Tipi primitivi del C

Un tipo è costituito da un insieme di **valori** ed un insieme di **operazioni** su questi valori.

Classificazione dei tipi primitivi del C

- scalari
  - aritmetici
    - interi: con segno, senza segno, caratteri, enumerati
    - reali
  - puntatori
- aggregati
  - array
  - strutture
  - unione
- funzione
- void (nessun valore e nessuna operazione)

Tipi aritmetici del C: per ciascun tipo consideriamo i seguenti **aspetti**:

1. intervallo di definizione
2. notazione per le costanti (nel codice, oppure in input/output)
3. operatori
4. predicati (operatori di confronto)
5. funzionalità di ingresso/uscita

Tipi interi

L'ANSI C richiede che gli **interi** siano **codificati in binario** (altrimenti molte delle operazioni a basso livello non sarebbero possibili).

Interi con segno

3 tipi: `short` (oppure `short int`, `signed short`, `signed short int`)
`int` (oppure `signed int`, `signed`)
`long` (oppure `long int`, `signed long`, `signed long int`)

Intervallo di definizione: da -2^{n-1} a $2^{n-1}-1$, dove n dipende dal compilatore

Vale che: `sizeof(short)` $\leq$ `sizeof(int)` $\leq$ `sizeof(long)`
`sizeof(short)` ≥ 2 (ovvero, almeno 16 bit)
`sizeof(long)` ≥ 4 (ovvero, almeno 32 bit)

LccWin32 e gcc: `short`: 16 bit, `int`: 32 bit, `long`: 32 bit

I valori limite sono contenuti nel file `limits.h`, che definisce le costanti:
`SHRT_MIN`, `SHRT_MAX`, `INT_MIN`, `INT_MAX`, `LONG_MIN`, `LONG_MAX`

Notazione per le costanti: in decimale: `0`, `10`, `-10`, ...
Per distinguere long (solo nel codice): `10L` (oppure `10l`, ma `l` sembra `1`).

Operatori: `+`, `-`, `*`, `/`, `%`, `==`, `!=`, `<`, `>`, `<=`, `>=`

Ingresso/uscita: tramite `printf` e `scanf`, con i seguenti specificatori di formato (dove `d` indica "decimale"):
`%hd` per `short`
`%d` per `int`
`%ld` per `long` (con `l` minuscola)

Esercizio: Compilare ed eseguire il file `tipi/intlim.c`.

Interi senza segno

3 tipi: `unsigned short` (oppure `unsigned short int`)
`unsigned int` (oppure `signed`)
`unsigned long` (oppure `unsigned long int`)

Intervallo di definizione: da 0 a $2^n - 1$, dove n dipende dal compilatore.
Il numero n di bit è come per i corrispondenti interi con segno.

Le costanti definite in `limits.h` sono:
`USHRT_MAX`, `UINT_MAX`, `ULONG_MAX` (si noti che il minimo è sempre 0)

Esercizio: Compilare ed eseguire il file `tipi/unsiglim.c`.

Notazione per le costanti:

- decimale: come per interi con segno
- esadecimale: `0xA`, `0x2F4B`, ...
- ottale: `012`, `027513`, ...

Nel codice si può far seguire le cifre dallo specificatore `u` (ad esempio `10u`).

Ingresso/uscita: tramite `printf` e `scanf`, con i seguenti specificatori di formato:

- `%u` per numeri in decimale
- `%o` per numeri in ottale
- `%x` per numeri in esadecimale con cifre `0`, ..., `9`, `a`, ..., `f`
- `%X` per numeri in esadecimale con cifre `0`, ..., `9`, `A`, ..., `F`

Per interi `short` si antepone `h`
`long` si antepone `l` (minuscola)

Operatori: tutte le operazione vengono fatte modulo 2^n.

Attenzione alla conversione tra `signed` e `unsigned` in operazioni miste: **il valore `signed` viene promosso a `unsigned`**.

