



Programmazione 1 (Canale O-Z)

Lezione 3: La Cassetta degli Attrezzi del C

Docente: Prof. Antonino Furnari

Email: `antonino.furnari@unict.it`

Sito Web: antoninofurnari.github.io/prog1-2627



Riepilogo Rapido: Cosa Sappiamo Già

- Un tipo di dato definisce **quanta memoria** allocare e **come interpretare i bit**.
- Conosciamo già:
 - `int`: 4 byte, numeri interi.
 - `double`: 8 byte, numeri reali in doppia precisione.
- Il C offre però una **famiglia più ampia** di tipi interi.

? Se un `int` occupa 32 bit, quanti numeri diversi può rappresentare? Possiamo salvarci 10 miliardi?





Dimensione dei Tipi: Dipendenza dalla Macchina

- In C, la dimensione esatta in byte dei tipi **non è fissa o universale**.
- Lo standard definisce solo requisiti minimi e relazioni d'ordine:
`sizeof(short) ≤ sizeof(int) ≤ sizeof(long)`
- La dimensione effettiva dipende dall'architettura della CPU (16/32/64 bit) e dal compilatore (sistema operativo e convenzioni di ABI).
- Solo il tipo `char` è **garantito occupare sempre 1 byte** (8 bit).

I Modificatori degli Interi

Adattano il tipo alle nostre esigenze senza sprecare memoria:

- `short int` (o `short`): tipicamente **2 byte**, da -32.768 a +32.767. Per piccoli numeri.
- `int`: tipicamente **4 byte**, da circa -2 miliardi a +2 miliardi.
- `long int` (o `long`): tipicamente **8 byte** sui sistemi a 64 bit, per numeri molto grandi.
- `unsigned int` / `unsigned long`: rimuovono il segno ➡ solo valori positivi, con range positivo **raddoppiato**.



Il Tipo `char` sotto la Maschera

- Un `char` memorizza caratteri (es. `'a'`, `'?'`), ma sotto il cofano è **un intero a 8 bit** (1 byte).
- Valori da -128 a $+127$ (con segno) o da 0 a 255 (`unsigned char`).
- La corrispondenza numero $\leftrightarrow$ simbolo segue la **tabella ASCII**: `'A'` in RAM è l'intero 65 (binario `01000001`).

```
#include <stdio.h>
int main(void) {
    char c = 'A';
    printf("Visualizzato come carattere: %c\n", c);
    printf("Visualizzato come intero:    %d\n", c);

    // Possiamo fare operazioni matematiche sui caratteri!
    c = c + 1;
    printf("Nuovo carattere:           %c\n", c);
    return 0;
}
```



char : Esecuzione

```
gcc -Wall char.c -o char && ./char
```

```
Visualizzato come carattere: A  
Visualizzato come intero:    65  
Nuovo carattere:            B
```

- Stessi bit, due letture diverse: `%c` li mostra come simbolo, `%d` come numero.
- `c + 1` vale `66`, che in ASCII è `'B'`.

? Quale numero corrisponde al carattere `'a'` minuscolo? Dove lo cerchiamo?

Consultare la Tabella ASCII

- Non memorizzare i codici ASCII! Consultali all'occorrenza.
- Nel terminale Unix/macOS:

```
man ascii
```

- Oppure online su: devdocs.io/c/language/ascii

 **Live:** aprire `man ascii` e cercare `A` (65) e `a` (97).

Tipi Interi e Reali (Macchina a 64 bit)

Tipo	Byte	Range dei Valori	Specificatore
char	1	da -128 a $+127$	%c / %d
unsigned char	1	da 0 a 255	%d / %u
short	2	da -32.768 a $+32.767$	%d
unsigned short	2	da 0 a 65.535	%u
int	4	da $-2.147.483.648$ a $+2.147.483.647$	%d
unsigned int	4	da 0 a 4.294.967.295	%u
long	8	da $\approx -9.22 \times 10^{18}$ a $\approx +9.22 \times 10^{18}$	%ld
unsigned long	8	da 0 a $\approx 1.84 \times 10^{19}$	%lu
float	4	da $\approx 1.17 \times 10^{-38}$ a $\approx 3.40 \times 10^{38}$	%f / %e / %g
double	8	da $\approx 2.22 \times 10^{-308}$ a $\approx 1.80 \times 10^{308}$	%f / %e / %g

