Programmazione 2

Lezione 2 | 26 Febbraio

Università degli Studi di Modena e Reggio Emilia — Unimore

Nota

Risorse: Libro "Programmare in C" — Cap. 5. Esempi pratici in `26-02/` su Moodle: `01_sizes_ranges.c` ... `08_const_and_lvalue.c`.

1. Variabili in C

Il C è un linguaggio **fortemente tipizzato**: ogni variabile deve essere dichiarata con un tipo prima di poter essere usata. Tre concetti distinti da non confondere:

Concetto	Significato
Dichiarazione	Introduce un nome e un tipo (può non allocare memoria)
Definizione	Alloca effettivamente la memoria per l'oggetto
Inizializzazione	Assegna un valore iniziale

```c
extern int g;          // dichiarazione pura (no memoria)
int g = 0;             // dichiarazione + definizione + inizializzazione

int main(void) {
    int x;             // definizione (stack), NON inizializzata -> spazzatura
    int y = 3;         // definizione + inizializzazione
    return 0;
}
```

Attenzione

Variabili non inizializzate:

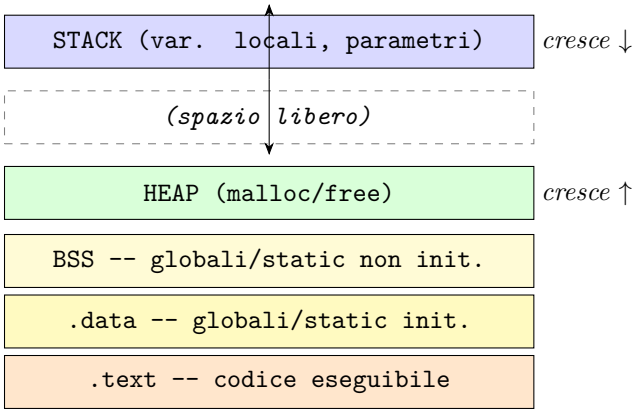
- **Globali / `static`**: inizializzate a zero di default.
- **Locali**: valore **indeterminato** (spazzatura). Fonte infinita di bug. Inizializzarle *sempre* prima di usarle.

2. Scope di una variabile

Lo **scope** dice *dove* un nome è visibile nel codice.

Tipo	Regola
Locale	Visibile solo dentro la funzione / blocco `{}` dove è dichiarata
Globale	Dichiarata fuori da tutte le funzioni → visibile ovunque, per tutta l'esecuzione
Parametri	Trattati come variabili locali alla funzione
`static`	Scope *locale*, ma lifetime *globale* (mantiene il valore tra le chiamate)
`extern`	Accede a variabili globali definite in altri file

3. Dove vive una variabile?



Nota

Lo **stack** cresce verso indirizzi bassi ad ogni chiamata di funzione e si ritira quando la funzione ritorna. L'**heap** è gestito manualmente con `malloc`/`free` — lo vediamo più avanti. BSS e .data sono la casa delle variabili globali: vivono per tutta l'esecuzione del programma.

Attenzione

In C puoi accedere esplicitamente agli indirizzi di memoria tramite i **puntatori**. Molto potente, ma prono ai bug: buffer overflow, dangling pointer, memory leak... Il compilatore non può aggiungere padding in modo predicibile, quindi non dare per scontato che due variabili dichiarate in sequenza siano contigue in memoria.

4. L'astrazione dei tipi di dato

Il **tipo** di una variabile definisce:

1. La **modalità di rappresentazione** in memoria (come vengono interpretati i bit).
2. La **quantità di memoria** da allocare.

4.1. Tipi fondamentali

Con segno	Senza segno
`char`	`unsigned char`
`short`	`unsigned short`
`int`	`unsigned int`
`long`	`unsigned long`
`long long`	`unsigned long long`
`float`	—
`double`	—
`void*`	— *(puntatore non tipizzato)*

4.2. Data models: le dimensioni non sono universali

Attenzione

Non dare per scontato che `long` valga sempre 4 byte — dipende dall'architettura!

Model	int	long	pointer	Dove
ILP32	32 bit	32 bit	32 bit	sistemi 32-bit
LP64	32 bit	64 bit	64 bit	Linux / macOS 64-bit
LLP64	32 bit	32 bit	64 bit	Windows 64-bit

Sigla mnemonica: I = int, L = long, P = pointer, LL = long long.

Il casino è esploso con il passaggio a 64 bit. Windows ha mantenuto LLP64 per compatibilità con il software 32-bit; Linux e macOS hanno adottato LP64. Risultato: `long` vale 4 byte su Windows e 8 byte su Linux/macOS.

