RETIS LAB
Real-Time Systems Laboratory

Richiami sul Linguaggio C

Giorgio Buttazzo

Real-Time Systems Laboratory

scuola superiore Sant'Anna
di studi universitari e di perfezionamento

Sommario

Il programma più semplice

File: hello.c

```
#include  <stdio.h>

int main()
{
      printf("Hello world!\n");
      return 0;
}
```

- #include - direttiva per il precompilatore che sostituisce la linea corrente con il contenuto del file *header* "stdio.h" contenente le definizioni di alcune funzioni di libreria, tra cui la funzione printf().

Il programma più semplice

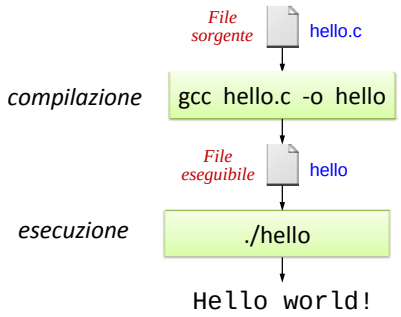
File: hello.c

```
#include  <stdio.h>

int main()
{
      printf("Hello world!\n");
      return 0;
}
```

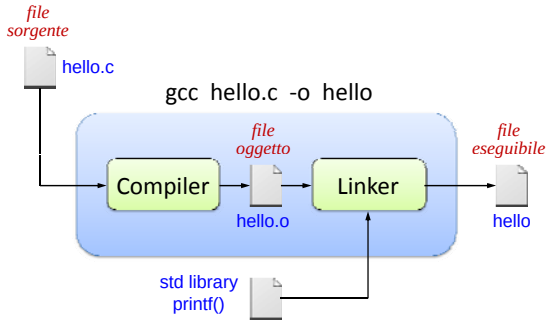
- int main() - definisce la funzione main(), funzione principale sempre presente in un programma C, che termina restituendo al sistema operativo un valore intero (il valore 0 indica una terminazione corretta).

Compilazione

In Linux:



Compilazione



Compilazione

Solo compilazione

> gcc -c mylib.c

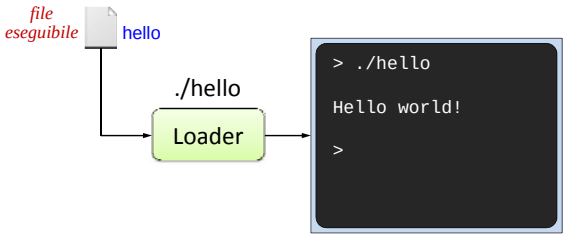
compila il file .c senza effettuare link

Compilazione e linking

> gcc prova.c mylib.o -o prova

- compila prova.c
- effettua il link con mylib.o
- produce il file eseguibile prova

Esecuzione



Loader: carica il programma dall'hard disk in RAM e comincia l'esecuzione chiamando la funzione main().

Struttura di un programma

```
inclusione file header
dichiarazioni globali
void main()
{
dichiarazioni locali
    corpo del programma
}
```

Esempio

```
#include  <stdio.h>

#define   PI   3.14159

int   main()
{
int   raggio = 10;
float cir;

      cir = 2*PI*raggio;
      printf("Raggio = %d\n", raggio);
      printf("Circonferenza = %f\n", cir);
      return 0;
}
```

Tipi di dati

Tipo	nome	# bit	range
char	carattere	8	[0, 255]
int	intero	16	[-32768, 32767]
float	virgola mobile	32	±3.4E-38, ±3.4E+38
double	doppia precisione	64	±1.7E-308, ±1.7E+308
void	indefinito	0	indefinito

NOTA: il numero di bit delle variabili dipende dall'architetture e dal compilatore. I dati in tabella si riferiscono a un'architettura a 16 bit.

Modificatori di tipo

unsigned senza segno
long intero a lunghezza doppia

Tipo	# bit	range
unsigned char	8	[0, 255]
unsigned int	16	[0, 65.535]
long int	32	[-2.147.483.648, +2.147.483.648]
unsigned long	32	[0, 4.294.967.295]

Operatore sizeof

sizeof(*tipo*)

Operatore che restituisce la dimensione in byte del tipo racchiuso tra parentesi.

Se viene applicato a una variabile non è necessaio utilizzare le parentesi tonde.

```
int a;

    printf("%d\n", sizeof(float));
    printf("%d\n", sizeof a);
```

Definizione di variabili

Il C è case sensitive, cioè differenzia minuscole da maiuscole.

Identificatori

- La lunghezza massima dipende dal compilatore. Lo standard garantisce 31 caratteri.
- Deve iniziare con una lettera o con underscore ('_') e può proseguire con lettere, underscore, o cifre.
- Non può essere una parola chiave del C.

Dichiarazione di variabili

Per convenzione, i nomi di variabili sono scritti in minuscolo:

```
char    c;
int     m, n;
float   x, y, z;
double  alpha;
```

Le variabili possono essere inizializzate:

```
char    c = 'a';
int     m, n = 8;
float   x = 5.2, y = 3.1, z;
double  alpha = 3.14;
```

ATTENZIONE

Nomi troppo lunghi peggiorano la leggibilità di un programma:

```
int     raggio_circonferenza_interna;
int     raggio_circonferenza_esterna;
```

DA EVITARE

```
int     raggio_int;
int     raggio_est;
```

molto meglio

I seguenti identificatori sono considerati diversi:

```
#define EPSILON  1.0E-5

double  epsilon;
double  Epsilon;
```

DA EVITARE

Dichiarazione di costanti

Costanti esplicite

32	costante intera (decimale)
0x20	costante intera (esadecimale)
3.1415	costante di tipo double (decimale)
'a'	costante di tipo carattere

Costanti simboliche

- Sono nomi simbolici che indicano valori prefissati.
- Hanno un tipo espresso implicitamente nel valore.
- Devono essere precedentemente dichiarate.
- Non occupano memoria (i simboli vengono sostituiti in fase di compilazione).

Dichiarazione di costanti

Le costanti simboliche possono essere dichiarate in 2 modi:

- attraverso la direttiva **`#define`**
- attraverso la parola chiave **`const`**

```
#define    ESC          27
#define    RET          '\015'
#define    SPACE        0x020
#define    CAR_A        'a'

const int  NMAX         10
const double PI         3.14159
const double EPSILON    1.0E-5
```

→ decimale
→ ottale
→ esadecimale
→ codice ASCII di 'a'

Commenti

I commenti sono racchiusi da due delimitatori:

`/*` inizio commento
`*/` fine commento

```
float    x, y;    /* coordinate punto    */

/*---------------------------------------*/
/*  Questo è un commento                 */
/*---------------------------------------*/

/*   questa dichiarazione viene ignorata
float    v, w;
*/
```

Commenti

È considerato commento anche il testo successivo al delimitatore `//`, fino alla fine del rigo:

```
float    x;        // coordinata x punto
float    y;        // coordinata y punto
float    alpha;    // angolo di rotazione
```

Operatori Aritmetici

+	somma
–	sottrazione
*	moltiplicazione
/	divisione
%	modulo (resto divisione intera)

```
c = 5 % 7;        /*  c = 5  */
c = 7 % 7;        /*  c = 0  */
c = 9 % 7;        /*  c = 2  */
```

Operatori Aritmetici

- Gli operatori (* / %) hanno precedenza su (+ –).
- La precedenza può essere modificata mediante l'uso delle parentesi tonde.

