codice, nome, dal prezzo e dalla quantità. Il magazzino viene gestito per 10 giorni complessivi. Ogni giorno l'utente può acquistare al massimo 5 prodotti se sono disponibili. Visualizzare il prodotto più venduto nei 10 giorni.

Esercizio 6 [✓]

Scrivere una funzione che ha come input i dati che identificano uno studente (nome, cognome, matricola) e che restituisce in output una struttura contenente i dati in input e il libretto univesritario con al massimo 20 esami.

Capitolo 5

Implementazione di algoritmi

In questo Capitolo presentiamo i programmi C che implementano i principali algoritmi sviluppati e analizzati durante il corso di Programmazione I. Tutti gli algoritmi sono realizzati come funzioni C.

5.1 Media, varianza, massimo e minimo

Iniziamo la nostra panoramica mostrando una funzione che calcola la **media** degli elementi di un array **a** di lunghezza **n**

```
double media(double a[], int n) {
 double s;
 int i;
 s = 0.0;
 for (i=0;i<n;i++)
    s = s + a[i];
 return s/n;
}
```

Il calcolo della **varianza** può essere effettuato conoscendo la media. La funzione è la seguente

```
double varianza(double a[], int n) {
 double s=0.0, m;
 int i;
 m = media(a,n);
 for (i=0;i<n;i++)
    s = s + pow(a[i]  m,2);
 return s / (n-1);
```

```
}
```

Quello che segue è il **main** del programma che richiama le due funzioni **media** e **varianza**

```
#include <stdio.h>
double media(double [], int);
double varianza(double [], int);
void legge_da_tastiera(double [], int );  /* lettura dei dati da tastiera ed inserimento nell'array */
void visualizza(double [], int ); /* visualizzazione dei dati dell'array sulla linea di comando */

void main () {
int n_elem;
double vet[100];
printf("inserire il numero di elementi (<=100):");
scanf("%d",&n_elem);
legge_da_tastiera(vet,n_elem);
media(vet,n_elem);
varianza(vet,n_elem);

printf("La media risulta %f - la varianza risulta %f", media, varianza);

}
```

Il seguente programma effettua il calcolo della media per ogni riga di una matrice bidimendionale

```
#include <stdio.h>
void visualizza_a2DD(double *,int,int);
void visualizza_a1(int [],int);

void main () {
double A[100][100],media_riga[100];
int i,j,n,m,somma;
printf("inserire il numero di righe (<=100):");
scanf("%d",&n);
printf("inserire il numero di colonne (<=100):");
scanf("%d",&m);
for (i=0;i<n;i++)
    for (j=0;j<m;j++)
      scanf("%d",&A[i][j]);
```

```
 for (i=0;i<n;i++)
     {
       somma = 0.0;
         for (j=0;j<m;j++){
        somma = somma+A[i][j];
        }
       media_riga[i] = somma/m;
      }
 printf ("le medie delle %d righe:\n",n);
 visualizza_a1(media_riga,n);
 printf ("l'array 2D e'\n");
 visualizza_a2DD(A,n,m);
}

void visualizza_a2DD(int *pa,int n_col, int n, int m) {
 int i,j;
 for (i=0;i<n;i++)
   {
     for (j=0;j<m;j++)
         printf("%5d", *(pa+n_col*i+j));
     printf("\n");
   }
}
```

La funzione del **massimo** determina l'elemento più grande di un insieme di valori di un array di size n

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

Notiamo che per calcolare il minimo basta inserire minore $<$ al posto del $>$ all'interno del costrutto **if**.

La seguente funzione ci permette di calcolare contemporaneamente il **massimo** e il **minimo** di un array

```
void max_min(int a[],int n, int *max, int *min)
{
int i;
*max = a[0];
*min = a[0];
for (i=1;i<n;i++)
   if(a[i] > *max)
         *max = a[i] ;
      else if(a[i] < *min)
            *min = a[i] ;
}
```

Se vogliamo determinare il **massimo** di una matrice bidimensionale, possiamo utilizzare le seguenti funzioni (con e senza puntatori)

```
int massimo_array2DI(int a[][100],int n,int m) {
 int max,i,j;
 max = a[0][0];
 for (i=0;i<n;i++)
      for (j=0;j<m;j++)
          if(a[i][j] > max)
            max = a[i][j];
 return max;
}
```

oppure

```
int massimo_array2DIp(int *pa,int n_col,int n,int m) {
 int max, i,j;
 max = *pa;
 for (i=0;i<n;i++)
      for (j=0;j<m;j++)
          if(*(pa+n_col*i+j) > max)
            max = *(pa+n_col*i+j);
 return max;
}
```

5.1.1 Esercizi

Esercizio 1 [✓]

Sviluppare una function C che, dati come parametri di input un array 2D di **int**, il numero delle righe, il numero delle colonne e un **int k**, determina e restituisce come parametro di output la somma degli elementi della **k**-sima riga.

Esercizio 2 [✓]

Sviluppare una function C che, dati come parametri di input un array 2D di **int**, il numero delle righe, il numero delle colonne e un **int k**, determina e restituisce come parametro di output la somma degli elementi della **k**-sima colonna.