Spazio in RAM: L'operatore `sizeof`

- `sizeof` restituisce, a tempo di compilazione, i byte occupati da un tipo o da una variabile.
- Restituisce un `size_t` (intero senza segno) ➡ si stampa con `%zu`, **non** con `%d`.

```
#include <stdio.h>

int main(void) {
    printf("char:      %zu byte\n", sizeof(char));
    printf("short:     %zu byte\n", sizeof(short));
    printf("int:        %zu byte\n", sizeof(int));
    printf("long:       %zu byte\n", sizeof(long));
    printf("float:      %zu byte\n", sizeof(float));
    printf("double:    %zu byte\n", sizeof(double));
    return 0;
}
```

Output (tipico, macchina a 64 bit):

char:	1 byte
short:	2 byte
int:	4 byte
long:	8 byte
float:	4 byte
double:	8 byte

 **Live:** `gcc -Wall sizeof.c -o sizeof && ./sizeof`. Provare anche `%d` al posto di `%zu`: cosa dice `-Wall`?

Trovare i Limiti Esatti: `<limits.h>` e `<float.h>`

- Non si imparano a memoria i limiti: si usano le **costanti predefinite** della libreria standard.

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

int main(void) {
    printf("INT_MAX   = %d\n", INT_MAX);
    printf("INT_MIN   = %d\n", INT_MIN);
    printf("UINT_MAX  = %u\n", UINT_MAX);
    printf("LONG_MAX   = %ld\n", LONG_MAX);
    printf("DBL_MAX    = %e\n", DBL_MAX);    // il più grande double positivo
    printf("DBL_MIN    = %e\n", DBL_MIN);    // il più piccolo double positivo normalizzato
    return 0;
}
```

Limiti: Esecuzione

```
gcc -Wall limiti.c -o limiti && ./limiti
```

```
INT_MAX   = 2147483647
INT_MIN   = -2147483648
UINT_MAX  = 4294967295
LONG_MAX  = 9223372036854775807
DBL_MAX   = 1.797693e+308
DBL_MIN   = 2.225074e-308
```

- La competenza chiave non è memorizzare questi numeri: è sapere **dove cercarli**.

12 34 Codifica IEEE 754: I Numeri Reali

Perché $0.1 + 0.2 \neq 0.3$

Campo	Bit	Ruolo
Segno	1	Positivo (0) / negativo (1)
Esponente	11	Scala il numero (potenza di 2), codificato con <i>bias</i> 1023
Mantissa	52	Le cifre significative

$$(-1)^s \times 1.\text{mantissa} \times 2^{(e-1023)}$$

- È la notazione scientifica, ma in **base 2**.

1234 Perché Esiste il Bias 1023?

- L'esponente reale può essere **negativo** (numeri molto piccoli) o **positivo** (numeri molto grandi).
- Gli 11 bit dell'esponente rappresentano valori da 0 a 2047. Per rappresentare esponenti negativi, il *bias* 1023 "centra" la gamma:
 - `e_mem = 0` ➡ `e_reale = -1023` (numeri minuscoli)
 - `e_mem = 2047` ➡ `e_reale = +1024` (numeri enormi)
- Così un `double` copre valori da circa 10^{-308} a 10^{308} .

¹²₃₄ Esempio: Rappresentazione di 0.1

0.1 in binario è infinito:

$$0.1 = 0.00011001100110011001100 \dots_2$$

Normalizzato:

$$0.1 = 1.1001100110011 \dots \times 2^{-4}$$

Quindi:

- segno = 0
- esponente reale = -4 (ci siamo "spostati" di 4 posizioni dopo la virgola)
- esponente memorizzato = $-4 + 1023 = \mathbf{1019}$
- mantissa = primi 52 bit dell'espansione infinita

1234 Tabella dei Campi del `double` per `0.1`

Campo	Bit (valore binario)	Significato	Valore decimale
Segno	0	Numero positivo	+
Esponente	01111111011 (1019)	Esponente memorizzato	Esponente reale = 1019 - 1023 = -4
Mantissa	10011001100110011001100110 01100110011001100110011001	Frazione binaria normalizzata	$\approx 1.100110011001100\dots$

- Ricostruzione: $0.1_{\text{IEEE754}} = 1.1001100110011\dots \times 2^{-4}$
- Valore effettivo memorizzato: `0.10000000000000000000555...` ➡ non è esattamente `0.1`.
- Come `1/3` in base 10, `0.1` in base 2 non ha una scrittura finita.

! L'Illusione della Precisione dei Double