$c = a + \frac{b}{2}$	c = a + b/2;
$c = \frac{a + b}{2}$	c = (a + b) / 2;
$c = \frac{a\ b}{a + b}$	c = (a * b) / (a + b);

Abbreviazioni

i++	equivale a: i = i + 1
i––	equivale a: i = i – 1
x = n++	prima assegna, poi incrementa n
x = ++n	prima incrementa n, poi assegna
x = n––	prima assegna, poi decrementa n
x = ––n	prima decrementa n, poi assegna
x += a	equivale a: x = x + a
x –= a	equivale a: x = x – a

Operatori Logici

< <=	minore, minore o uguale
> >=	maggiore, maggiore o uguale
==	uguale
!=	diverso
&&	AND
||	OR
!	NOT

Ogni espressione diversa da zero ha valore logico **TRUE**, e ogni espressione uguale a zero ha valore logico **FALSE**.

Operatori Logici sul bit

&	AND
|	OR
^	XOR
<<	Shift Left
>>	Shift Right
~	Complemento

Operatori Logici sul bit

Esempi:

```
int   x = 15;
int   y = 3;
int   z;
```

Quanto vale z?

z = x & y	3
z = x | y	15
z = x ^ y	12
z = y << 3	24
z = x >> 1	7
z = x & ~07	8

Conversioni di tipo

Sono necessarie quando si eseguono operazioni fra variabili di tipo diverso:

```
int    a, b, c;
float  x, y;

       x = a/b + y;

       c = x*y;
```

In C sono possibili due modi di conversione:

- Conversioni implicite
- Conversioni esplicite (casting)

Conversioni implicite

Sono effettuate dal compilatore come segue:

Operazioni a due operandi

L'operando a minor precisione viene convertito nel tipo dell'operando a maggior precisione.

Assegnamento

L'operando sorgente viene convertito nel tipo dell'operando destinatario.

➡ Se il tipo destinatario è a minor precisione c'è il rischio di troncamento o overflow.

Conversioni implicite

```
char   h = '\a', k;
int    a = 128, b = 256, c;
float  x = 4.6;
```

```
k = b + h;
```
← h convertito in int e poi sommato a b. Risultato convertito a char.

```
c = a/b + x;
```
← a/b fra interi, poi convertito in float e sommato a x. Risultato convertito a int.

Conversioni esplicite

Per evitare simili problemi, è possibile forzare la conversione di tipo:

```
int    a = 5, b = 10, c;
float  x = 1.5;

  c = a/b + 3*x;
  printd("c = %d\n");                  → 4

  c = a/(float)b + 3*x;
  printd("c = %d\n");                  → 5

  c = a/(float)b + 3*(int)x;
  printd("c = %d\n");                  → 3
```

Libreria math.h

sin(x)	seno di x;
cos(x)	coseno di x;
tan(x)	tangente di x;
asin(x)	arcoseno di x, fra $-\pi/2$ e $\pi/2$, con $-1 \le x \le 1$;
acos(x)	arcocoseno di x, fra 0 e π, con $-1 \le x \le 1$;
atan(x)	arcotangente di x, fra $-\pi/2$ e $\pi/2$;
atan2(y,x)	arcotangente di y/x, fra $-\pi$ e π (il risultato dipende dal segno degli argomenti);

argomenti e valori di ritorno sono tutti di tipo double

Libreria math.h

log(x)	logaritmo naturale (in base *e*) di x;
log10(x)	logaritmo in base 10 di x;
abs(n)	valore assoluto di n (argomento e risultato sono interi);
fabs(x)	valore assoluto di x (argomento e risultato sono double);

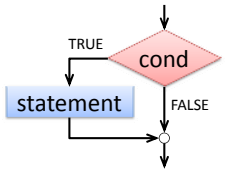
Libreria math.h

ceil(x)	restituisce il più piccolo intero maggiore o uguale a x;
floor(x)	restituisce il più grande intero minore o uguale a x;
sqrt(x)	radice quadrata di x;
exp(x)	esponenziale: *e* elevato a x;
pow(x,y)	x elevato a y.

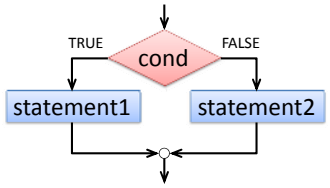
Controllo del flusso del programma

IF

```
if (cond) {
      statement;
}
```

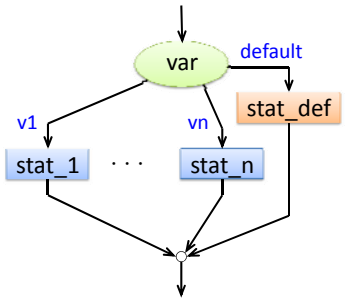



```
if (cond) {
      statement1;
}
else {
      statement2;
}
```



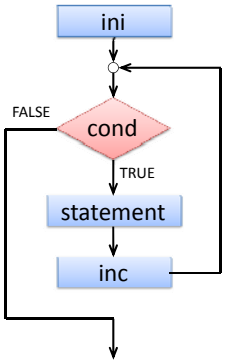
SWITCH

```
switch (var) {

   case v1:
        stat_1;
        break;
     :
   case vn:
        stat_n;
        break;

   default:
        stat_def;
        break;
}
```



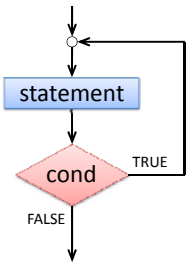
FOR

```
for (ini; cond; inc) {

      statement;

}
```



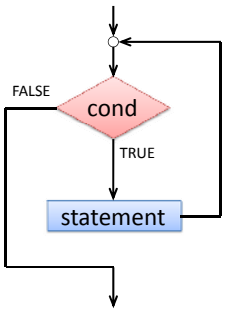
DO

```
do {

      statement;

} while (cond);
```



WHILE

```
while (cond) {

      statement;

}
```



Abbreviazioni

In un blocco contenente una sola istruzione, le parentesi graffe possono essere omesse:

```
if (a > 0) x = 5;
else x = 3;

while (i > 0) i--;

do x = a + x;
while (x < a);

for (i=1; i<=10; i++)
      printf("i = %d\n", i);
```

Istruzione break

E' sempre possibile uscire da un ciclo mediante l'istruzione **break**:

```
for (i=1; i<=10; i++) {

      printf("Passo numero %d\n", i);
      if (i == 7) break;
}
```

⇒ Esegue solo le prime 7 iterazioni.

Istruzione continue

Quando all'interno di un ciclo viene eseguita l'istruzione **continue**, il flusso salta immediatamente all'inizio del ciclo, senza eseguire le istruzioni successive a **continue**:

```
for (i=1; i<=10; i++) {

      if (i%2 == 0) continue;
      printf("Passo numero %d\n", i);
}
```

⇒ Stampa solo nelle iterazioni dispari.