Esercizio 3 [✓]

Sviluppare una function C che, dati come parametri di input un array 2D di **int**, il numero delle righe e il numero delle colonne, determina e restituisce come parametro di output il massimo tra le somme degli elementi di ogni riga.

Esercizio 4 [✓]

Sviluppare una function C che, dati come parametri di input un array 2D di double, il numero delle righe e il numero delle colonne, determina e restituisce come parametro di output il massimo tra le somme degli elementi di ogni colonna.

Esercizio 5 [✓]

Sviluppare una function C che, dati come parametri di input un array di int e il suo size, determina e restituisce come parametri di output l'indice di inizio e la lunghezza della più lunga sequenza ordinata (senso crescente) contenuta nell'array.

Esercizio 6 [✓]

Sviluppare una function C che, dato come parametro di input un array di tipo **struct punto** { **double x; double y** } e il suo size, determina e restituisce come parametro di output la massima distanza tra i punti.

Esercizio 7 [✓]

Sviluppare una function C che, dati come parametri di input un array di tipo **struct punto** {**double x; double y** } e il suo size, determina e restituisce come parametri di output gli indici dei due punti che hanno distanza minima tra loro.

Esercizio 8 [✓]

Sviluppare una function C che, dati come parametri di input un array di **int** e il suo size, determina e restituisce come parametro di output il secondo più grande elemento dell'array (N.B.: l'array non deve essere ordinato).

Esercizio 9 [✓]

Sviluppare una function C che, dati come parametri di input un array di **int**, il suo size e un **int k**, determina e restituisce come parametro di output il **k**-imo più grande elemento dell'array (N.B.: l'array non deve essere ordinato).

Esercizio 10 [✓]

Scrivere un programma per la gestione delle presenze a un corso unversitario. Ogni studente è indetificato dal nome, dal cognome e dalla matricola. Il corso ha 10 studenti ed è composto complessivamente da 15 lezioni. Permettere di visualizzare i nome degli studenti presenti ad una data lezione e lo studente che ha più presenze su tutte le lezioni.

5.2 Ricerca

L'algoritmo di **ricerca sequenziale** ricerca un dato (**chiave**) all'interno di un array monodimensionale. Costruiamo un funzione che dato un array e un valore restituisce 1 se l'elemento è stato trovato e 0 altrimenti. La funzione è chiamata **appartiene** e considera sia la chiave da cercare che l'array di dimensione **n**, di tipo **float**. Come è noto, la complessità dell'algoritmo di ricerca sequenziale è lineare ($O(n)$).

```
int appartiene(float chiave, float a[], int n)
 {
  int i;
```

```
 i = 0;
 for(i=0; i<n; i++)
     if (chiave == a[i])
        return 1;
 return 0;
}
```

Se l'array è ordinato,l'algoritmo iterativo di **ricerca binaria** effettua la ricerca con un tempo minore (come è noto l'algoritmo di ricerca binaria ha complessità di tempo logaritmica $O(\log(n))$). Questo algoritmo si basa sull'approccio del **divide-et-impera** che permette di sezionare il problema in diversi sottoproblemi, che sono simili quello originale ma coinvolgono un minore numero di dati. Risolvendo i sottoproblemi e combinando le loro soluzioni si ottiene la soluzione del problema di partenza. La funzione **ric_bin** implementa l'algoritmo di ricerca binaria

```
int ric_bin(char chiave,char elenco[],int n) {
 int mediano, primo = 0, ultimo = n-1;
 while(primo <= ultimo)
 {
  mediano = (primo + ultimo)/2;
  if(chiave == elenco[mediano])
     return mediano;
   else if(chiave < elenco[mediano])
            ultimo = mediano-1;
          else
            primo = mediano+1;
 }
 return -1;
}
```

Vogliamo effettuare una ricerca in un elenco ordinato di stringhe. Un elenco di stringhe è realizzato con un array di stringhe (array di puntatori a **char**). Per esempio

```
char *settimana[7] = {"domenica","giovedi","lunedi","martedi",
"mercoledi","sabato","venerdi"};
```

La funzione **ricerca_bin_S** implementa l'algoritmo di ricerca binaria di una chiave stringa in un array ordinato di stringhe

```
int ric_bin_S(char chiave[],char *elenco[],int n) {
 int mediano, primo = 0, ultimo = n-1;
 while(primo <= ultimo)
 {
  mediano = (primo+ultimo)/2;
  if(strcmp(chiave,elenco[mediano]) == 0)
     return mediano;
   else if(strcmp(chiave,elenco[mediano]) < 0)
            ultimo = mediano-1;
         else
            primo = mediano+1;
 }
 return -1;
}
```

5.2.1 Esercizi

Esercizio 1 [✓]

Scrivere un programma che dato un array di puntatori alla seguente struttura

```
struct studente {char *nome; char *cognome; int matricola;};
```

ricerchi un dato studente, utilizzando l'algoritmo di ricerca sequenziale.