```
#include <stdio.h>

int main(void) {
    double a = 0.1 + 0.2;
    printf("%.1f\n", a);    // 1 cifra decimale
    printf("%.17f\n", a);   // 17 cifre decimali
    return 0;
}
```

? Cosa ci aspettiamo di vedere nelle due righe?

0.1 + 0.2 : Esecuzione

```
gcc -Wall precisione.c -o precisione && ./precisione
```

```
0.3  
0.30000000000000004
```

- $0.1 \approx 0.10000000000000000555$ e $0.2 \approx 0.200000000000000001110$
- Somma $\approx 0.300000000000000004441$: `%.1f` ci **inganna**, `%.17f` mostra la verità.

 **Regola d'oro:** non confrontare mai due `double` con `==`: `(0.1 + 0.2) == 0.3` vale `0`! Usare una tolleranza: `fabs(a - b) < 1e-9`.

Formati di Stampa `printf`: La Guida Completa

Formato	Tipo C	Esempio di output
<code>%d</code>	<code>int</code> (con segno)	<code>42</code> , <code>-7</code>
<code>%u</code>	<code>unsigned int</code>	<code>4294967295</code>
<code>%ld</code>	<code>long int</code>	<code>9223372036854775807</code>
<code>%lu</code>	<code>unsigned long</code>	<code>18446744073709551615</code>
<code>%zu</code>	<code>size_t</code> (risultato di <code>sizeof</code>)	<code>8</code>
<code>%f</code>	<code>double</code> (virgola fissa, 6 decimali default)	<code>3.141593</code>
<code>%e</code>	<code>double</code> (notazione scientifica)	<code>3.141593e+00</code>
<code>%g</code>	<code>double</code> (sceglie il formato più compatto)	<code>3.14159</code>
<code>%c</code>	<code>char</code>	<code>A</code>
<code>%s</code>	stringa (<code>char *</code>)	<code>hello</code>
<code>%%</code>	carattere letterale <code>%</code>	<code>%</code>

Controllo della Precisione Decimale

Il modificatore `.<N>` tra `%` e la lettera fissa il numero di cifre decimali:

```
#include <stdio.h>

int main(void) {
    double pi = 3.14159265358979;
    printf("Default:      %f\n", pi);
    printf("2 decimali: %.2f\n", pi);
    printf("0 decimali: %.0f\n", pi);
    printf("Scientifica: %.3e\n", pi);
    return 0;
}
```

Output:

```
Default:      3.141593
2 decimali: 3.14
0 decimali: 3
Scientifica:3.142e+00
```

- `%.17f`: utile per esporre gli errori di arrotondamento IEEE 754.



Ampiezza Minima e Padding

Un numero tra `%` e la lettera fissa l'**ampiezza minima** del campo:

Specificatore	Effetto
<code>%10d</code>	Almeno 10 caratteri, allineato a destra (spazi a sinistra)
<code>%-10d</code>	Allineato a sinistra (spazi a destra)
<code>%010d</code>	Padding con zeri invece di spazi
<code>%10.2f</code>	Almeno 10 caratteri totali, di cui 2 decimali

? Come stampiamo una tabella di prodotti con le colonne allineate?



Padding: Una Tabella Allineata