Esempio:
```
unsigned int u;
if (u > -1) ...
```

La condizione `(u > -1)` è **sempre falsa** perchè `-1` viene convertito in `unsigned` e diventa il più grande `int` (numero con bit pari a $111\cdots111$).

Caratteri

Servono per rappresentare caratteri alfanumerici attraverso opportuni **codici**, tipicamente il codice **ASCII** (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange).

Un codice associa ad ogni carattere un intero:
Esempio: Codice ASCII:

carattere:	`'#'`	`'0'`...`'9'`	`';'`	`'A'`...`'Z'`	`'a'`...`'z'`	`{`	`}`	...
intero (in decimale):	35	$48\cdots57$	59	$65\cdots90$	$97\cdots122$	123	125	...

In C i caratteri possono essere **usati come gli interi** (un carattere coincide con il codice che lo rappresenta).

3 tipi: `char`, `signed char`, `unsigned char`.

I tipi `signed char` e `unsigned char` hanno lo stesso tipo di rappresentazione degli interi (rispettivamente complemento a 2 e binario).

Il tipo `char` può essere (dipende dall'implementazione):

- signed (come `signed char`)
- unsigned (come `unsigned char`)
- un misto tra i due (solo valori nonnegativi, ma come signed per le conversioni)

Esempio:

```
unsigned char uc = -1;    int i = uc;
signed char   sc = -1;    int j = sc;
char           c = -1;    int k = c;

printf("i = %d,  j = %d,  k = %d\n", i, j, k);
```

Se `char` equivale a `signed char` stampa: `i = 255,  j = -1,  k = -1`

Se `char` equivale a `unsigned char` stampa: `i = 255,  j = -1,  k = 255`

Esercizio: Stabilire se in LccWin32 i caratteri sono rappresentati con o senza segno.

Intervallo di definizione: dipende dal compilatore

Vale che: `sizeof(char)` $\leq$ `sizeof(int)`
Tipicamente i caratteri sono rappresentati con 8 bit.

Esercizio: Compilare ed eseguire il file `tipi/charlim.c`.

Operatori: sono gli stessi di `int` (operazioni effettuate utilizzando il codice del carattere).

Costanti: `'A'`, `'#'`, ...

Esempio:

```
char x, y, z;        posso usare?
x = 'A';             x = 65;
y = '\n';            y = 10;
z = '#';             z = 35;
```

Non è sbagliato, però è **pessimo stile** di programmazione.
Non è detto che il codice sia ASCII. $\Longrightarrow$ Il programma **non sarebbe portabile**.

Ingresso/uscita: tramite `printf` e `scanf`, con specificatore di formato `%c`

Attenzione: in ingresso non vengono saltati gli spazi bianchi e gli a capo

Esempio:

```
int i, j;
char c;
printf("Immetti due interi\n");
scanf("%d%c%d", &i, &c, &j);
printf("%d %d %d\n", i, c, j);
```

```
Immetti due interi
# => 18 25^
18 32 25
```

```
Immetti due interi
# => 18a25^
18 97 25
```


Funzioni per la stampa e lettura di un singolo carattere:

- `putchar(c);` ... stampa il carattere memorizzato in `c`
- `c = getchar();` ... legge un carattere e lo assegna alla variabile `c`

Esempio:

```
char c;
putchar('A');
putchar('\n');
c = getchar();
putchar(c);
```

Esempio: Leggere da tastiera due interi x ed y ed un carattere op che rappresenta un operatore tra $+$, $-$, $*$, $/$, e stampare $x\ op\ y$.

Implementazione: file `tipi/charop.c`

Tipi reali

I reali vengono rappresentati in virgola mobile (floating point).

Intervallo di definizione: 3 tipi, le cui caratteristiche sono stabilite da uno standard ISO/IEEE:

	sizeof	cifre significative	min esp.	max esp.
`float`	4	6	-37	38
`double`	8	15	-307	308
`long double`	12	18	-4931	4932

Le caratteristiche sono descritte nel file `float.h`.