Esercizio 2 [✓]

Scrivere un programma che dato un array di puntatori alla seguente struttura

```
struct studente {char *nome; char *cognome; int matricola;};
```

ricerchi un dato studente, utilizzando l'algoritmo di ricerca binaria.

5.3 Fusione di array ordinati

La funzione **fusione** implementa l'algoritmo di fusione di due array ordinati [1]

[1] Uno degli algoritmi efficienti per l'ordinamento di elementi di un array è il **MergeSort**. Esso ha una complessità di tempo del tipo logaritmica. Utilizza una tecnica di tipo ricorsiva e contemporaneamente la fusione di due array ordinati.

```
void fusioneC(char a[],int n_a,char b[],
            int n_b,char c[])
{
  int i_a=0,i_b=0,i_c=0;
  while (i_a < n_a && i_b <n_b)
    {
     if(a[i_a] < b[i_b])
         c[i_c++] = a[i_a++];
       else
         c[i_c++] = b[i_b++];
     }
  while (i_a < n_a)
     c[i_c++] = a[i_a++];
  while (i_b < n_b)
     c[i_c++] = b[i_b++];
}
```

5.3.1 Esercizi

Esercizio 1 [✓]

Scrivere una funzione che dati in input due array ordinati di puntatori alla seguente struttura

```
struct studente {char *nome; char *cognome; int matricola;};
```

restituisca in output l'array fusione dei due array. La fusione deve avvenire in base al campo matricola.

Esercizio 2 [✓]

Scrivere una funzione che dati in input due array ordinati di puntatori alla seguente struttura

```
struct studente {char *nome; char *cognome; int matricola;};
```

restituisca in output l'array fusione dei due array. La fusione deve avvenire in base al campo cognome.

5.4 Uguaglianza tra array

La funzione **uguaglianza** verifica se due array sono uguali (uguaglianza di tutte le componenti di posto uguale)

```
int uguaglianza(char a[],char b[],int n) {
 int i=1,uguale=1;
 while(uguale && i<n)
    {
     if(a[i] != b[i])
        uguale = 0;
     i++;
     }
 return uguale;
}
```

5.4.1 Esercizi

Esercizio 1 [✓]

Scrivere una funzione che dati in input due array di puntatori alla seguente struttura

```
struct prodotto {char *nome; int codice; double prezzo;};
```

verifichi se i due array di **struct** sono uguali.

Esercizio 2 [✓]

Scrivere una funzione che dato in input due array di strutture del seguente tipo

```
struct prodotto {char *nome; int codice; double prezzo;};
```

verifichi se gli array di **struct** sono uguali e in caso affermativo restituisca l'indice del prezzo più basso.

5.5 Funzioni su griglia

Nel campo del calcolo scientifico spesso è necessario calcolare i valori di una funzione matematica complessa su ascisse appartenenti a un intervallo. Per esempio, lo scopo

potrebbe essere quello di visualizzare l'andamento di questa funzione matematica. Per ottenere ciò in C bisogna calcolare un insieme di valori nell'intervallo stabilito, dopo aver fissato un passo di campionamento. Ad esempio consideriamo la funzione

$$\frac{3\sin(x)}{x^2+10} \tag{5.1}$$

definita nell'intervallo $[\pi, 3\pi]$. In questo intervallo vengono scelti 50 punti. Il programma seguente effettua il campionamento e visualizza i valori della funzione (5.1)

```
 #include <stdio.h>
 #include <math.h>
 void campiona(float fun(float),float,float,int ,float []); float mia_fun(float );
void main() {
  const float pi=3.1415926F;
  float mia_a,mio_b;
  int mio_n,i;
  float mio_f_c[100];
  mia_a = pi;
  mio_b = 3.F*pi;
  mio_n = 50;
  campiona(mia_fun,mia_a,mio_b,mio_n,mio_f_c);
  for (i=0;i<mio_n;i++)
    printf(" %f",mio_f_c[i]);
}

void campiona(float fun(float),float a,    float b,int n,float
f_c[]) {
 float passo,p_griglia;
 int i;
 passo = (b-a)/(n-1);
 p_griglia = a;
 for (i=0;i<n;i++)
   {
    f_c[i] = fun(p_griglia);
     p_griglia = a + (i+1)*passo;
   }
}

float mia_fun(float x) {
```

```
    return (3.0*sin(x))/(pow(x,2.)+10.0);
}
```

5.5.1 Esercizi

Esercizio 1 [✓]

Scrivere un programma che calcoli i valori di

$$\sin(x) \tag{5.2}$$

con x definito nell'intervallo $[0, 2\pi]$ e passo di campionamento 0.1.

Esercizio 2 [✓]

Scrivere un programma che calcoli i valori della seguente funzione di due variabili

$$x^2 + y^2 \tag{5.3}$$

definita nel quadrato $[0, 10]$x$[0, 10]$ con x ed y definite entrambe nell'intervallo $[0, 10]$ e con passo di campionamento 1. Determinare anche il numero totale di valori calcolati.