Istruzioni di ingresso/uscita

Stampa su video

printf ("*stringa di formattazione*", var1, var2, ...);

La "*stringa di formattazione*" può contenere caratteri ordinari, come "Hello world", oppure caratteri di controllo, quali:

\n	a capo, newline <LF> (line feed)
\r	ritorno a capo <CR> (carriage return)
\t	tabulazione orizzontale <TAB>
\b	spazio indietro <BS> (backspace)
\a	segnale di alert <BELL> (beep)
\'	apice
\"	doppio apice
\\	backslash (\)

Stampa su video

printf ("*stringa di formattazione*", var1, var2, ...);

I seguenti caratteri di controllo consentono di stampare il valore di una variabile:

%d	intero (decimale)
%u	unsigned
%o	intero (ottale)
%x	intero (esadecimale)
%c	carattere
%f	float
%s	stringa
%%	stampa il carattere '%'

Formattazione numeri

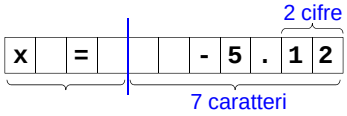
Quando si stampa un float o double, è possibile specificare il numero di cifre dopo il punto:

```
double x = -5.1234;
printf("x = %7.2f\n", x);
```

width: numero minimo di caratteri per la stampa (incluso segno e punto)

precision: numero di cifre dopo il punto (6 per default)

stampa prodotta:



Esempio

Si scriva un programma che stampi la seguente tabella di corrispondenze tra gradi Fahrenheit e gradi Celsius:

```
> ./gradi
  F       C
  0    -17.8
 10    -12.2
 20     -6.7
 30     -1.1
 40      4.4
 50     10.0
 60     15.6
 70     21.1
 80     26.7
 90     32.2
100     37.8
```

Esempio

File: gradi.c

```
#include <stdio.h>

int    main()
{
int    gf;     /* gradi fahrenheit */
float  gc;     /* gradi centigradi */

       printf("F\tC\n");
       for (gf=0; gf<=100; gf=gf+10) {
           gc = (5.0/9.0)*(gf - 32);
           printf("%d\t%5.1f\n", gf, gc);
       }
       return 0;
}
```

Ingresso da tastiera

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

Blocca l'esecuzione del programma finché non viene premuto il tasto INVIO (newline). Quindi, legge un intero dal buffer di tastiera e lo assegna alla variabile n, di indirizzo &n.

- I caratteri di controllo sono gli stessi di printf.
- L'uso di scanf richiede l'inclusione del file header stdio.h

Esempio

File: cerchio.c

```
#include <stdio.h>
#define PI 3.14159

int    main()
{
int    raggio;
float  circo;
       printf("Raggio = ");
       scanf("%d", &raggio);
       circo = 2*PI*raggio;
       printf("Circonferenza = %f\n", circo);
       return 0;
}
```

Esempio

File: converti.c

```
#include <stdio.h>

int   main()
{
int   n;
      while (1) {
          printf("n = ");
          scanf("%d", &n);
          if (n == 0) break;
          printf("OCT = %o, EXA = %x\n", n, n);
      }
      return 0;
}
```

putchar

```
int   putchar(char c);
```

Scrive un carattere sul video e restituisce il carattere scritto (come numero intero).

```
char c = 'a';
int  i;

     for (i=0; i<10; i++) {
         putchar(c+i);
     }
}
```

getchar

```
int   getchar();
```

Blocca l'esecuzione del programma fino alla pressione del tasto INVIO (newline) e restituisce il carattere letto (come numero intero).

```
char c;

     do {
         c = getchar();
         putchar(c);
     } while ((c < '0') || (c > '9'));
```

Puntatori ed Array

Puntatore

Un puntatore è una variabile che contiene l'indirizzo di un'altra variabile:

```
int   x, y;
int   *p;
```
← p è un puntatore ad un intero

```
      p = &x;
```
← p prende l'indirizzo di x

```
      y = *p;
```
← y prende il contenuto della variabile puntata da p

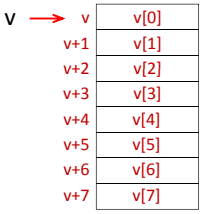
Array

E' un insieme di variabili tutte dello stesso tipo, allocate in memoria in locazioni consecutive:

```
int   v[8];
```

Alloca 8 variabili intere consecutive, per un totale di 16 byte:

- Gli elementi sono accessibili come v[i], dove i è un indice da 0 a 7.
- Il nome dell'array è anche l'indirizzo del primo elemento: dunque v equivale a &v[0].

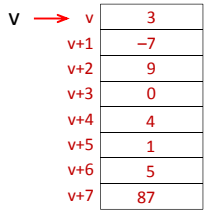


Array

- Un array può essere inizializzato in fase dichiarativa:

```
int   v[8] = {3, -7, 9, 0, 4, 1, 5, 87};
```

```
v[2] = v[3];
```
copia l'elemento v[3] in v[2]



- L'elemento i-esimo ha indirizzo &v[i] corrispondente a (v + i):

  l'indirizzo è calcolato come v + i*dimensione_intero

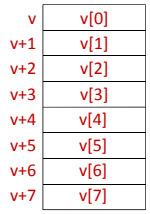
- L'espressione v[i] è quindi una forma semplifica per *(v + i).

Array e puntatori

Si noti dunque l'equivalenza delle due notazioni :

```
int   v[8];
```

array	puntatori
&v[0]	v
v[0]	*v
&v[i]	(v+i)
v[i]	*(v+i)



Dimensione di un array

La dimensione di un array dev'essere nota al momento di compilazione, dunque dev'essere una costante numerica o simbolica:

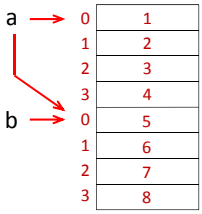
```
#define DIM    4

int   n = 10;

int   a[5];      /* corretto */

int   b[DIM];    /* corretto */

int   c[n];      /* errato! */
```

Assegnazione di array

Non esiste l'assegnazione di array:

```
#define   DIM  4

int   a[DIM] = {1, 2, 3, 4};
int   b[DIM] = {8, 8, 8, 8};

      a = b;
```

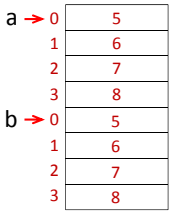
Assegna ad a l'indirizzo di b. Dunque a diventa inaccessibile!



Assegnazione di array

L'assegnazione corretta richiede un ciclo **for**:

```
#define   DIM  4

int   a[DIM] = {1, 2, 3, 4};
int   b[DIM] = {8, 8, 8, 8};
int   i;

for (i=0; i<DIM; i++)
      a[i] = b[i];
```



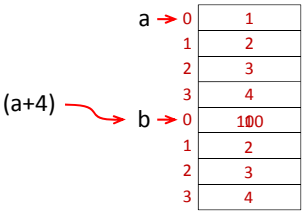
Limiti di un array

I compilatori non controllano se un indice eccede la fine dell'array, né durante la compilazione, né durante l'esecuzione.

Il controllo spetta al programmatore:

```
a[4] = 100;
```

Sovrascrive un'altra locazione di memoria: l'elemento b[0].