Esercizio: Compilare ed eseguire il file `tipi/floatlim.c`

Costanti: con punto decimale o notazione esponenziale

Esempio:
```
double x, y, z, w;
x = 123.45;
y = 0.0034;        y = .0034;
z = 34.5e+20;      z = 34.5E+20;
w = 5.3e-12;
```


Nei programmi, per denotare una costante di tipo

- `float`, si può aggiungere `f` o `F` finale
  Esempio: `float x = 2.3e5f;`
- `long double`, si può aggiungere `L` o `l` finale
  Esempio: `long double x = 2.34567e520L;`

Operatori: stessi che per gli interi (tranne "`%`")

Ingresso/uscita: tramite `printf` e `scanf`, con diversi specificatori di formato

Output con `printf` (per `float`):

- `%f` ... **notazione in virgola fissa**
  `%8.3f` ... 8 cifre complessive, di cui 3 cifre decimali

  Esempio:
  ```
  double x = 123.45;
  printf("|%f| |%8.3f| |%-8.3f|\n", x, x, x);
  ```
  ```
  |123.449997| | 123.450| |123.450 |
  ```


- `%e` (oppure `%E`) ... notazione **esponenziale**
  `%10.3e` ... 10 cifre complessive, di cui 3 cifre decimali

  Esempio:
  ```
  double x = 123.45;
  printf("|%e| |%10.3e| |%-10.3e|\n", x, x, x);
  ```
  ```
  |1.234500e+02| | 1.234e+02| |1.234e+02 |
  ```
- `%g` (oppure `%G`)... forma ottimizzata
  - come `%e` se l'esponente è < -4 oppure il numero è $\geq$ precisione specificata
  - altrimenti come `%f`

  Però: `%8.3g` ... 8 cifre complessive, di cui 3 cifre significative (non necessariamente dopo la virgola)

Input con `scanf` (per `float`): si può usare indifferentemente `%f`, `%e`, o `%g`.

Riassunto degli specificatori di formato per i tipi reali:

	`float`	`double`	`long double`
`printf`	`%f`, `%e`, `%g`	`%f`, `%e`, `%g`	`%Lf`, `%Le`, `%Lg`
`scanf`	`%f`, `%e`, `%g`	`%lf`, `%le`, `%lg`	`%Lf`, `%Le`, `%Lg`

Conversioni di tipo

Situazioni in cui si hanno conversioni di tipo

- quando in un'espressione compaiono operandi di tipo diverso
- durante un'assegnazione `x = y`, quando il tipo di `y` è diverso da quello di `x`
- nel passaggio dei parametri a funzione
- attraverso il valore di ritorno di una funzione
- esplicitamente, tramite l'operatore di **cast**

Una conversione può o meno coinvolgere un **cambiamento nella rappresentazione** del valore.

Esempio: si ha cambiamento: da `short` a `long` (dimensioni diverse)
da `int` a `float` (anche se stessa dimensione)
non si ha cambiamento: da `int` a `unsigned int`

Conversioni implicite tra operandi di tipo diverso nelle espressioni

`short` → `int` → `long` → `float` → `double` → `long double`
`char` → `int`

Operando di tipo più a sinistra viene convertito nel tipo più a destra, che è anche il tipo del risultato.

Esempio: `int x; double y;`
Nel calcolo di `(x+y)`:

1. `x` viene convertito in `double`
2. viene effettuata la somma tra valori di tipo `double`
3. il risultato è di tipo `double`

Inoltre: `signed` (`short`/`long`) viene convertito in `unsigned` (`short`/`long`)

N.B. Se il valore `signed` è negativo, il valore `unsigned` risultante sarà un numero positivo con bit più significativo pari a 1.