```
#include <stdio.h>

int main(void) {
    printf("%-20s %8s %10s\n", "Prodotto", "Quantità", "Prezzo");
    printf("%-20s %8d %10.2f\n", "Mela", 12, 0.89);
    printf("%-20s %8d %10.2f\n", "Cioccolato", 3, 2.50);
    printf("%-20s %8d %10.2f\n", "Acqua minerale", 6, 0.45);
    return 0;
}
```

Output:

Prodotto	Quantità	Prezzo
Mela	12	0.89
Cioccolato	3	2.50
Acqua minerale	6	0.45

Non Serve Imparare Tutto a Memoria

- Non è necessario memorizzare tutti i flag di `printf`.
- Documentazione di sistema:

```
man 3 printf
```

- Documentazione interattiva: devdocs.io/c/io/fprintf

 **Live:** modificare la tabella (es. `%08d`, `%-8d`, `%.3f`), ricompilare con `-Wall` e osservare l'effetto; cercare il flag `0` in `man 3 printf`.

Costanti: Letterali Tipizzati

? Se scriviamo `3.14159` cinque volte nel programma e sbagliamo una cifra in un solo punto?

- I **letterali** sono i valori scritti direttamente nel codice. Hanno un **tipo implicito**:

```
75           // int
75u          // unsigned int
75l          // long int
3.14         // double (attenzione: non float!)
3.14f        // float
3.14159L     // long double
```

- Conta quando si passano argomenti a funzioni o si sommano tipi diversi.



const : Variabile Immutabile

```
const double PI = 3.14159265358979;  
const int MAX_STUDENTI = 200;
```

- Il compilatore **vieta** qualunque modifica successiva:

```
const double PI = 3.14;  
PI = 3.1415; // ERRORE: assignment of read-only variable 'PI'
```

- Dichiarazione e inizializzazione **sulla stessa riga**: `const int x;` seguito da `x = 5;` è un errore.
- **Quando usarla**: valori con significato preciso che non devono mai cambiare (aliquote, costanti fisiche, ...).

 **Live**: compilare con `gcc -Wall` il frammento con `PI = 3.1415;` e leggere l'errore.

⚙️ #define : Sostituzione del Preprocessore

```
#define VOTO_MINIMO 18
#define CORSO "Programmazione 1"
```

- Non crea una variabile: il **preprocessore** sostituisce meccanicamente ogni occorrenza del nome con il testo, **prima** che il compilatore veda il codice.
- Per convenzione i nomi delle macro sono in **MAIUSCOLO**.

```
/* demo_define.c – senza nessuna #include per vedere l'output pulito */
#define VOTO_MINIMO 18

int main(void) {
    return VOTO_MINIMO;
}
```

? Che cosa vedrà il compilatore al posto di `VOTO_MINIMO` ?

#define : Guardiamo il Preprocessore

Ci fermiamo dopo la fase di preprocessing con `-E` (come nella Lezione 1):

```
gcc -E demo_define.c -o demo_define.i
cat demo_define.i
```

Output (semplificato):

```
# 1 "demo_define.c"
# 1 "<built-in>" 1
...

int main(void) {
    return 18;
}
```

- `VOTO_MINIMO` è **scomparso**: il compilatore vedrà solo `18`.
- Nessuna cella di memoria e **nessun tipo** (niente controlli del compilatore): per le costanti numeriche semplici si preferisce `const`.

Costanti Predefinite dalla Libreria Standard

La libreria standard offre molte costanti pronte, definite tramite `#define`:

Costante	Header	Significato
<code>M_PI</code>	<code><math.h></code>	π con la massima precisione da <code>double</code>
<code>M_E</code>	<code><math.h></code>	il numero di Eulero e
<code>INT_MAX</code> , <code>INT_MIN</code>	<code><limits.h></code>	limiti degli interi
<code>DBL_MAX</code> , <code>DBL_EPSILON</code>	<code><float.h></code>	limiti e precisione dei double

Conversione Implicita (Promozione)

- Mescolando tipi diversi, il C promuove automaticamente al tipo **più capiente**:
 - `int + double` ➡ l'`int` viene convertito in `double` prima di sommare.
- Assegnando un `double` a un `int`, la parte decimale viene **troncata** (non arrotondata!).