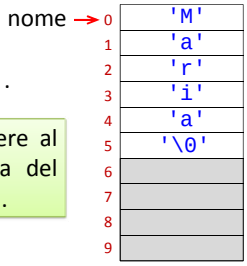


Stringhe

Una stringa è un array di caratteri che termina con il numero zero ('\0'):

```
char   nome[10] = "Maria";
```

Definisce un array di 10 byte e inizializza i primi 6 byte con 'M', 'a', 'r', 'i', 'a', '\0'.



- La stringa nome può contenere al massimo 9 caratteri, a causa del carattere di terminazione ('\0').

Lunghezza di una stringa

Questo pezzo di codice calcola la lunghezza di una stringa contando i caratteri diversi da '\0':

```
char  nome[20];     /* stringa del nome     */
int   ls = 0;       /* contatore caratteri */

      printf("Inserisci un nome: ");
      scanf("%s", nome);

      while (nome[ls] != '\0') ls++;

      printf("Il nome è di %d caratteri\n", ls);
```

Libreria string.h

strlen(s)

restituisce la lunghezza della stringa s, senza considerare il carattere terminatore.

strcmp(s1, s2)

restituisce un numero intero di valore:

- = 0 se s1 == s2
- < 0 se s1 < s2
- > 0 se s1 > s2

strcpy(s1, s2)

copia s2 in s1 (incluso il terminatore) e restituisce il puntatore a s1.

strcat(s1, s2)

aggiunge s2 alla fine di s1 (prima del terminatore) e restituisce il puntatore a s1.

Libreria stdlib.h

atoi(s)	(ascii to int): converte la stringa s in un valore intero, restituito in uscita. Restituisce 0 se la stringa non può essere convertita.
atol(s)	(ascii to long): converte la stringa s in un valore long, restituito in uscita. Restituisce 0 se la stringa non può essere convertita.
atof(s)	(ascii to float): converte la stringa s in un valore float, restituito in uscita. Restituisce 0 se la stringa non può essere convertita.
itoa(n, s, b)	(int to ascii): converte un intero n espresso in base b in una stringa s.

Array multidimensionali

Sono realizzati come array di array:

```
int   matrice[3][4];
```

array di 3 elementi

ogni elemento è un array di 4 elementi

Esso rappresenta una matrice 3x4 (3 righe e 4 colonne):

```
int   matrice[3][4];
```

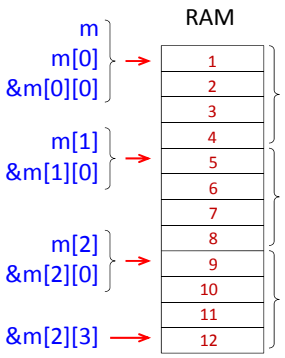
righe colonne

Array multidimensionali

```
int   m[3][4] = {
      {1,  2,  3,  4},
      {5,  6,  7,  8},
      {9, 10, 11, 12}};
```

Definisce una matrice 3x4 (3 righe e 4 colonne) così inizializzata:

$$m = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$



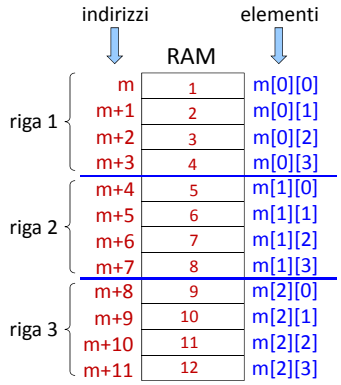
Array multidimensionali

$$m = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

In generale, se

```
int   m[NR][NC];
```

L'elemento m[i][j] ha indirizzo: (m + i∗NC + j)

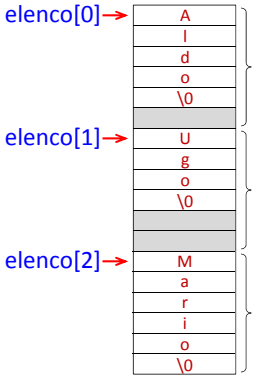


Array di stringhe

```
char  elenco[3][6] = {
      "Aldo",
      "Ugo",
      "Mario"};
```

Definisce un array di 3 stringhe lunghe al più 5 caratteri:

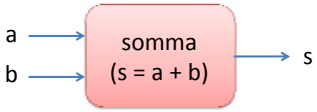
- elenco[1] è un puntatore che contiene l'indirizzo della stringa "Ugo"
- elenco[2][1] è una variabile che contiene il carattere 'a'



Funzioni

Elementi di una funzione

In C, una funzione è un modulo di programma che riceve in ingresso dei parametri (argomenti) e restituisce al più un solo valore in uscita:

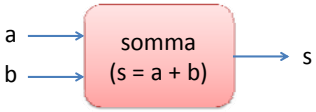


Per poter utilizzare una funzione è necessario:

- dichiarare il prototipo (prima del main)
- definire il codice sorgente (dopo il main)
- chiamare la funzione (dal codice di un'altra)

Prototipo di funzione

Il prototipo di una funzione è una dichiarazione che informa il compilatore sul numero e tipo di argomenti e sul tipo del valore di ritorno.



Se si lavora con numeri interi il prototipo risulta:

```
int    somma(int a, int b);
```

tipo valore di ritorno

tipo e numero di argomenti

Definizione di funzione

La definizione di una funzione definisce l'interfaccia e il codice sorgente:

```
int    somma(int a, int b)
{
int    s;

       s = a + b;
       return s;
}
```

argomenti

variabile locale

entrambi esistono solo all'interno della funzione

Chiamata di funzione

La funzione può essere chiamata come segue:

```
int    main()
{
int    x, y, z;

       printf("x = ");
       scanf("%d", &x);
       printf("y = ");
       scanf("%d", &y);

       z = somma(x,y);
       printf("somma = %d\n", z);
       return 0;
}
```

Programma completo

```
#include <stdio.h>

int      somma(int a, int b);

/*---------------------- MAIN -------------------------*/
int      main()
{
int      x = 5, y = 3, z;

         z = somma(x,y);
         printf("somma = %d\n", z);
         return 0;
}

/*--------------------- FUNZIONI ----------------------*/
int      somma(int a, int b)
{
         return (a + b);
}
```

Struttura del programma

Nell'ordine, un programma C ha la seguente struttura:

1. inclusione dei file header
2. definizione delle costanti globali
3. dichiarazione delle variabili globali
4. dichiarazione dei prototipi delle funzioni
5. definizione della funzione main()
6. definizione delle altre funzioni

Funzioni di tipo void

Una funzione che non restituisce alcun valore deve essere dichiarata di tipo **void**.

La seguente funzione riceve come argomento un intero e stampa il valore del cubo:

```
void    stampa_cubo(int a)
{
int     c;

        if (abs(a) >= 32) printf("Too big\n");
        else {
            c = a*a*a;
            printf("%d\n", c);
        }
}
```

Funzione booleane

In C non esistono variabili booleane (ossia variabili che assumono solo due valori: TRUE, FALSE).