Conversioni sicure e non

- Le conversioni di sopra sono **sicure**: non si perde il valore.
  Eccezione: da `long` a `double` ci può essere perdita di precisione.
- Se invertiamo le conversioni sicure si può perdere il valore:
  `short` ← `int` ← `long` ← `float` ← `double` ← `long double`
  `char` ← `int`
  - ci può essere overflow
  - da reali a interi si ha troncamento della parte frazionaria

Conversioni in assegnazione

- Il risultato viene **sempre** convertito nel tipo della variabile a sinistra (con eventuale overflow o troncamento).
- Se la conversione non è possibile si ha errore.

Esempio:

```
int i;
float x = 2.3,  y = 4.5;
i = x + y;
printf("%d", i);            /* stampa 6 */
```

*Conversioni esplicite (operatore di **cast**)*

Sintassi:

```
(tipo)espressione
```

Converte il valore di *espressione* nel corrispondente valore del *tipo* specificato.

Esempio:
```
int somma, n;
float media;
...
media = somma / n;           /* divisione tra interi */
media = (float)somma / n;    /* divisione tra reali  */
```

L'operatore di cast "`(tipo)`" ha precedenza più alta degli operatori binari.

Esempio riassuntivo su input/output per i tipi aritmetici e conversioni di tipo:
file `tipi/tipiarit.c`

Formattazione dell'input/output

Tutto l'input/output è eseguito attraverso gli stream.

- uno **stream** è una sequenza di caratteri organizzata in righe (terminate da "\n")
- ANSI C: righe di almeno 254 caratteri
- al momento dell'esecuzione, al programma vengono connessi automaticamente 3 stream:
  - standard input: di solito tastiera
  - standard output: di solito schermo
  - standard error: di solito schermo

  Tipicamente il sistema operativo consente di ridirigere gli stream standard, ad esempio su un file.

Formattazione dell'output con `printf`

Specificatori di formato:

- interi: `%d`, `%o`, `%u`, `%x`, `%X`
  per `short`: si antepone `h`
  per `long`: si antepone `l` (minuscola)
- reali: `%e`, `%E`, `%f`, `%g`, `%G`
  per `double`: non si antepone nulla
  per `long double`: si antepone `L`
- caratteri: `%c`
- stringhe: `%s` (le vedremo più avanti)
- puntatori: `%p`

Flag: messi subito dopo il "`%`"

- "`-`": allinea a sinistra
- altri flag: vedi libro

Sequenze di escape: `\%`, `\'`, `\"`, `\?`, `\\`, `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v`

Formattazione dell'input con `scanf`

Specificatori di formato: come per l'output, tranne che per i reali

- `double`: si antepone `l`
- `long double`: si antepone `L`

Gruppo di scansione: `%[`*`gruppo-caratteri`*`]`

- utilizzato per la lettura di una sequenza di caratteri
- argomento corrispondente: vettore di caratteri (puntatore a `char`)
- vengono letti solo i caratteri compresi nel gruppo
- l'input si ferma al primo carattere fuori dal gruppo

Esempio:

```
char v[10];  int i;
scanf("%[aeiou]%d", v, &i);
printf("v = %s\n", v);
printf("i = %d\n", i);
```

```
# => aauei245^
v = aauei
i = 245
```

- un gruppo di scansione può contenere un intervallo
  Esempio: `%[0-9]` ... sequenza di cifre

- un gruppo di scansione può essere invertito: `%[^...]`

Esempio:

```
scanf("%[^0-9] %d", v, &i);
printf("v = %s\n", v);
printf("i = %d\n", i);
```

```
# => aa&;uei245
v = aa&;uei
i = 245
```

Soppressione dell'input: mettendo "`*`" subito dopo "`%`"
Non ci deve essere un argomento corrispondente allo specificatore di formato.

Esempio: Lettura di una data in formato `gg/mm/aaaa` oppure `gg-mm-aaaa`.

```
int g, m, a;
scanf("%d%*c%d%*c%d", &g, &m, &a);
```

Esempio: Salta tutti i caratteri e poi salta "`\n`".

```
scanf("%*[^\n]");
getchar();
```