5.6 Elaborazione dei testi

L'elaborazione dei testi, ovvero di stringhe, è fondamentale in molti problemi applicativi. Per esempio, il trattamento testi (word-processor correttori ortografici estrazione parole chiave di un libro), antivirus (ricerca della firma del virus), siti web (classificazione dei siti in funzione di determinate parole rilevate nel contenuto delle pagine), bioinformatica (ricerca di determinate sequenze all'interno del DNA).

5.6.1 Testo

La seguente funzione determina il numero di caratteri, il numero di parole e il numero di linee di un testo.

```
#define FALSE 0
#define TRUE 1
```

```
 void conta_tutto(char *testo,int *conta_c,int *conta_p,
           int *conta_l)
{
  int i,in_p;
  i = 0; *conta_c = 0; *conta_p = 0; *conta_l = 0;
  in_p = FALSE;
  while (testo[i] != '\0')
  {
    (*conta_c)++;
    if (testo[i] == '\n')
     (*conta_l)++;
    if (testo[i] == ' '|| testo[i]== '\n' || testo[i]== '\t')
      in_p = FALSE;
      else if (in_p == FALSE)
           {
           in_p = TRUE;
             (*conta_p)++;
            }
    i++;
  }
  (*conta_l)++;
}
```

5.6.2 Matching

La funzione **string_matching** determina il numero di occorrenze di una stringa data (**chiave**) come sottostringa in un'altra stringa (algoritmo di **string matching**).

```
int string_matching(char chiave[],char testo []) {
  int n, m, i, conta_chiave;
  n = strlen(chiave);
  m = strlen(testo);
  conta_chiave = 0;
  for (i=0;i<m-n;i++)
     if(strncmp(chiave,&testo[i],n) == 0)
        conta_chiave++;
  return conta_chiave;
}
```

All'interno di un programma la funzione può essere richiamata con

```
char miotesto[100],miachiave[20];
...
n_occorrenze = string_matching(miachiave,miotesto);
```

La funzione **matching_migliore** implementa l'algoritmo di **best matching** che risolve il problema di determinare la sottostringa di una stringa che ha più caretteri in comune con una stringa chiave

```
int matching_migliore(char *chiave,char *testo) {
 int i,n,m,punteggio_max,punteggio,indice=0;
 n = strlen(chiave);
 m = strlen(testo);
 punteggio_max = 0;
 for (i=0;i<m-n;i++)
  {
     punteggio = punteggio_matching(chiave,&testo[i],n);
     if (punteggio > punteggio_max)
     {
      punteggio_max = punteggio;
        indice = i;
      }
   }
 return indice;
}

int punteggio_matching(char *a,char *b,int n) {
 int i, n_caratteri_uguali;
 n_caratteri_uguali = 0;
 for (i=0;i<n;i++)
     if (a[i] == b[i])
     n_caratteri_uguali++;
 return n_caratteri_uguali;
}
```

5.6.3 Esercizi

Esercizio 1 [✓]

Sviluppare una function C che, data come parametro di input una stringa che rappresenta un testo in italiano, determina e restituisce come parametro di output il numero di parole di tre lettere contenute nel testo. Nel testo le parole sono separate da un unico *spazio*.

Esercizio 2 [✓]

Sviluppare una function C che, data come parametro di input una stringa che rappresenta un testo in italiano, determina e restituisce come parametro di output il numero di parole che terminano in *are* contenute nel testo. Nel testo le parole sono separate da un unico *spazio*.

Esercizio 3 [✓]

Sviluppare una function C che, data come parametro di input una stringa che rappresenta un testo in italiano, determina e restituisce come parametro di output il numero di parole che iniziano con *a* e terminano con e contenute nel testo. Nel testo le parole sono separate da un unico *spazio*.

Esercizio 4 [✓]

Sviluppare una function C che, data come parametro di input una stringa che rappresenta un testo in italiano, determina e restituisce come parametro di output il numero delle parole contenute nel testo che hanno almeno 5 vocali. Nel testo le parole sono separate da un unico *spazio*.

Esercizio 5 [✓]

Sviluppare una function C che, data come parametro di input una stringa che rappresenta un testo in italiano, determina e restituisce come parametri di output la parola di lunghezza massima contenuta nel testo e la sua lunghezza. Nel testo le parole sono separate da un unico *spazio*.

Esercizio 6 [✓]

Sviluppare una function C che, data come parametro di input una stringa che rappresenta un testo in italiano, determina e restituisce come parametri di output la parola di lunghezza massima contenuta nel testo e la posizione di inizio della parola nella stringa. Nel testo le parole sono separate da un unico *spazio*.

Esercizio 7 [✓]

Sviluppare una function C che, dati come parametri di input un array di **char** e il suo size, determina e restituisce come parametro di output l'array (di size 21) del numero delle occorrenze delle 21 lettere dell'alfabeto italiano.

Esercizio 8 [✓]

Sviluppare una function C che, dati come parametri di input un array di char e il suo size, determina e restituisce come parametro di output l'array (di size 21) del numero delle occorrenze dell'evento **a** precede ognuna delle 21 lettere dell'alfabeto italiano (cioè il numero di volte in cui accade che **a** precede **a** , il numero di volte in cui accade che **a** precede **b**, il numero di volte in cui accade che **a** precede **c**, ...).