Se si vuole definire una funzione che restituisca TRUE o FALSE, è sufficiente restituire un valore positivo (1) o nullo (0):

```
int     pari(int a)
{
        if (a%2 == 0) return 1;
        else return 0;
}
```

Esercizi

- sfrenata(v, a) → restituisce lo spazio frenata in funzione di vel. v e attrito a
- gauss(n) → $\sum_{i=1}^{n} i$
- fattoriale(n) → n!
- primo(n) → restituisce 1 se n è primo
- parity(n) → restituisce 1 se n contiene un numero pari di bit a 1

Funzioni ricorsive

Una funzione può chiamare se stessa:

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

fatt(3) → 3 * fatt(2) → 3 * (2 * fatt(1)) →

→ 3 * (2 * 1) → 3 * 2 → 6

Visibilità delle variabili

Ogni variabile ha una sua zona di validità (scope) in cui è visibile all'interno del programma.

Variabili globali	sono dichiarate all'inizio, all'esterno di ogni funzione, e sono visibili da tutte.
Variabili locali	sono dichiarate all'interno di una funzione e sono visibili solo da essa.

Una variabile locale non conserva il contenuto tra una chiamata e l'altra, a meno che non venga dichiarata **static**.

Una variabile locale con lo stesso nome di una globale maschera quella globale.

Variabili static

Sono di tipo permanente e possono essere di tipo locale e globale:

Variabili globali statiche

Sono dichiarate all'inizio, all'esterno di ogni funzione, e sono visibili solo dalle funzioni definite nello stesso file.

Variabili locali statiche

Sono dichiarate all'interno di una funzione e sono accessibili solo da essa. Tuttavia non vengono distrutte all'uscita della funzione, ma conservano il proprio valore tra chiamate successive. L'eventuale inizializzazione viene fatta una sola volta all'inizio del programma.

Visibilità delle variabili

```
int     max = 10;          <- globale

int     main()
{
int     i;                 <- locale

        for (i=0; i<max; i++) fun();
}

void    fun()
{
int     a = 0;             <- locale
static  int b = 0;         <- locale statica

        if (b == 0) printf("START\n");

        printf("%d,%d\n", a, b);
        a++;
        b++;
}
```

Allocazione statica

La dimensione di un array dev'essere nota al momento della compilazione:

```
int     n;
        n = 10;
int     v[n];     /* ERRATO */
int     w[10];    /* CORRETTO */
```

Se la dimensione effettiva di un array non è nota al momento della compilazione, occorre:

- allocarlo con dimensione massima tra quelle possibili;
- operare su di esso mediante una variabile che ne fissi la dimensione effettiva.

Allocazione statica

Esempio:

```
#define MAX_DIM  10;

int     v[MAX_DIM];
int     dim;

        /* calcola dim dinamicamente */
        if (dim > MAX_DIM) exit(1);

        /* opera sull'array effettivo */
        for (i=0; i<dim; i++)
              v[i] = i;
```

Allocazione dinamica

E' possibile allocare/deallocare memoria dinamicamente attraverso due funzioni (dichiarate in stdlib.h):

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

Cerca di allocare size byte di memoria. Se la memoria è disponibile, restituisce il puntatore al primo byte allocato, altrimenti restituisce NULL.

```
void    free(void *p);
```

Rilascia un blocco di memoria puntato da p, precedentemente allocato.

Uso di malloc e free

```
#include <stdlib.h>
int  main()
{
int  *p;
int  n;

     n = 10;
     /* alloca un blocco di n interi */
     p = (int *)malloc(n*sizeof(int));
     if (p == NULL) exit(1);

     /* libera il blocco di n interi */
     free(p);
```

casting esplicito del puntatore

dimensione calcolata dal programma

Gestione della memoria

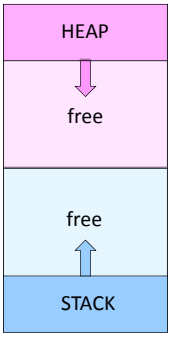
I diversi tipi di oggetti sono gestiti in modo diverso e risiedono in segmenti diversi della memoria RAM:

codice eseguibile	TEXT SEGMENT
variabili globali inizializzate	DATA SEGMENT
variabili globali non inizializzate	BSS
variabili allocate con malloc	HEAP SEGMENT
variabili locali	STACK SEGMENT

Heap e Stack

Se la memoria richiesta è superiore a quella disponibile nello heap, la malloc restituisce un puntatore **NULL**.

Se durante l'esecuzione lo stack cresce oltre la dimensione massima, viene generato un errore di **stack overflow**.



Funzioni e puntatori

Lo scopo principale dei puntatori è consentire a una funzione di manipolare una grossa struttura dati senza dover passare tutti i dati:

```
int   main()
{
int   v[8] = {1, 2, 3, 4, 5, 6, 7, 8};
int   m;

      m = max_array(v,8);
      printf("max = %d\n", m);
      return 0;
}
```

puntatore all'array

dimensione array necessaria poiché la funzione non ha modo di sapere la lunghezza.

Passaggio di array

La funzione max_array() dovrà quindi accettare come argomenti un puntatore a intero e un intero:

```
int   max_array(int v[], int dim)
{
int   i, max;

      max = v[0];   /* primo elemento */

      for (i=1; i<dim; i++)
           if (v[i] > max) max = v[i];

      return max;
}
```

Passaggio di array

Alternativamente si può anche scrivere:

```
int   max_array(int *v, int dim)
```

Poiché la funzione ha accesso ad un array esterno, essa è potenzialmente in grado di modificarlo.

Per evitare che ciò accada per errore, è possibile dichiarare l'oggetto puntato come costante:

```
int   max_array(const int v[], int dim)
```

Se si tenta una modifica, il compilatore genera un errore: "cannot modify a const object"

Passaggio di matrici

Analogamente, il passaggio di una matrice richiede il puntatore al suo primo elemento e le sue dimensioni:

```
int   main()
{
int   m[2][3] = {
      {1, 2, 3},
      {4, 5, 6}};

          stampa_mat(m,2,3);
          return 0;
}
```

puntatore all'array

dimensioni matrice necessarie poiché la funzione non ha modo di conoscerle.

Passaggio di matrici

La funzione stampa_mat() dovrà quindi essere definita come segue:

```
void  stampa_mat(int m[2][3], int r, int c)
{
int   i, j;

      for (i=0; i<r; i++) {
           for (j=0; j<c; j++)
                printf("%d ", m[i][j]);
           printf("\n");
      }
}
```

Passaggio di matrici

NOTA: Per una matrice di dimensioni NR x NC, l'indirizzo dell'elemento m[i][j] viene calcolato come (m + i*NC + j), pertanto la prima dimensione (NR) può essere omessa:

```
void  stampa_mat(int m[][3], int r, int c)
```

Se la matrice ha n dimensioni, il calcolo dell'indirizzo richiede la specifica di tutte le dimensioni tranne la prima.