```
double x = 7.9;
int y = x;           // y vale 7, non 8! Troncamento silenzioso.
printf("%d\n", y);  // stampa: 7
```

- Il troncamento silenzioso è una fonte comune di bug: **non ignorate i warning** del compilatore.

 **Live:** compilare con `gcc -Wall -Wextra` e poi aggiungendo `-Wconversion`: quale dei due comandi segnala la conversione `double` ➡ `int`?

Cast Esplicito

Per una conversione **deliberata**: il tipo di destinazione tra parentesi prima dell'espressione.

```
int a = 5, b = 2;

double risultato = (double)a / b;    // cast prima della divisione
double sbagliato = (double)(a / b);  // divisione intera, poi cast

printf("%.1f vs %.1f\n", risultato, sbagliato);
```

2.5 vs 2.0

- `(double)a / b`: `a` diventa `5.0` ➡ divisione **reale**.
- `(double)(a / b)`: la divisione **intera** (`2`) è già avvenuta ➡ `2.0`.

? Dove mettiamo le parentesi fa la differenza: perché?

Arrotondare con `round`, `floor`, `ceil`

Il troncamento di `(int)x` non è sempre ciò che vogliamo:

Funzione	Semantica	Esempi
<code>round(x)</code>	arrotondamento tradizionale	2.4 ➡ 2.0, 2.5 ➡ 3.0
<code>floor(x)</code>	sempre verso il basso (<i>pavimento</i>)	2.9 ➡ 2.0, -2.1 ➡ -3.0
<code>ceil(x)</code>	sempre verso l'alto (<i>soffitto</i>)	2.1 ➡ 3.0, -2.9 ➡ -2.0

Restituiscono un `double`: per ottenere un `int` si combinano con il cast.

```
#include <math.h>
```

```
int n = (int)round(7.6);    // n = 8
int m = (int)floor(7.6);    // m = 7
int p = (int)ceil(7.2);     // p = 8
```



La Libreria Matematica `<math.h>`

- Trigonometria, potenze e radici richiedono istruzioni macchina specializzate (**FPU**, *Floating Point Unit*): non sono nel nucleo del linguaggio.
- Si importano esplicitamente con `#include <math.h>`.
- **Linux/GCC**: la libreria va collegata con `-lm`:

```
gcc -Wall programma.c -lm -o programma
```

- Su macOS e con Clang `-lm` non è necessario.



<math.h> : Le Funzioni Principali

Funzione	Descrizione	Esempio
<code>sqrt(x)</code>	Radice quadrata	<code>sqrt(9.0)</code> ➡ 3.0
<code>pow(x, y)</code>	x^y	<code>pow(2, 10)</code> ➡ 1024.0
<code>fabs(x)</code>	Valore assoluto per <code>double</code>	<code>fabs(-3.5)</code> ➡ 3.5
<code>sin(x)</code> , <code>cos(x)</code> , <code>tan(x)</code>	Funzioni trigonometriche (in radianti)	<code>sin(M_PI)</code> $\approx$ 0.0
<code>log(x)</code>	Logaritmo naturale ($\ln$)	<code>log(M_E)</code> ➡ 1.0
<code>log10(x)</code>	Logaritmo in base 10	<code>log10(100.0)</code> ➡ 2.0
<code>round(x)</code> , <code>floor(x)</code> , <code>ceil(x)</code>	Arrotondamenti	(slide precedente)

- `M_PI` definisce π con la massima precisione del `double`.
- 📖 Documentazione completa: devdocs.io/c/numeric/math

Demo: Moto Uniformemente Accelerato

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

int main(void) {
    double v0, a, t;

    printf("Velocità iniziale (m/s): ");
    scanf("%lf", &v0);
    printf("Accelerazione (m/s²): ");
    scanf("%lf", &a);
    printf("Tempo (s): ");
    scanf("%lf", &t);

    double s = v0 * t + 0.5 * a * t * t; // spazio percorso
    double vf = v0 + a * t;             // velocità finale
    double ec = 0.5 * vf * vf;          // energia cinetica specifica (massa=1 kg)

    printf("\n--- Risultati ---\n");
    printf("Spazio percorso:      %10.3f m\n", s);
    printf("Velocità finale:        %10.3f m/s\n", vf);
    printf("Energia cinetica:         %10.3f J/kg\n", ec);

    return 0;
}
```