Esercizio 9 [✓]

Sviluppare una function C che, dati come parametri di input un array di **char** e il suo size, determina e restituisce come parametro di un dato logico che indica se il testo nell'array è un pangramma, ovvero è un testo che contiene, almeno una volta, tutte le 21 lettere dell'alfabeto italiano.

Esercizio 10 [✓]

Sviluppare una function C che, dati come parametri di input un array di **char** e il suo size, determina e restituisce come parametro di output il carattere più frequente.

Esercizio 11 [✓]

Sviluppare una function C che, dati come parametri di input un array di **char** e il suo size, determina e restituisce come parametro di output il carattere meno frequente.

Esercizio 12 [✓]

Sviluppare una function C che, dati come parametri di input un array di **char** e il suo size, determina e restituisce come parametri di output il carattere più frequente e il carattere meno frequente.

5.7 Ordinamento

In questo paragrafo presentiamo l'implementazione di alcuni algoritmi di complessità quadratica per l'ordinamento.

5.7.1 Inserimento

Come è noto, uno degli algoritmi più semplici per l'ordinamento degli elementi di un array è quello per **inserimento**. Per capire l'algoritmo possiamo immaginare di ordinare in maniera crescente le carte da gioco. A ogni passo, dobbiamo inserire una nuova carta facendo eventualmente slittare le carte fino a trovare il posto giusto per la nuova carta. La complessità di questo algoritmo è al più $O(n^2)$ (confronti) dove n è la dimensione dell'array di input. La funzione **ordin_inser** implementa l'algoritmo di ordinamento per inserimento

```
void ord_inser(char array[],int n)
 {
  int i,j;
  char el_da_ins;
  for (i=1;i<n;i++)
  {
      el_da_ins = array[i];
      j = i-1;
      while(j>=0  &&  el_da_ins < array[j])
      {
          array[j+1] = array[j];
```

```
            j--;
        }
        array[j+1] = el_da_ins;
    }
  }
```

5.7.2 Selezione

L'ordinamento per **selezione** effettua l'ordinamento determinando a ogni passo il minimo (o il massimo) di un sottoinsieme dell'array. La complessità in questo algoritmo è $O(n^2)$ confronti. La funzione **ord_sel_min** implementa l'algoritmo di ordinamento per selezione del **minimo**

```
 void ord_sel_min(char array[],int n)
 {
  int i,indice_min;
  char min_array;
  for (i=0;i<n-1;i++)
     {
    min_val_ind(&array[i],n-i,&min_array,&indice_min);
    scambiare_c(&array[i],&array[indice_min+i]);
     }
 }
void min_val_ind(char a[],int n,char *min_array, int *i_min) {
  int i;
  *min_array = a[0];
  *i_min = 0;
   for (i=1;i<n;i++)
       if (a[i] < *min_array)
    {
        *min_array = a[i];
        *i_min = i;
    }
}
```

L'algoritmo di ordinamento per selezione del **massimo** è implementato da

```
void ord_sel_max(char array[],int n)
 {
```

```
  int i,indice_max;
  char max_array;
  for (i=n-1;i>0;i--)
     {
    max_val_ind(&array[0],i+1,&max_array,&indice_max);
    scambiare_c(&array[i],&array[indice_max]);
     }
 }
void max_val_ind(char a[],int n,char *max_array, int *i_max) {
  int i;
  *max_array = a[0];
  *i_max = 0;
   for (i=1;i<n;i++)
       if (a[i] > *max_array)
    {
        *max_array = a[i];
        *i_max = i;
    }
}
```

Consideriamo il problema dell'ordinamento di un array di stringhe. La funzione **ord_sel_max_S** implementa l'algoritmo per ordinamento di selezione di massimo e agisce su array di stringhe

```
void ord_sel_max_S(char array_Stringhe[][50],int n)
 {
  int i;
  for (i=n-1;i>0;i--)
    scambiare_S(array_Stringhe[i],
              array_Stringhe[max_ind_S(array_Stringhe,i+1)]);
 }


 int max_ind_S(char array_Stringhe[][50],int n)
 {
  int i,i_max;
  i_max = 0;
  for (i=1;i<n;i++)
    if (strcmp(array_Stringhe[i],array_Stringhe[i_max])>0)
        i_max = i;
```

```
 return i_max;
}
```

Il **main** è il seguente

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

void ord_sel_max_S(char [][50],int);
int max_ind_S(char [][50],int); void scambiare_S(char *,char *);

void main() {
  int n, i;
  char elenco[][50]={"Pippo","Gastone","Qui","Pluto", "zio Paperone","Paperino","Paperina"};
  n = 7;
  printf("elenco non ordinato \n");
  for (i=0;i<n;i++)
    printf(" %s\n",elenco[i]);
  ord_sel_max_S(elenco,n);
  printf(\nelenco ordinato \n");
  for (i=0;i<n;i++)
    printf(" %s\n",elenco[i]);
}
 void scambiare_S(char *s1,char *s2)
{
  char temp[100];
  strcpy(temp,s1);  strcpy(s1,s2);  strcpy(s2,temp);
}
```

Se si considera invece un array di puntatori a **char**, otteniamo