Passaggio di matrici

Alternativamente, è possibile passare il solo puntatore, calcolando esplicitamente l'indirizzo:

```
void  stampa_mat(int *m, int r, int c)
{
int   i, j, a;

      for (i=0; i<r; i++) {
           for (j=0; j<c; j++) {
                a = *(m + i*c + j);
                printf("%d ", a);
           }
           printf("\n");
      }
}
```

Passaggio di matrici

In tal caso, la funzione deve essere chiamata come :

```
int   main()
{
int   m[2][3] = {{1, 2, 3}, {4, 5, 6}};

          stampa_mat((int *m), 2, 3);
          return 0;
}
```

- La conversione è necessaria in quanto m non è un puntatore a int ma ad array di 3 interi: int (*m)[3]; (NOTA: int *m[3]; dichiara un array di 3 puntatori a int).
- La conversione non è richiesta per array singoli, perché le dichiarazioni int v[ ] e int *v sono equivalenti.

Ancora puntatori

int *v;	v è un puntatore ad un intero
int *v[8];	v è un array di 8 puntatori a intero
int (*v)[8];	v è un puntatore ad un array di 8 interi
int *fun();	fun è una funzione che restituisce un puntatore ad intero
int (*fun)();	fun è un puntatore a funzione che restituisce un intero

Esercizi

- scalare(a, b) → $\sum_{i=1}^{n} a_i b_i$
- slung (s) → lunghezza della stringa s
- vocali(s) → numero vocali in s
- inverti(s1, s2) → inverte s1 e la scrive in s2
- ordina(v) → ordina l'array v di interi
- det(a, n) → determinante di a[N][N]

La funzione main()

La funzione main() è una funzione come tutte le altre, con le seguenti differenze:

- ha un nome prefissato;
- viene invocata dal sistema operativo quando viene eseguito il programma;
- può ricevere parametri dalla linea di comando;
- restituisce un intero, interpretato come codice di errore (0 indica terminazione corretta).

Parametri del main

La funzione main() può ricevere dei parametri attraverso la linea di comando:

```
> myprog par1 par2 par3
```

I parametri vengono memorizzati in un array di stringhe e sono accessibili attraverso 2 argomenti:

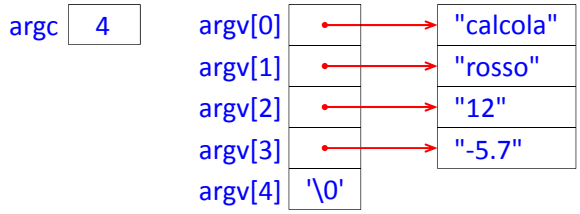
- argc — intero contenente il numero di stringhe digitate nella linea di comando;
- argv — array di stringhe contenente le stringhe digitate nella linea di comando.

Parametri del main

Ad esempio, se si digita

```
> calcola rosso 12 -5.7
```

si ha:



Definizione del main

```
int    main(int argc, char *argv[])
{
int    p1;
float  p2;

   printf("num. par. = %d\n", argc-1);
   printf("par. 1: %s\n", argv[1]);
   p1 = atoi(argv[2]);
   p2 = atof(argv[3]);
   printf("par. 2: %d\n", p1);
   printf("par. 3: %f\n", p2);
   return 0;
}
```

Strutture

Strutture: dichiarazione

Sono insiemi di più variabili anche di tipo diverso. Ciascuna è detta *campo* o *membro* della struttura:

```
struct  persona {
   char nome[20];
   char cognome[20];
   int  eta;
   long telefono;
};
```

```
struct  punto {
   int  x;
   int  y;
};
```

queste sono solo dichiarazioni di tipo: pertanto non viene allocata memoria!

Strutture: definizione

Definizione di variabili di tipo struttura:

```
struct  persona uomo;

struct  persona prof = {
   "Ugo", "Rossi", 45, 232553};

struct  persona studente[50];

struct  punto p1, p2;

struct  punto origine = {0,0};
```

Strutture: definizione

Le variabili di tipo struttura possono essere definite anche contestualmente alla dichiarazione:

```
struct   libro {
    char titolo[50];
    char autore[20];
    char editore[50];
    int  anno;
}lib1, lib2, elenco[100];
```

```
struct   punto {
    int  x;
    int  y;
}p1, p2;
```

Strutture: accesso

I campi di una struttura sono acceduti come segue:

```
prof.cognome        puntatore alla stringa "Rossi"
prof.eta            intero che vale 45
prof.nome[1]        carattere che vale 'g'

p1.x = 3;           assegna a p1 le coordinate (3,5)
p1.y = 5;

prof.nome[1] = 't';
printf("%s", prof.nome);      stampa  Uto
printf("%d", prof.eta+1);     stampa  46
```

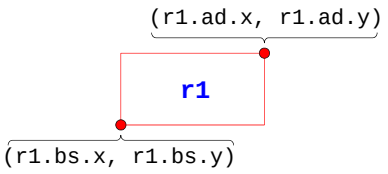
Strutture: assegnazione

Le strutture possono essere assegnate in blocco:

```
uomo = prof;

p1 = origine;


printf("%s", uomo.cognome);  stampa  Rossi
printf("%d", uomo.eta);      stampa  45
printf("%d,%d", p1.x, p1.y); stampa  0,0
```

Strutture di strutture

```
struct rettangolo {
    struct   punto bs;    /* basso sx */
    struct   punto ad;    /* alto  dx */
};

struct rettangolo r1;
```



Passaggio di strutture

La seguente funzione prende come argomenti due strutture di tipi punto e restituisce la loro distanza:

```
double  dist(struct punto a,
             struct punto b)
{
        return sqrt(
         (a.x - b.x)*(a.x - b.x) +
         (a.y - b.y)*(a.y - b.y));
}
```

NOTA: la funzione non è in grado di modificare i campi delle strutture, in quanto passate per valore!

Chiamata della funzione

La chiamata della funzione avviene come segue:

```
int  main()
{
struct point p1 = {3,5};
struct point p2 = {6,2};
double d;

     d = dist(p1,p2);
     printf("distanza = %f\n", d);
     return 0;
}
```

Passaggio per indirizzo

Se vogliamo che una funzione modifichi i campi di una struttura, questa va passata per indirizzo:

```
int  main()
{
struct point q;

     make_point(&q, 9, 7);
     printf("q = (%d,%d)\n", q.x, q.y);
     return 0;
}
```

Puntatori a strutture

```
void    make_point(
        struct punto *p, int a, int b)
{
        p->x = a;
        p->y = b;
}
```

NOTA
L'accesso ai campi di una struttura puntata da un puntatore avviene attraverso l'operatore **->** :

p->x equivale a **(*p).x**

File

File

Unità logica di memorizzazione (persistente) su memoria di massa (hard disk):

- sequenza di componenti (record logici) che termina con una *marca*: **EOF** (End Of File).

I file C vengono distinti in due categorie:

file di testo: visti come sequenze di caratteri;

file binari: visti come sequenze di bit.

File di testo

- Sono file di caratteri organizzati in linee.
- Ogni linea termina con il carattere **'\n'** (newline).

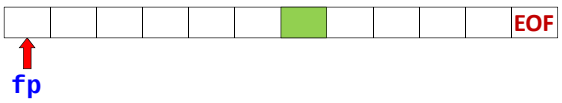
```
Questo è un file di testo:<LF>
<LF>
Aldo Rossi, 23<LF>
Bruno Bianchi, 34<LF>
Dino Sauro, 82<LF>
Ugo Verdi, 52<LF>
Ivo Neri, 71<LF>
<EOF>
```

File di testo

Un file ha una struttura sequenziale:

- per accedere ad un particolare record logico è necessario scorrere tutti quelli che lo precedono
- Lo scorrimento avviene per mezzo di un **puntatore al file**:

```
#include <stdio.h>

FILE  *fp;
```



Operazioni su file

Apertura: verifica l'esistenza del file e inizializza il puntatore al file.

