Tipi di dati

- Un tipo di dato è determinato da un *insieme* di elementi e di *operazioni* definite sugli elementi.
- In C i tipi fondamentali sono i tipi interi e i tipi reali.

Tipi interi

- Permettono la rappresentazione di insiemi di interi.
- I tipi fondamentali sono `int` e `char`.
- Gli altri tipi si ottengono tramite i qualificatori `signed`, `unsigned`, `short` e `long`.

```
char                    signed char           unsigned char
short (int)             int                   long (int)
unsigned short (int)    unsigned (int)        unsigned long (int)
```

I nomi fra parentesti possono essere omessi (es., `short` è sinonimo di `short int`).

- I tipi elencati differiscono esclusivamente nella *dimensione*; per conoscere la dimensione di un tipo, si può usare l'operatore `sizeof`.
- Un tipo intero `T` di dimensione n bit permette di rappresentare un insieme di 2^n numeri.

  Più precisamente:
  - Il tipo `unsigned T` rappresenta il sottoinsieme degli interi

    $$\{0, \ldots, 2^n - 1\}$$

  - Il tipo `signed T` rappresenta il sottoinsieme degli interi

    $$\{-2^{n-1}, \ldots, 2^{n-1} - 1\}$$

Alcuni vincoli sui tipi interi

- L'ANSI C non stabilisce la dimensione di tutti i tipi, ma che debbano essere rispettati alcuni vincoli:

  `sizeof(char)` $=$ 1*byte*

  `sizeof(short int)` $\leq$ `sizeof(int)` $\leq$ `sizeof(long int)`

  `sizeof(signed T)` $=$ `sizeof(unsigned T)` $=$ `sizeof(T)`

  dove `T` è un qualunque tipo intero.
- Il qualificatore `signed` è in genere omesso.
- I valori massimo e minimo di ciascun tipo sono determinati da costanti definite nel file `<limits.h>`, che possono essere utilizzate nei programmi.

La scelta del tipo di una variabile intera va effettuata in base ai valori che la variabile deve assumere.

Esempio

- In

  ```
  for(k=0; k<10; k++)
      ...
  ```

  la variabile `k` può essere dichiarate di tipo `short int` oppure `char`.
- Se una variabile intera `k` deve assumere valori grandi, conviene dichiararla di tipo `long int` oppure, se $k \geq 0$, di tipo `unsigned long int`.

Costanti intere

Esempi di costante intere di tipo `int`:

```
123     -10     +27
```

Una costante di tipo `long int` è una costante intera che termina con `l` (oppure `L`):

```
123456L        -232341
```

Vedere i manuali per la sintassi completa delle costanti.

Esercizio

Provare a eseguire il programma

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

int main(){
  short int s=0;
  printf("sizeof(short int) = %d\n", sizeof(short int));
  while(s <  SHRT_MAX){
    printf("s =  %d\n",  s);
    s = s + 10000;
  }
  return 0;
}
```

dove `SHRT_MAX` è il massimo intero rappresentabile con il tipo `short int` (vedere la definizione il `limits.h`). Cosa succede in esecuzione?

Assumendo che il tipo `short int` abbia dimensione 2 byte (=16 bit), i valori rappresentati sono gli interi compresi fra $-2^{15} = -32768$ e $2^{15} - 1 = 32767$ (valore di `SHRT_MAX`). In tale ipotesi, la somma $30000 + 10000$ dà come risultato -25536.

char e int

- Il C non fa distinzioni fra caratteri e interi, e su di essi sono definite le stesse operazioni (visione *poco astratta* del tipo di dato "carattere").
- La differenza fra i due tipi sta nel numero di valori che si possono rappresentare:
  - Il tipo `char` occupa 1 byte, quindi permette di rappresentare 2^8 valori (gli interi compresi fra 0 e $2^8 - 1$ oppure gli interi da -2^7 a $2^7 - 1$, dipende dal sistema).
  - Il tipo `int` occupa in genere 4 byte, quindi permette di rappresentare 2^{32} valori (gli interi compresi fra -2^{31} e $2^{31} - 1$).

Caratteri

- In C un carattere è una costante intera il cui valore è dato dal codice ASCII corrispondente.

  Esempio
  Il carattere `'a'` vale 97.
  Il carattere `'B'` vale 66.
  Il carattere `'0'` vale 48.
  Il carattere `' '` (*spazio*) vale 32.

- Il simbolo `\` (carattere di *escape*) permette di definire ulteriori caratteri.

  Esempio
  `'\n'` è il carattere *newline* (nuova linea) e vale 10.
  `'\a'` è il carattere *bell* (segnale acustico) e vale 7.
  `'\0'` è il carattere *nullo* e vale 0.

char = signed char?

- Se `char` equivale a `signed char`, il tipo `char` permette di rappresentare gli interi n tali che:

$$-2^7 \leq n \leq 2^7 - 1$$

- Se `char` equivale a `unsigned char`, il tipo `char` permette di rappresentare gli interi n tali che:

$$0 \leq n \leq 2^8 - 1$$

Per verificarlo, provare a eseguire:

```
char c = -1;
signed char s = -1;
unsigned char u = -1;
printf("c = %d, s = %d, u = %d\n", c, s, u);
```


Dopo le inizializzazioni, in memoria si ha:

```
Variabile    Valore

   c         11111111   (= rappr. di -1 nell'aritmetica complemento a 2)
   s         11111111
   u         11111111
```

- Se `char` è sinonimo di `signed char`, il valore di `c` viene interpretato come un "intero con segno" e viene stampato

  `c = -1, s = -1, u = 255`

- Se `char` è sinonimo di `unsigned char`, il valore di `c` viene interpretato come un intero positivo.
  Poiché il byte 1111 1111 corrisponde all'intero $2^8 - 1 = 255$, viene stampato.

  `c = 255, s = -1, u = 255`

Generalmente `char` equivale a `signed char`.

Esempio 1

```
char c; int i;
```

I seguenti assegnamenti sono leciti (anche non hanno alcun significato come operazioni su caratteri)

```
c = 'a'; /* c vale 97 */

i = 'a' * 10 + 1;  /* i vale 97*10+1 = 971 */

i = i - 'i';    /* i vale 971-105 = 866  */

c = c + '0'   /* c vale 97 + 48 = 145 */
```


Esempio 2

L'istruzione di alto livello

char c;
IF(c e' una lettera minuscola)
  THEN *Istr*

si traduce in

```
if(c>= 97 && c<= 122)
    Istr
```

oppure

```
if(c>= 'a' && c<= 'z')
    Istr
```

Le istruzioni sono equivalenti in quanto `'a'` vale 97, `'z'` vale 122.
La seconda soluzione è preferibile in quanto il codice è più comprensibile.

Esempio 3

char c;
IF(c e' una vocale minuscola)
  THEN $lstr_1$
  ELSE $lstr_2$

si traduce in

```
if(c== 97 || c== 101 || c== 105 || c== 111 || c== 117)
    Istr1
 else
    Istr2
```

oppure *(meglio)*

```
if(c== 'a' || c== 'e' || c== 'i' || c== 'o' || c== 'u')
  ....
```


Riconoscimento di caratteri

L'ANSI C dispone di funzioni per classificare i caratteri (occorre includere `<ctype.h>`), ad esempio:

Funzione	Valore restituito
`isalpha(c)`	$k \neq 0$ se c è una lettera, 0 altrimenti
`isupper(c)`	$k \neq 0$ se c è una lettera maiuscola, 0 altrimenti
`islower(c)`	$k \neq 0$ se c è una lettera minuscola, 0 altrimenti
`isdigit(c)`	$k \neq 0$ se c è una cifra, 0 altrimenti
`isalnum(c)`	$k \neq 0$ se c è un carattere *alfanumerico* (lettera o cifra), 0 altrimenti
`isspace(c)`	$k \neq 0$ se c è un carattere di spaziatura, 0 altrimenti
...	...

Nota

Essendo il valore di k non specificato dall'ANSI C (non è detto che k sia 1), evitare di scrivere istruzioni del tipo