```
void ord_sel_max_S(char *array_Stringhe[],int n)
 {
  int i;
  for (i=n-1;i>0;i--)
    scambiare_pS(&array_Stringhe[i],
       &array_Stringhe[max_ind_S(&array_Stringhe[0],i+1)]);
 }
 int max_ind_S(char *array_Stringhe[],int n)
 {
```

```
 int i,i_max;
 i_max = 0;
 for (i=1;i<n;i++)
     if (strcmp(array_Stringhe[i],array_Stringhe[i_max])>0)
      i_max = i;
 return i_max;
}
void scambiare_pS(char **s1,char **s2)
{
 char *temp;
 temp = *s1;  *s1 = *s2;  *s2 = temp;
}
```

In questo caso il **main** è

```
#include <stdio.h>
#include <string.h>
void ord_sel_max_S(char *[],int); int max_ind_S(char *[],int);
void scambiare_pS(char**,char **);


void main() {
  int n, i;
  char *elenco[]={"Pippo","Gastone","Qui","Pluto", "zio Paperone","Paperino","Paperina"};
  n = 7;
  printf("elenco non ordinato \n");
  for (i=0;i<n;i++)
    printf(" %s\n",elenco[i]);
  ord_sel_max_S(elenco,n);
  printf(\nelenco ordinato \n");
  for (i=0;i<n;i++)
    printf(" %s\n",elenco[i]);
}
```

5.7.3 Esercizi

Esercizio 1 [✓]

Scrivere una procedura che dato in input un array di strutture, ordini l'array in base al prezzo usando un algoritmo di ordinamento per inserimento. La struttura è

```
struct prodotto {char *nome; int codice; double prezzo;};
```

Esercizio 2 [✓]

Scrivere una procedura che dato in input un array di strutture, ordini l'array in base al prezzo usando un algoritmo di ordinamento per selezione di minimo. La struttura è

```
struct prodotto {char *nome; int codice; double prezzo;};
```

Esercizio 3 [✓]

Scrivere un programma per la gestione della segreteria studenti. Ogni studente è identificato dal nome, dal cognome e dalla matricola. Il corso ha complessivamente 10 studenti. Ordinare gli studenti in ordine alfabetico usando un algoritmo di ordinamento per inserimento. La struttura identificativa di uno studente è

```
struct sudente {char *nome; char *cognome; unsigned matricola;};
```

Esercizio 4 [✓]

Scrivere un programma per la gestione della segreteria studenti. Ogni studente è indetificato dal nome, dal cognome e dalla matricola. Il corso ha complessivamente 10 studenti. Ordinare gli studenti in ordine alfabetico usando un algoritmo di ordinamento per selezione di massimo. La struttura identificativa di uno studente è

```
struct sudente {char *nome; char *cognome; short matricola;};
```

Esercizio 5 [✓]

Dato un elenco di persone partecipanti ad un concorso, ordinare l'elenco in ordine alfabetico (ordinamento per selezione di minimo). La struttura che identifica il partecipante è

```
struct persona {char *nome; char *cognome;};

typedef struct persona id;

struct partecipante {id *utente; unsigned short codice; };
```

5.8 Ricorsione

In C, le funzioni possono essere usate in modo ricorsivo; cioè una funzione può richiamare se stessa direttamente o indirettamente. Come è noto, la ricorsione permette di ottenere la soluzione di un problema in maniera elegante senza l'utilizzo di cicli. Un programma ricorsivo richiede una quantità di memoria aggiuntiva perchè è necessario memorizzare in uno *stack* valori da processare relativi alle autoattivazioni finchè non si arriva all'istanza banale e quindi alla ricostruzione della soluzione finale.
Come primo esempio di funzione C ricorsiva consideriamo il calcolo del fattoriale. Dato un numero intero n il **fattoriale** è calcolato come $n = n \cdot (n-1) \cdot (n-2) \ldots \cdot 1$. La funzione C è

```
int fattoriale(int n)
{
  if (n <= 1)
       return 1;      /* soluzione del caso base */
   else
       return n * fattoriale(n-1);  /* autoattivazione */
}
```

Consideriamo ora la tecnica di programmazione ricorsiva per l'algoritmo incrementale di **somma** dei primi n numeri naturali

```
int somma_ric(int n) {
  if (n <= 1)
     return 1;
   else
     return n+somma_ric(n-1);
}
```

La funzione ricorsiva **somma_a_ricAI** implementa l'algoritmo ricorsivo di **somma** degli elementi di un array (**approccio incrementale** [2])

```
int somma_a_ricAI(int a[],int n) {
 if (n == 1)
     return a[0];
    else
```

[2] L'approccio incrementale per via ricorsiva viene anche detto **tail recursion** (ricorsione nella coda)

```
      return a[n-1] + somma_a_ricAI(a,n-1);
}
```

Lo stesso risultato può essere ottenuto considerando l'approccio divide-et-impera insieme alla tecnica ricorsiva. In questo caso, la funzione ricorsiva è