Chiusura: memorizza il file su disco.

Accesso: permette la lettura o scrittura di dati.

Controllo: verifica se il puntatore ha raggiunto la fine del file <EOF>.

Apertura di un file

```
FILE   *fopen(char *name, char *mode);
```

name array di caratteri che rappresenta il nome del file.

mode modalità di accesso al file:

- "r" in lettura (read);
- "w" in scrittura (write);
- "a" scrittura con aggiunta in fondo (append);
- "b" a fianco ad una delle precedenti indica che il file è binario.

Apertura in lettura

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

int   main()
{
FILE  *fp;
      fp = fopen("dati.txt", "r");
      if (fp == NULL) {
          printf("Il file non esiste\n");
          exit(1);
      }
```

Se il file esiste, fopen restituisce il puntatore al primo record del file, altrimenti restituisce il valore **NULL**.

Apertura in scrittura

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

int   main()
{
FILE  *fp;
      fp = fopen("dati.txt", "w");
      if (fp == NULL) {
          printf("Spazio insufficiente\n");
          exit(2);
      }
```

Se il file non esiste, lo crea; se esiste, lo cancella e ne crea uno nuovo. Se non c'è spazio su disco, restituisce **NULL**.

Chiusura di un file

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

Salva il file su disco e restituisce:

0 se la chiusura è stata eseguita con successo;

EOF se la chiusura non è andata a buon fine.

```
      err = fclose(fp);
      if (err == EOF) {
          printf("Errore di chiusura file\n");
          exit(3);
      }
```

Controllo di fine file

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

La funzione **feof(fp)** controlla se e` stata raggiunta la fine del file **fp** nell'operazione di lettura o scrittura precedente. Restituisce:

- 0 (falso), se non e` stata raggiunta la fine del file;
- un valore diverso da zero (vero), altrimenti.

```
      if (feof(fp)) {
          printf("Fine del file\n");
          exit(4);
      }
```

Accesso al file

Lettura

```
int   fscanf(FILE *fp, char *format, …);
int   fgetc(FILE *fp);
char *fgets(char *s, int nmax, FILE *fp);
```

Scrittura

```
int   fprintf(FILE *fp, char *format, …);
int   fputc(int ch, FILE *fp);
int   fputs(char *s, FILE *fp);
```

fscanf - fprintf

```
int   fscanf(FILE *fp, char *format, …);
int   fprintf(FILE *fp, char *format, …);
```

Equivalgono a **scanf** e **printf**, solo che accedono al file puntato da **fp**. Restituiscono il numero di elementi letti/scritti, oppure **EOF** in caso di errore.

```
char  x, y = 100;
FILE  *fip, *fop;

      fip = fopen("file1.txt", "r");
      fop = fopen("file2.txt", "w");
      fscanf(fip, "%d", &x);
      fprintf(fop, "%d\n", y);
```

fgetc - fputc

```
int   fgetc(FILE *fp);
int   fputc(int ch, FILE *fp);
```

Leggono/scrivono un carattere nel file **fp**. Restituiscono il carattere letto/scritto (come intero) in caso di successo, **EOF** altrimenti.

```
char  c;
FILE  *fip, *fop;

      fip = fopen("file1.txt", "r");
      fop = fopen("file2.txt", "w");
      while ((c = fgetc(fip)) != EOF)
          fputc(fop);
```

fgets

```
char *fgets(char *s, int nmax, FILE *fp);
```

Trasferisce nella stringa **s** una linea di **(nmax-1)** caratteri dal file puntato da **fp**, fino ad incontrare **'\n'** o **EOF**.

- L'eventuale **'\n'** incontrato viene inserito in **s**.
- Dopo l'ultimo carattere letto, in **s** viene inserito **'\0'**.
- Restituisce **s** in caso di successo e **NULL** quando viene incontrato **EOF** senza che alcun carattere sia stato letto.

fputs

```
int   fputs(char *s, FILE *fp);
```

Trasferisce la stringa **s** nel file puntato da **fp**. Il carattere terminatore **'\0'**, non viene copiato.

- Restituisce l'ultimo carattere scritto in caso di terminazione corretta, **EOF** altrimenti.

File standard di I/O

Esistono tre file di testo aperti automaticamente all'inizio di ogni esecuzione:

stdin	standard input (tastiera), aperto in lettura;
stdout	standard output (video), aperto in scrittura;
stderr	standard error (video), aperto in scrittura;

stdin, **stdout**, **stderr** sono variabili di tipo puntatore a file definite in "stdio.h", pertanto non è necessario definirle.

fscanf - fprintf

Si noti che

```
int  scanf(FILE *fp, char *format, …);
int  printf(FILE *fp, char *format, …);
```

sono equivalenti a

```
int  fscanf(stdin, char *format, …);
int  fprintf(stdout, char *format, …);
```

File binari

Sono file che memorizzano dati di qualunque tipo (interi, reali, vettori o strutture).

- Per accedere ad un file binario è necessario aprirlo in con modalità **"b"**.
- Un file binario consente la lettura o scrittura di un intero blocco di dati attraverso le funzioni:

```
int  fread(void *v, int size, int n, FILE *fp);
int  fwrite(void *v, int size, int n, FILE *fp);
```

fread

```
int  fread(void *v, int size, int n, FILE *fp);
```

Legge al più **n** oggetti di dimensione **size** dal file puntato da **fp** e li trasferisce nel buffer puntato da **v**.

- Restituisce il numero di oggetti letti (se < **n**, vuol dire che si è verificato un errore o si è raggiunta la fine del file).

```
int   dati[100], n;
FILE  *fp;

      fp = fopen("dati.bin", "rb");
      n = fread(dati, sizeof(int), 100, fp);
      printf("dati letti = %d\n", n);
```

fwrite

```
int  fwrite(void *v, int size, int n, FILE *fp);
```

Scrive **n** oggetti di dimensione **size**, nel file puntato da **fp** prelevandoli dal buffer puntato da **v**.

- Restituisce il numero di oggetti effettivamente scritti (un numero minore di **n** indica che si è verificato un errore).

```
int   i;
FILE  *fp;

      fp = fopen("dati.bin", "wb");
      for (i=0; i<100; i++)
          fwrite(i, sizeof(int), 1, fp);
```

Esercizi

Scrivere i seguenti programmi:

- **> show nome** → stampa il file nome
- **> conta nome** → conta le parole del file nome
- **> ord nome** → ordina le parole del file nome
- **> cerca p nome** → cerca le occorrenze della parola p nel file nome

Generazione di numeri casuali

Numeri casuali

Il generatore di numeri casuali in C è basato su due funzioni, dichiarate in stdlib.h:

void srand(unsigned int seed);

inizializza il generatore per mezzo di un *seme* (seed). Inizializzazioni diverse con lo stesso seme generano la stessa sequenza.

int rand();

restituisce un numero pseudo-casuale tra 0 e RAND_MAX (dove RAND_MAX = 32767).