```
if(isspace(c) == 1)
  Istr 1
else
  Istr 2
```

La versione corretta è

```
if(isspace(c))
  Istr 1
else
  Istr 2
```

Infatti:

- Se c è un carattere di spaziatura, `isspace(c)` restituisce 0 e viene eseguita `Istr2`.
- Altrimenti, `isspace(c)` restituisce k ed, essendo $k \neq 0$, viene eseguito `Istr1`.

I tipi reali

L'ANSI C definisce tre tipi reali:

`float` `double` `long double`

- I tipi reali permettono la rappresentazione di valori non interi.
- Non confondere i tipi reali con i numeri reali.
  I numeri reali in genere richiedono una descrizione infinita, quindi possono essere rappresentati da un valore di *tipo reale* solamente in modo approssimato.
- Come per i tipi interi, la differenza fra i tipi reali consiste nella loro dimensione (verificarla con `sizeof`), quindi nel numero di valori rappresentabili (vedere i dettagli sul libro).
- In genere si usa il tipo `double`

Costanti reali

`1.234` denota una costante di tipo `double`

`2.5F` (oppure `2.5f`) denota una costante di tipo `float`

`1.2367L` (oppure `1.23671`) denota una costante di tipo `long double`

È possibile usare la notazione esponenziale:

`1.2345e6` *rappresenta il numero* $1,2345 \times 10^6$

`45.34e-3` *sta per* $45,34 \times 10^{-3}$.

Nota

Per denotare una costante di tipo `double` occorre usare il punto.
Ad esempio:

`3` ha tipo `int` (e valore 3)

`3.0` ha tipo `double` (e valore 3)

Operatori aritmetici

Espressione	Valore
E_1 `+` E_2	Valore(E_1) $+$ Valore(E_2)
E_1 `-` E_2	Valore(E_1) $-$ Valore(E_2)
E_1 `*` E_2	Valore(E_1) $*$ Valore(E_2)
E_1 / E_2	Divisione fra Valore(E_1) e Valore(E_2)
E_1 `%` E_2	Resto della divisione intera fra Valore(E_1) e Valore(E_2)

- L'operatore `%` (modulo) può essere applicato solamente a espressioni intere.
- L'operatore `/` denota sia la divisione intera che la divisione non intera (overloading).
  Il significato è determinato dal *tipo* degli operandi.

Esempi

```
char a = 10;  int  b = 'a';
```

Espressione	Valore
`a + 87`	97 (o, equivalentemente, 'a')
`a + 'a'`	107 (oppure 'k')
`'X'-'x'`	32
`'D'+ ('X'-'x')`	100 (oppure 'd')
`'a'- ('F'-'f')`	65 (oppure 'A')
`a- ('F'-'f')`	-22
`b - ('S'-'s')`	65 (oppure 'A')
`'b' - ('S'-'s')`	66 (oppure 'B')
`b + 1`	98 (oppure 'b')
`b + '1'`	146
`'b'+ 1`	99 (oppure 'c')
`'b'+ '0'`	146
`a / 7`	1
`a % 7`	3
`'a' % '7'`	42

Tipo delle espressioni aritmetiche

Consideriamo l'espressione aritmetica

$$E_1 \odot E_2$$

dove E_1 ha tipo T_1, E_2 ha tipo T_2 e $\odot$ è un operatore aritmetico.

- Se $T_1 = T_2$, l'espressione $E_1 \odot E_2$ ha tipo T_1.
- Se $T_1 \neq T_2$, l'espressione è detta *mista*.
  Quando l'espressione è valutata, viene effettuata un cast implicito dei valori di E_1 ed E_2 ad uno stesso tipo T (in genere $T = T_1$ oppure $T = T_2$).

  Denotando con (T) l'operazione di "cast (conversione) al tipo T":

  $$E_1 \odot E_2 \quad \text{diventa} \quad (\,(T)\,E_1\,) \odot (\,(T)\,E_2\,)$$

  L'espressione ha tipo T.

Le regole di conversione sono complesse perché devono prevedere tutti i possibili casi per T_1 e T_2

In genere, la conversioni implicite trasformano un elementi di un tipo in un elemento di un tipo "più grande" (non c'è perdita di informazione). Ad esempio, se T_1 è `int` e T_2 è `double`, il tipo comune T è `double`.
Vedere i dettagli sui manuali.

Nota
Il tipo di un'espressione è determinabile a compile-time (infatti, dipende esclusivamente dalle dichiarazioni presenti nel programma e non dall'esecuzione).

Esempi