```
int somma_a_ricDI(int a[],int primo,int ultimo) {
 int mediano;
/* soluzione del caso base */
  if (primo == ultimo)
     return a[primo];
   else
    /* autoattivazioni */
    {
      mediano = (primo+ultimo)/2;
     return somma_a_ricDI(a,primo,mediano) +  somma_a_ricDI(a,mediano+1,ultimo);
      }
}
```

Il problema della determinazione del massimo **massimo** di un array è risolto dalla seguente funzione ricorsiva che implementa un algoritmo basato sull'approccio incrementale insieme alla tecnica ricorsiva

```
int massimo_a_ricAI(int a[],int n) {
 if (n == 1)
   return a[0];
  else
   return max_I(a[n-1],massimo_a_ricAI(a,n-1));
}
```

Lo stesso problema può essere risolto da una funzione ricorsiva basata sull'approccio divide-et-impera

```
int massimo_a_ricDI(int a[],int n) {
 int mediano;
 if(n == 1)
    return a[0];
  else
    {
    mediano = (n-1)/2;
```

```
    return max_I(massimo_a_ricDI(a,mediano+1),massimo_a_ricDI(a+mediano+1,n-mediano-1));
    }
}
```

Consideriamo l'implementazione dell'algoritmo ricorsivo per il calcolo della successione di **fibonacci**

```
int fib_ric_el(int n) {
 int n;
 if (n == 1 !! n == 0)
    return n;
  else
     return fib_ric_el(n-1) + fib_ric_el(n-2);
}
```

La complessità di questo algoritmo è del tipo esponenziale ($T(n) = O(1.6^n)$). Per ottenere un algoritmo con minore complessità $T(n) = n - 1$ ricorriamo alla tecnica di **programmazione dinamica** che è basata sull'uso di una matrice globale che contiene passo dopo passo le soluzioni già calcolate dalle chiamate di fibonacci effettuate fino a quel passo. Come è noto, l'algoritmo di ricerca binaria può essere espresso in modo molto elegante e sintetico mediante la tecnica ricorsiva. La funzione **ric_bin_ricTF** implementa l'algoritmo di **ricerca binaria** usando una tecnica ricorsiva. La complessità di tempo è la stessa del caso iterativo

```
Logical ric_bin_ricTF(char chiave,char elenco[],int n) {
 int mediano;
 if(n == 0)
    return False;
 mediano = (n-1)/2;
 if(chiave == elenco[mediano])
   return True;
  else if(chiave < elenco[mediano])
      return ric_bin_ricTF(chiave,elenco,mediano);
    else
      return ric_bin_ricTF(chiave,elenco+mediano+1,
                                        n-mediano-1);
}
```

e il programma principale potrebbe essere

```
#include <stdio.h>
#include <string.h>
typedef enum {False,True} Logical;

Logical ric_bin_ricTF(char,char [],int);

void main(){
 char elenco[100],cdr;
 Logical k;
 printf("inserire l'elenco di caratteri\n");
 gets(elenco);
 printf("inserire il carattere da ricercare: ");
 scanf("%c",&cdr);
 k = ric_bin_ricTF(cdr,elenco,strlen(elenco));
 if(k == True)
   printf("chiave trovata\n");
 else
   printf("chiave non trovata\n");
}
```

5.8.1 Esercizi

Esercizio 1 [✓]

Scrivere una funzione che calcoli con un approccio **ricorsivo** la somma dei prezzi di 10 prodotti identificati dalla seguente struttura

```
struct prodotto {int *id; char *nome; int prezzo;};
```

Esercizio 2 [✓]

Scrivere una funzione che calcoli con un approccio **divide-et-impera ricorsivo** la somma dei prezzi di 10 prodotti identificati dalla seguente struttura

```
struct prodotto {int *id; char *nome; int prezzo;};
```

Esercizio 3 [✓]

Scrivere una funzione che calcoli con un approccio **ricorsivo** il massimo dei prezzi di 10 prodotti identificati dalla seguente struttura

```
struct p_prodotto {char *costo; char *quantità;};

typedef struct p_prodotto id_prezzo;

struct partecipante {id_prezzo *prezzo; unsigned short codice; };
```

Esercizio 4 [✓]

Scrivere una funzione che calcoli con un approccio **divide-et-impera ricorsivo** il massimo dei prezzi di 10 prodotti identificati dalla seguente struttura

```
struct p_prodotto {char *costo; char *quantità;};

typedef struct p_prodotto id_prezzo;

struct partecipante {id_prezzo *prezzo; unsigned short codice; };
```

Esercizio 5 [✓]

Scrivere un programma che dato un array di puntatori alla seguente struttura

```
struct studente {char *nome; char *cognome; int matricola;};
```

ricerchi un dato alunno, usando l'algoritmo di **ricerca binaria ricorsiva**.

Esercizio 6 [✓]

Descrivere, brevemente, le caratteristiche durante la fase di esecuzione del seguente codice