Numeri casuali

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

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

      printf("Inserire seme:");
      scanf("%d", &seed);
      srand(seed);

      for (i=0; i<10; i++) {
          a = rand();
          printf("a%d = %d\n", i, a);
      }
      return 0;
}
```

Numeri casuali

Per generare sequenze diverse ad ogni esecuzione, si inizializza srand con un valore temporale, ottenuto con la funzione time(), dichiarata in time.h:

long int time(NULL);

restituisce il numero di secondi trascorsi dalle ore 00:00 del 1 gennaio 1970.

NOTA: La funzione srand() dev'essere chiamata una sola volta, all'inizio del programma.

Numeri casuali

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

int   main()
{
int   i;
float x;

      srand(time(NULL));  /* iniz. sequenza */

      for (i=0; i<10; i++) {
          x = rand() / RAND_MAX;
          printf("x = %f\n", x);
      }
}
```

Cosa vi aspettate che stampi questo programma?

stampa 10 numeri uguali a zero

Numeri casuali

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

int   main()
{
int   a, n = 100;
float x;

      srand(time(NULL));  /* iniz. sequenza */

      /* genera un intero casuale in [0,n-1] */
      a = rand() % n;

      /* genera un float casuale in [0,1] */
      x = rand() / (float)RAND_MAX;
```

Esercizi

Scrivere i seguenti programmi:

> indovina n → chiede di indovinare un numero tra 0 e n

> primo n → genera un numero primo casuale tra 0 e n

> dadi n → simula il lancio di n dadi

> isto n → stampa l'istogramma dei risultati di 100 lanci di n dadi

Errori comuni

"=" != "=="

Qual è il risultato di questa funzione?

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

(n = 0) ≡ falso;
(n = 1) ≡ vero;

Si entra in un ciclo infinito => stack overflow

Controllare l'indice di array

```
int    v[10];
int    i;

       for (i=0; i<=10; i++) v[i] = 0;
```

L'ultimo elemento dell'array ha indice 9, non 10, pertanto si va ad azzerare una cella di memoria non voluta!

Attenzione agli apici

Le costanti di tipo carattere vanno indicate tra **apici**, mentre le costanti di tipo stringa tra **doppi apici**:

Si noti la differenza tra le seguenti dichiarazioni:

```
char   c = '1';
char   s[10] = "1";
int    i = 1;

       if (s[0] == "1")    /* ERRORE */
```

Quando manca il break

```
int  color;

     switch (color) {
        case 0: prinf("black");
        case 1: prinf("blue");
        case 2: prinf("green");
     }
```

Se color vale 1, la stampa prodotta è bluegreen, in quanto (mancando i **break**) il flusso prosegue con l'istruzione successiva. Se color vale 0, viene stampato blackbluegreen.

& mancante

Questo codice sporca la memoria allocata ad altre variabili:

```
int  a = 100;

   printf("Inserire un intero: ");

   scanf("%d", a);

   printf("a = %d\n", a);
```

a viene interpretato come indirizzo, e l'intero è scritto all'indirizzo 100, sporcando la memoria del sistema operativo (segmentation fault).

& di troppo

Questo codice sporca la memoria allocata ad altre variabili:

```
char s[50];

   printf("Inserire un nome: ");

   scanf("%s", &s);

   printf("s = %s\n", s);
```

s è già un puntatore che contiene l'indirizzo della stringa, per cui **&s** fornisce un indirizzo errato.

; di troppo

Si noti la differenza tra queste istruzioni, entrambe corrette:

```
   if (v[i] > max);
      max = v[i];
```

Viene sempre eseguita

```
   if (v[i] > max)
      max = v[i];
```

Viene eseguita solo se la condizione è vera.

Gli spazi sono utili!

```
int   a = 2;
int   b = 10;
int   x, y;
int   *p;

      p = &a       /* p punta ad a */;
      x = b/*p     /* divide b per a */;
      y = b / *p   /* divide b per a */;
```

Dopo le operazioni si ha che x = 10 e y = 5, in quanto

```
/*p      /* divide b per a */
```

è considerato come un commento!

Tipi diversi

Attenzione alle conversioni automatiche di tipo:

```
#define  LATO  40

int   x;
int   y = 5;

      x = LATO*(y/10);
```

produce x = 0

```
      x = LATO*(y/10.0);
```

produce x = 20

{ } possono fare la differenza

```
void  maius(char *s)
{
int   i = 0;

      while (s[i] != '\0')
          if ((s[i] >= 'a') && (s[i] <= 'z'))
              s[i] - 'a' + 'A';
          i++;
}
```

Questa funzione genera un ciclo infinito, in quanto il while (non avendo le parentesi graffe) contiene una sola istruzione, pertanto l'indice i non viene incrementato.

A chi è associato l'else?

```
if (x == 0)
    if (y == 0) error();
else {
    a = x + y;
    b = x/y;
}
```

In assenza di parentesi, l'**else** è sempre associato al più vicino **if**, per cui è interpretato come:

```
if (x == 0) {
    if (y == 0) error();
    else {
        a = x + y;
        b = x/y;
    }
}
```

A chi è associato l'else?

Se vogliamo che il codice venga eseguito come espresso dall'indentazione iniziale, dobbiamo scriverlo come segue:

```
if (x == 0) {
    if (y == 0) error();
}
else {
    a = x + y;
    b = x/y;
}
```

Controllo del range

Questa funzione ha due potenziali fonti di errore:

```
float  media(int *v, int n)
{
int    i, sum;
       sum = 0;
       for (i=0; i<n; i++)
            sum = sum + v[i];

       return sum/n;
}
```

Se n è grande, la somma può facilmente eccedere il range della variabile!

La divisione è fra due interi: pertanto il risultato viene prima troncato e poi convertito a float!

Controllo del range

Questa funzione è più robusta:

```
float  media(int *v, int n)
{
int    i
float  med;

       for (i=0; i<n; i++)
            med = (med*i + v[i])/(i+1);

       return med;
}
```

Il valore di med non dipende da n e la conversione a float viene fatta prima della divisione!

*s != s[10]

```
char *s;
char a = 'x';
int  n = 10;

     printf("Inserire stringa: ");
     scanf("%s", s);
```

s è solo un puntatore a carattere, manca la dimensione della stringa, per cui la memoria non viene allocata e la stringa va a sovrascrivere altre variabili.

Puntatore a fantasma

```
char  *copia(char *s)
{
char  dest[40];
int   i;

      i = 0;
      while (s[i] != '\0') {
          dest[i] = s[i];
          i++;
      }
      dest[i] = '\0';
      return dest;
}
```

dest è un'array locale e come tale viene distrutto all'uscita della funzione. Pertanto la funzione chiamante riceve un puntatore ad una zona di memoria non più valida.

Approccio corretto

Meglio fare allocare la memoria alla funzione chiamante:

```
char  s1[20] = "acqua", s2[20];

      copia(s1, s2);
```

```
void  copia(char *s, char *dest)
{
int   i = 0;
      while (s[i] != '\0') {
          dest[i] = s[i];
          i++;
      }
      dest[i] = '\0';
}
```