4.3. Soluzioni per la portabilità

Soluzione 1 — `<stdint.h>`: tipi a dimensione fissa

```c
#include <stdint.h>
int8_t  a;   // signed  8 bit
int32_t b;   // signed 32 bit
uint64_t c;  // unsigned 64 bit
// garantiti identici su ogni piattaforma
```

Soluzione 2 — `sizeof()`: controlla a runtime

```c
printf("int       = %zu byte\n", sizeof(int));
printf("long      = %zu byte\n", sizeof(long));
printf("long long = %zu byte\n", sizeof(long long));
// %zu e' il formato corretto per size_t (non %lu)
```

Vedi `01_sizes_ranges.c` e `05_data_models.c`.

5. Signed e Unsigned

Con b bit, i range rappresentabili sono:

$$\text{signed:} \quad \left[-2^{b-1},\ +2^{b-1}-1\right]$$

$$\text{unsigned:} \quad \left[0,\ 2^{b}-1\right]$$

Se sai che la variabile non andrà mai in negativo, usa `unsigned`: sfrutti l'intero range positivo senza sprecare il bit di segno.

5.1. Range — `<limits.h>` e `<float.h>`

```c
#include <limits.h>   // CHAR_MIN, CHAR_MAX, INT_MIN, INT_MAX,
    UINT_MAX ...
#include <float.h>    // FLT_MIN, FLT_MAX, DBL_MIN, DBL_MAX ...
```

5.2. Overflow

```c
unsigned u = 0u;
u = u - 1u;        // wrap-around definito -> UINT_MAX (ok, ma non
    farlo senza motivo)

int x = INT_MAX;
x = x + 1;         // overflow signed -> UNDEFINED BEHAVIOR
```

Pericolo

L'overflow su `signed` è **Undefined Behavior**: il compilatore può fare letteralmente qualsiasi cosa, incluso ottimizzare via il tuo controllo di guardia. Non è mai tollerabile.

Vedi `03_signed_unsigned_mix.c`.

6. Costanti numeriche (numeric literals)

Tipo	Nota	Esempio
`int`	default intero	`42`
`unsigned int`	suffisso `u/U`	`42u`
`long`	suffisso `l/L`	`42l`
`long long`	suffisso `ll/LL`	`42ll`
`float`	suffisso `f/F`	`42.0f`
`double`	**default virgola mobile!**	`42.0`

Attenzione

`42.0` senza suffisso è un `double` (8 byte). Se volevi un `float` (4 byte) e non metti la `f`, stai usando il doppio della memoria nelle espressioni — e le performance peggiorano silenziosamente.

6.0.1. Notazioni per interi

```c
int a;
a = 11;      // decimale
a = 013;     // ottale    (prefisso 0)
a = 0b1011; // binario   (prefisso 0b)
a = 0xB;     // esadecimale (prefisso 0x)
// tutte e quattro valgono 11
```

6.0.2. `char` sono interi

```c
char value;
value = 'a';     // carattere ASCII
value = 0x61;    // stessa cosa in hex
value = 97;      // stessa cosa in decimale
```

Vedi `02_bases.c`.

7. `printf` e i format specifier

`printf` viene da `<stdio.h>`. Il placeholder deve corrispondere *esattamente* al tipo della variabile.

Attenzione

Compilare sempre con `-Wformat`: il compilatore non segnala sempre errori di mismatch tra formato e tipo, ma con questo flag li vediamo tutti.

Specifier	Tipo	Azione
`%d`, `%i`	`signed int`	Decimale con segno
`%u`	`unsigned int`	Decimale senza segno
`%o`	`unsigned int`	Ottale
`%x`, `%X`	`unsigned int`	Esadecimale (min/MAI)
`%f`, `%F`	`double`	Virgola mobile
`%s`	`char*`	Stringa (fino a `\0`)
`%c`	`int`	Carattere
`%zu`	`size_t`	Risultato di `sizeof`
`%ld`	`long`	Long intero
`%lld`	`long long`	Long long intero

Formato con precisione: `[<ampiezza>][.precisione][modificatore]tipo`
Esempio: `%.17f` stampa un double con 17 cifre decimali.

Vedi `06_printf_traps.c`.