```
char c=20; short s=10; int i=1000;
float f=1.5; double d=-2.5;  long double ld=2.33;
```

Espressione	Tipo dell'espressione	Valore
`s + i`	`int`	1010
`c + 5`	`int`	25
`c + 'd'`	`int`	120
`c + 5.1`	`double`	25.1
`s * i`	`int`	10000
`i / 3`	`int`	333
`i / 3.0`	`double`	333.33...
`d - ld`	`long double`	-4.83
`f + 'f'`	`float`	103.5

In

```
i / 3.0
```

essendo `3.0` una costante di tipo `double`, il valore di di `i` viene convertito implicitamente al tipo `double` e `/` denota la divisione non intera.

Conversioni di tipo esplicite

È sempre possibile effettuare dei cast espliciti di espressioni usando l'operatore unario '(..)' (dove fra parentesi va scritto il tipo a cui si vuole convertire l'espressione).

Esempio

- L'espressione
  ```
  'a' + 5.0
  ```
  ha tipo `double` e valore 102 ($= 97 + 5$).
  Per convertire il valore di `'a'+5.0` a `int` occorre fare:
  ```
  (int) ('a' + 5.0)
  ```
- Attenzione alle parentesi:
  ```
  (int) 'a' + 5.0
  ```
  equivale a
  ```
  ((int) 'a') + 5.0
  ```
  che ha tipo `double`.

Cast pericolisi

Alcune conversioni creano perdita di informazione. Ad esempio, se `d` è una variabile di tipo `double`, il cast

```
(int)  d
```

è sintatticamente corretto; tuttavia, se il valore di `d` non è intero, si ha perdita di informazione. Va usato solo se `d` ha valore intero.
Non sempre il compilatore segnala i cast pericolosi!

```
int i; double d;
i =  (int) d ; /* corretto, anche se pericoloso */
(int) d = i  ; /*  scorretto */
```

L'ultima istruzione non è sintatticamente corretta perché non si può fare il cast di una variabile a sinistra di un assegnamento.

Esercizio

Dire cosa fa ciascuna delle seguenti istruzioni. Verificare le risposte con l'aiuto di un programma; per capire il motivo di alcuni strani comportamenti, compilare usando opzione `-Wall` (si tenga presente che le conversioni di formato non provocano il cast implicito dei corrispondenti argomenti).

```
int i = 11;
double d = 2.32;

printf("%f\n", 1);
printf("%f\n", (double) 1);
printf("%f\n",  1 + 0.0);
printf("%d\n",  1.00);
printf("%d\n",  (int)1.00);
printf("%d\n",  (int)1.25);
printf("%d\n", d);
printf("%d\n",  (int) d);
printf("%d\n",  i/2 );
printf("%d\n",  i/2.0 );
printf("%f\n",  i/2.0 );
printf("%f\n",  i /((double)  2));
printf("%f\n",  (double) (i/2) );
printf("%f\n",  ((double) i) /2);
```


Standard input

Le operazioni di lettura avvengono tramite lo standard input, che è una sequenza (*stream*) di caratteri.

- Per default, lo standard input è associato alla tastiera.
- È possibile chiedere al *sistema operativo* di associare lo standard input a un file esterno (redirezione dello standard input).

Esempio

Con il comando

```
a.exe < in.txt
```

si chiede alla shell di mandare in esecuzione il programma `a.exe` redirigendo lo standard input da `in.txt` (si noti che il programma `a.exe` non è consapevole di leggere l'input da un file esterno).

Standard output

Le operazioni di scrittura avvengono tramite lo standard output (*stream* di caratteri).

- Per default, lo standard output è associato allo schermo.
- È possibile redirigere lo standard output su un file.

Esempio

```
a.exe > out.txt
```

l'output di `a.exe` è scritto nel file `out.txt`.
Se il file `out.txt` non esiste, viene creato.
Se il file `out.txt` esiste già, il suo contenuto è perso.