Moto Accelerato: Esecuzione

```
gcc -Wall moto.c -lm -o moto
./moto
```

```
Velocità iniziale (m/s): 10
Accelerazione (m/s2): 2
Tempo (s): 3

--- Risultati ---
Spazio percorso:      39.000 m
Velocità finale:      16.000 m/s
Energia cinetica:     128.000 J/kg
```

- Tipi, `scanf` con `%lf`, padding `%10.3f`: i risultati sono **incolonnati**.

 **Live:** eseguire con altri valori e verificare che le colonne restino allineate.

Overflow degli Interi

? `INT_MAX` è circa 2 miliardi. Cosa succede se aggiungiamo 1?

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

int main(void) {
    int x = INT_MAX;
    printf("INT_MAX      = %d\n", x);
    printf("INT_MAX + 1 = %d\n", x + 1);
    return 0;
}
```

Overflow: Esecuzione

```
INT_MAX      = 2147483647
INT_MAX + 1  = -2147483648
```

- Il risultato è **negativo**: siamo tornati al minimo rappresentabile, come un orologio che dopo le 23:59 torna alle 00:00 (*wraparound*).
- Questo fenomeno si chiama **overflow**.

⚠ **Comportamento indefinito**: in C l'overflow degli interi con segno è *Undefined Behavior*. In pratica produce il wraparound, ma lo standard non lo garantisce. Per somme su grandi valori, promuovete prima a `long`: `(long)x + 1`.

★ Reali: i Valori Speciali `inf` e `nan`

I reali non "girano su se stessi": producono **valori speciali** IEEE 754.

```
#include <stdio.h>

int main(void) {
    float troppo_grande = 10e10f * 10e30f;
    printf("%f\n", troppo_grande);
    printf("%f\n", 0.0 / 0.0);
    printf("%f\n", 1.0 / 0.0);
    printf("%f\n", -1.0 / 0.0);
    return 0;
}
```

Output:

```
inf
nan
inf
-inf
```

(su Linux la seconda riga può comparire come
`-nan`)

- `inf`: troppo grande per essere rappresentato. `-inf`: infinito negativo.
- `nan` (*Not A Number*): operazione indefinita (`0.0 / 0.0`, `sqrt(-1.0)`).
- ⚠ Si **propagano**: ogni operazione con `nan` produce `nan`, senza errori visibili.



Regola d'Oro dello Sviluppatore

- Compilate **sempre** attivando tutti gli avvertimenti del compilatore:

```
gcc -Wall -Wextra -o mio_programma sorgente.c
```

- Un programma che produce avvertimenti (warnings) non è un programma corretto o sicuro.
- Collegare sempre le librerie corrette (es. `-lm` per `<math.h>`).
- Utilizzate la documentazione integrata di sistema:

```
man 3 printf
```

✓ Cosa Abbiamo Imparato Oggi

- Le dimensioni dei tipi **dipendono dalla macchina**: si misurano con `sizeof` e si consultano in `<limits.h>` / `<float.h>`.
- `char` è un **intero a 8 bit**: la tabella ASCII lega numeri e simboli.
- IEEE 754: `0.1 + 0.2` **non** è esattamente `0.3` ➡ mai `=` tra `double`.
- `printf`: formati (`%u`, `%ld`, `%e`, ...), precisione (`%.2f`) e padding (`%-20s`, `%10.2f`).
- Costanti: letterali tipizzati, `const`, `#define`.
- Conversioni: troncamento implicito, cast esplicito, `round` / `floor` / `ceil`.
- `<math.h>` con `-lm`; overflow, `inf`, `nan`; sempre `-Wall -Wextra`.

📖 **Per casa:** esercizi della Lezione 3 sul sito del corso.

Grazie per l'attenzione!

Sito Web del Corso: antoninofurnari.github.io/prog1-2627