8. Rappresentazione interna: ripasso architetture

8.1. Complemento a 2 (Two's Complement)

Metodo standard per i numeri interi con segno:

1. I numeri positivi (e zero) si rappresentano normalmente in binario.
2. Per ottenere il negativo corrispondente: **inverti tutti i bit**, poi **somma 1**.

$$1 \rightarrow \texttt{00000001} \xrightarrow{\text{NOT}} \texttt{11111110} \xrightarrow{+1} \texttt{11111111} = -1$$

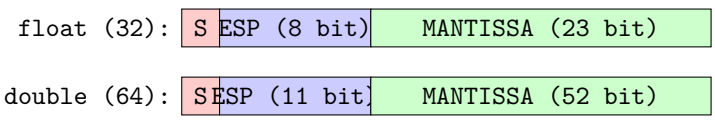
Perché è utile? Basta un solo circuito sommatore per fare addizioni e sottrazioni su qualsiasi numero.

8.1.1. Il trabocchetto signed/unsigned mix

```c
int i = -1;
unsigned u = 1;
printf("%d\n", i < u);  // Output: 0 (FALSO, non VERO!)
// Il -1 viene "promosso" a unsigned -> diventa 4294967295
// 4294967295 < 1 e' falso -> stampato 0
```

Vedi `03_signed_unsigned_mix.c`. Flag consigliati: `-Wall -Wextra -Wconversion -Wsign-conversion`.

8.2. IEEE-754: floating point



8.2.1. Il problema della precisione: `0.1 + 0.2 != 0.3`

```c
#include <math.h>
double z = 0.1 + 0.2;
printf("equal?  %d\n", z == 0.3);               // 0 (FALSO)
printf("close?  %d\n", fabs(z-0.3) < 1e-12);  // 1 (vero)
printf("z    = %.17f\n", z);   // 0.30000000000000004
printf("0.3  = %.17f\n", 0.3); // 0.29999999999999999
```

Pericolo

Regola: non usare mai `==` tra `float`/`double`. Usare sempre un delta di tolleranza con `fabs()`.

Vedi `07_float_precision.c`.

8.3. Endianness

- **Big endian**: il byte più significativo è all'indirizzo più basso (come scriviamo i numeri noi).
- **Little endian**: il byte più significativo è all'indirizzo più alto. ← usato dai processori x86/x64 moderni.

9. Conversioni di tipo

9.1. Implicite

Avvengono automaticamente in tre situazioni:

1. **Assegnazione/inizializzazione**: il valore viene convertito al tipo della destinazione.
2. **Chiamate di funzione / return**: argomenti e valore di ritorno vengono convertiti.
3. **Espressioni aritmetiche**: si applicano le *usual arithmetic conversions* (prossime lezioni).

```c
// Esempio classico di errore silenzioso:
int a = -10;
printf("Valore di a: %u\n", a); // Output: 4294967286 (comportamento
    indefinito)
```

9.2. Esplicite (cast)

```c
long long n = ...;

if (n > INT_MAX) {
    /* gestisci l'errore */
}
int m = (int) n;  // cast esplicito -- attenzione all'overflow se n e
    ' fuori range
```

Vedi `04_casts_conversions.c`.

10. La keyword const

```c
const int a = 3;
a = 4;  // ERRORE di compilazione
```

Nota

`const` dice al compilatore che la variabile non può essere riassegnata. Occupa comunque la stessa memoria del tipo corrispondente (stack se locale, .data se globale). Utile per proteggere valori che non devono cambiare e per rendere esplicite le intenzioni.

Vedi `08_const_and_lvalue.c`.

11. Checklist pratica

Da portare a casa

- Usare flag aggressivi: `-Wall -Wextra -Wconversion -Wsign-conversion -Wformat`
- Scegliere i tipi in base al **dominio** (range) e alla **portabilità** (considerare `<stdint.h>`)
- **Non mischiare signed/unsigned** senza pensarci — è una sorgente silenziosa di bug
- Per `float`/`double`: **mai `==`**, usare epsilon + `fabs()`
- Stampare con precisione (`%.17f`) quando serve capire cosa sta succedendo davvero

Output di riferimento — `01_sizes_ranges.c`

```
== sizeof (byte) ==
char      : 1
short     : 2
int       : 4
long      : 8
long long : 8
float     : 4
double    : 8

== integer ranges (from limits.h) ==
INT_MIN  = -2147483648
INT_MAX  =  2147483647
UINT_MAX =  4294967295

== float ranges (from float.h) ==
FLT_MIN = 1.175494e-38
FLT_MAX = 3.402823e+38
DBL_MIN = 2.225074e-308
DBL_MAX = 1.797693e+308
```