```
a.exe >> out.txt
```

A differenza del caso precedente, se `out.txt` esiste già il suo contenuto non è perso.

`getchar()`

La funzione standard `getchar()` legge un carattere da standard input e restituisce il carattere letto.

Esempio

Supponiamo che lo standard input sia la sequenza di caratteri

A	2		c	d	@	.....

(la casella bianca contiene il carattere spazio) e di eseguire

```
char a,b,c,d;
a = getchar();
b = getchar();
c = getchar();
d = getchar();
```

Al termine delle istruzioni la variabile `a` contiene il carattere `'A'` (ossia, il valore 65), `b` contiene il carattere `'2'` (valore 50), `c` contiene il carattere spazio (ovalore 32), `d` contiene il carattere `'c'` (valore 99). La prossima operazione di lettura da standard input partirà dal carattere `'d'`.

putchar()

La funzione standard putchar() stampa su standard output il carattere passato come argomento.

Esempio

```
putchar('a');
```

stampa su standard output

a

Si ottiene la stessa cosa facendo

```
putchar(97);
```

mentre non è sintatticamente corretta la chiamata

```
putchar('97');
```

in quanto '97' non è un carattere.


```
putchar('\n');
```

oppure

```
putchar(10);
```

stampa il carattere "a capo", e la prossima operazione di scrittura avverrà su una nuova linea.

```
putchar('\a');
```

oppure

```
putchar(7);
```

produce un segnale acustico.

End-of-file

Fra i caratteri leggibili da standard input c'è anche il segnale di "end-of-file".

- Se lo standard input è associato alla tastiera, il carattere "end-of-file" può essere inserito con apposite combinazioni di tasti (in genere, $<$ *control* $>+d$ con Unix, $<$ *control* $>+z$ con MS-DOS), da non confondere con il segnale di interruzione del programma (tipicamente, $<$ *control* $>+c$).
- Se lo standard input è rediretto da un file, il carattere "end-of-file" corrisponde al segnale di fine file.

Quando `getchar()` legge il carattere "end-of-file" restituisce il valore dell'identificatore EOF, definito in `stdio.h` (verificarlo).

Tipico valore di `EOF` è -1 (non può essere un valore positivo perché si confonderebbe con i valori dei caratteri).

```
/* stdio.h  */
...
# define EOF (-1)
...
```


Esempio

```
#include <stdio.h>

int main(){
  int c;
  c = getchar();
  while(c!= EOF){
       putchar(c);
       c = getchar();
   }
  printf("Fine\n");
  return 0;
}
```

Il programma stampa su standard output i caratteri letti da standard input e termina quando il carattere letto è "end-of-file". Quando `getchar()` restituisce `EOF`, il ciclo `while` termina, viene emesso un segnale acustico e viene stampata la parola `Fine`.
Il programma termina restituendo 0 al sistema (se invece il programma è interrortto, viene restituito un valore diverso).

Nota 1

La variabile `c` è stato dichiarata di tipo `int` anziché `char`.
Infatti, in

```
c = getchar();
```

la variabile c deve memorizzare non solo i valori corrispondenti ai caratteri, ma anche il valore della costante `EOF`.
Essendo `EOF` un valore negativo, occorre usare un tipo intero che permetta la rappresentazione degli interi negativi.

Nota 2

Quando lo standard input è associato alla tastiera, quando viene premuto il tasto $<$ *Invio* $>$ per inserire i dati viene introdotto su standard input il carattere *newline*.
Ad esempio, supponiamo di eseguire le linee di codice

```
int a1,a2,a3,a4,a5;
  a1 = getchar();
  a2 = getchar();
  a3 = getchar();
  a4 = getchar();
  a5 = getchar();
  printf("a1 = %d\n", a1);
  printf("a2 = %d\n", a2);
  printf("a3 = %d\n", a3);
  printf("a4 = %d\n", a4);
  printf("a5 = %d\n", a5);
```

con standard input associato alla tastiera.

Se si premono in successione i tasti $<$ *a* $>$, $<$ *Spazio* $>$, $<$ *b* $>$, $<$ *Invio* $>$, $<$ *c* $>$, $<$ *Invio* $>$, $<$ *d* $>$, viene stampato:

```
a1 = 97
a2 = 32
a3 = 98
a4 = 10
a5 = 99
```

Gli ultimi due caratteri su standard input ($<$ *Invio* $>$ e $<$ *d* $>$) sono ancora su sandard input, ma non sono stati letti.

Nota 3

- `getchar()` e `putchar()` sono gli strumenti base per le operazioni di lettura e scrittura.
- Poiché tali funzioni permettono la lettura e scrittura di *un singolo* carattere per volta, il loro uso diventa macchinoso se occorre trattare dati sintatticamente più compessi (numeri interi, numeri con virgola, stringhe,...).
- L'ANSI C mette a disposizione delle funzioni più potenti (e più complesse da usare) che permettono la scrittura e lettura di elementi di qualunque tipo.
  Nel corso utilizzeremo:
  - `scanf()` per la lettura.
  - `printf()` per la scrittura.

La funzione di output `printf()`

La funzione `printf()` (*formatted print*) permette di stampare valori in diversi formati. Può avere un numero arbitrario di argomenti.

- Il primo argomento è una *stringa di controllo* contenuta fra doppi apici. La funzione stampa i caratteri contenuti nella stringa di controllo, tranne quelli preceduti da % (*specifica di conversione*).
- Le specifiche di conversione si riferiscono agli argomenti di `printf()` scritti dopo la stringa di controllo. Al posto di una specifica di conversione viene stampato il valore del corrispondente argomento secondo il formato specificato.
- La funzione restituisce il numero di caratteri stampati.

Nota
L'ANSI C non obbliga a utilizzare il valore restituito da una funzione.

Carattere di conversione	Formato con cui è stampato il corrispondente argomento
d	intero in notazione decimale
u	intero senza segno in notazione decimale
c	carattere
o	intero ottale
x	intero esadecimale
f	numero reale
e	numero reale in notazione scientifica
...	...

Per la documentazione completa si raccomanda la lettura del libro di testo (o di un manuale).

Le istruzioni

```
printf("%c", n);        putchar(n);
```

hanno lo stesso effetto

Esempio

Le linee di codice

```
int n = 48, k;
k = printf("Valore di n: %c %d %o %x\n", n, n, n, n);
printf("Valore di k: %d\n",k);
```

danno in output

```
Valore di n: 0 48 60 30
Valore di k: 24
```

Nella prima linea viene stampato il valore di `n` in quattro diversi formati: carattere, intero, ottale ed esadecimale. Il valore restituito dalla prima chiamata a `printf()` (assegnato a `k` e stampato con la seconda `printf()`) è 24, infatti vengono stampate 9 lettere, 7 cifre, 6 spazi, il carattere ':' e il carattere "a capo" (`'\n'`).

Le istruzioni

```
double d = 0.72;
printf("Valore di d: %f = %e\n", d, d);
```

stampano

```
Valore di d: 0.720000 = 7.200000e-01
```

(`7.200000e-01` significa $7.2 * 10^{-1}$).

L'istruzione

```
printf("a%clbe%co",'l','r' );
```

stampa la parola `albero`.

Si ottiene lo stesso effetto con una delle seguenti istruzioni:

```
printf("a%clbe%co", 108, 'r');
printf("a%clbe%co", 'l', 114);
printf("a%clbe%co", 108, 114);
```

Invece

```
printf("a%clbe%do", 108, 'r');
```

stampa `albe114o`.

Esercizio

Cosa stampa il seguente programma?