```
int f(int x)
 {
 if (x<=1) return 1;
 else return (f(x)*f(x-1));
}
```

Esercizio 7 [✓]

Scrivere una funzione che usi un approccio ricorsivo per il calcolo della seguente funzione matematica

```
f(x) = 1                      se x <= 1

x*f(x-1) + f( x - 2 )^2 + 2   altrimenti
```

dove x assume valori interi da 1 a 10. Effettuare almeno due test.

Esercizio 8 [✓]

Scrivere una funzione che determini il risultato della seguente formula ricorrente quando $k = 20$

$$y_k = (y_{k-1} - y_(k-2))^2 \tag{5.4}$$

con $y_0 = y_1 = y_2 = 1$;

Capitolo 6

I File

Negli esempi visti nei Capitoli precedenti i dati di input sono prelevati dal dispositivo standard di input (tastiera) e i dati di output sono visualizzati sul dispositivo standard di output (schermo), che sono entrambi definiti dal sistema operativo locale. Molte operazioni possono essere fatte in memoria, altre invece necessitano di un accesso al disco fisso dove vengono conservati i dati e i programmi.

6.1 Accesso ai file

Prima di essere letto o scritto, un file deve essere aperto con la funzione di libreria **fopen**. La dichiarazione per poter usare un file è la seguente

```
File *fp;
File *fopen(char *name, char *mode);
```

Questa dichiarazione afferma che **fp** è un puntatore a **FILE** e che **fopen** restituisce un puntatore a **FILE**. **FILE** è una struttura che è definita nella libreria **stdio.h**.
Il primo argomento di **fopen** è una stringa contenente il nome del file. Il secondo argomento è il modo, una stringa, che indica come utilizzare il file

- **r** apre in lettura
- **w** crea in scrittura
- **a** appende
- **b** file binari
- **r+** apre il file di testo in aggiornamento (lettura e scrittura)

- **w+** crea il file in aggiornamento, scarta il vecchio contenuto
- **a+** appende

Per l'input e l'output formattati sui file si possono usare le funzioni **fscanf** e **fprintf**. Esse sono identiche a **scanf** e **printf**, ad eccezione del fatto che il loro primo argomento è un file pointer che specifica il file da cui leggere o scrivere

```
int fscanf(FILE *fp, char *format, ...)

int fprintf(FILE *fp, char *format, ...)
```

Per leggere e scrivere caratteri da un file possono essere usate anche le funzioni **getc** e **putc**.

```
int getc(FILE *fp);

int putc(int c, FILE *fp);
```

La funzione **getc** ritorna il prossimo carattere dal file **fp** e ritorna **EOF** per errore o fine file.
Infine, dopo aver aperto un file devo essere chiuso mediante l'utilizzo della funzione **fclose**

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

Quando un programma C entra in esecuzione, l'ambiente del sistema operativo si incarica di aprire tre file e di fornire i ripsettivi puntatori. Questi file sono lo standard input, lo standard output e lo standard error e i puntatori corrispondenti sono **stdin**, **stdout** e **stderr**.
Il seguente codice mostra un programma che permette di concatenare due file

```
#include <stdio.h>

/* cat: concatena file, prima versione */

void filecopy(FILE *, FILE *);

int main(int argc, char *argv[])
 {
  FILE *fp;
```

```
  if (argc == 1)

      filecopy(stdin, stdout);

  else
     while(--argc >0)
         if (( fp = fopen(*++argv,"r")) == NULL)
           {
            printf("cat: non posso aprire %s\n",*argv);
            return 1;
           }

         else
            {
            filecopy(fp, stdout);
            fclose(fp);
           }

 return 0;
}



/* filecopy: copia il file ifp sul file ofp */
main(int argc, char *argv[])
 {
  FILE *fp;

  if (argc == 1)

      filecopy(stdin, stdout);

  else
     while(--argc >0)
         if (( fp = fopen(*++argv,"r")) == NULL)
           {
            printf("cat: non posso aprire %s\n",*argv);
            return 1;
           }
```

```
          else
             {
             filecopy(fp, stdout);
             fclose(fp);
            }

 return 0;
}

/*filecopy: copia il file ifp sul file ofp */


void filecopy(FILE *ifp, FILE *ofp)
{
  int c;

     while(( c = getc(ifp)) != EOF)
     putc(c, ofp);
}
```

6.1.1 Esercizi

Esercizio 1 [✓]

Scrivere un programma che dato un file di testo memorizzato su disco permette di ricercare e visualizzare, con l'algoritmo del best matching, la migliore sottosequenza.

Esercizio 2 [✓]

Scrivere un programma che dato un file di testo memorizzato su disco permette di caricarlo, modificarlo e memorizzarlo. La modifica deve avvenire scambiando tutte le vocali con un simbolo scelto dall'utente.

Esercizio 3 [✓]

Scrivere un programma che memorizzi sul disco l'elenco completo degli studenti di un corso. Gli studenti sono identificati dalla seguente struttura

```
struct sudente {char *nome; char *cognome; unsigned matricola;};
```

Il programma deve permettere di memorizzare l'elenco, caricarlo e visualizzare gli studenti in ordine alfabetico.