```
#include <stdio.h>

int main(){
    int a, b, c, d, e, n, m;
    a = 49;
    b = 65;
    c = 97;
    d = 7;
    e = 10;
    printf(" %d %d %d %d %d", a, b, c, d, e);
    printf(" %c %c %c %c %c", a, b, c, d, e);
    a = '1';
    b = 'A';
    c = 'a';
    d = '\a';
    e = '\n';
    n = printf(" %d %d %d %d %d", a, b, c, d, e);
    m = printf(" %c %c %c %c %c", a, b, c, d, e);
    printf("m = %d  n = %d \n", m, n);
    return 0;
}
```


La funzione di input `scanf()`

La funzione `scanf()` effettua una o più operazioni di lettura da standard input interpretando i caratteri letti secondo il formato richiesto. Può avere un numero arbitrario di argomenti.

- Il primo argomento è *una stringa di controllo* (racchiusa fra doppi apici) contenente i *caratteri di conversione* (preceduti da %) che specificano come vanno interpretati i dati in input. Se una operazione di lettura ha successo, i dati sono memorizzati nella locazione di memoria di cui si è passato come argomento l'indirizzo.
- Restituisce il numero di operazioni di lettura effettuate con successo oppure `EOF` se nella prima operazione di lettura incontra "end-of-file".

Carattere di conversione	Formato con cui è letto il corrispondente argomento
d	intero in notazione decimale
c	carattere
lf	numero reale (di tipo `double`)
...	...

Le istruzioni

```
a = getchar();        scanf("%c", &a);
```

hanno lo stesso effetto.

- La lettura è un tipico esempio di funzione i cui parametri vanno passati "per indirizzo", va quindi passato l'*indirizzo* della locazione di memoria in cui si vuole memorizzare il valore letto.
  Ad esempio, per leggere un intero nella variabile `x` di tipo `int` occorre fare:

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

  Cosa succede scrivendo `x` al posto di `&x`?
- Con la specifica `%d`, vengono automaticamente saltati i caratteri di spaziatura (spazio, a capo, tabulazioni, ...) presenti su standard input prima dell'intero da leggere. Stesso discorso per la specifica `%lf`.
- Con la specifica `%c` viene invece letto il carattere corrente dello standard input, qualunque esso sia (come avviene usando `getchar()`).

Esempio 1

```
int a=0, b=0, c=0;
scanf("%d%d%d", &a, &b, &c);
```

L'istruzione richiede la lettura di tre interi da standard input, i cui valori vanno memorizzati rispettivamente nelle variabili a, b e c. Gli eventuali caratteri di spaziatura (spazio, a capo, tabulazioni,...) che precedono un intero sono automaticamente saltati.

Se lo standard input contiene

```
  12          -124
     +128       234
```

al termine dell'operazione di lettura si ha:

Variabile	Valore
a	12
b	-124
c	128

Il valore restituito da `scanf()` è 3 (valore non utilizzato).
La successiva operazione di lettura parte dallo spazio che segue l'ultima cifra letta.

Quindi, se viene eseguita l'istruzione

```
scanf("%c", &a);    /* OPPURE:  a = getchar();  */
```

si ha:

Variabile	Valore
a	32 (carattere spazio)
b	-124
c	128

Se invece si esegue

```
scanf("%d", &a);    /* OPPURE:  a = getchar();  */
```

si ha:

Variabile	Valore
a	234
b	-124
c	128

Esempio 2

```
int a=0, b=0, c=0;
scanf("%d%d%d", &a, &b, &c);
```

Se lo standard input contiene

```
  12-124+a128      234
```

al termine dell'operazione di lettura si ha:

Variabile	Valore
a	12
b	-124
c	0

e il valore restituito da `scanf()` è 2.

Infatti, sono state effettuate con successo due letture di interi. La lettura del terzo intero è fallita in quanto dopo il carattere `'+'` non è stata trovata alcuna cifra.

La prossima operazione di lettura parte dal carattere `'a'` (carattere su cui la precedente lettura è fallita), quindi se l'istruzione è

```
scanf("%c", &c);
```

la variabile c vale 97 (valore della costante `'a'`).

Se invece si esegue

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

l'operazione di lettura fallisce (`scanf()` restituisce 0 e il valore di c non cambia) e la prossima operazione di lettura parte di nuovo dal carattere `'a'` sullo standard input.

Esercizio

Eseguire il programma

```
#include <stdio.h>

int main (void){
  int a, b, c, k;
  k = scanf("%d%d%d", &a, &b, &c);
  printf("k = %d\n", k);
  c = getchar();
  putchar(c);
  printf("\n a = %d, b = %d, c = %d\n", a, b, c);
  return 0;
}
```

con varie sequenze di input, ad esempio

```
12  -23  7876

612-2-45

-5-2w

10 w-23

-2-3-a
```

Motivare l'output ottenuto.