



Programmazione 1 (Canale O-Z)

Lezione 2: La Logica dei Numeri e l'Input/Output

Docente: Prof. Antonino Furnari

Email: `antonino.furnari@unict.it`

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



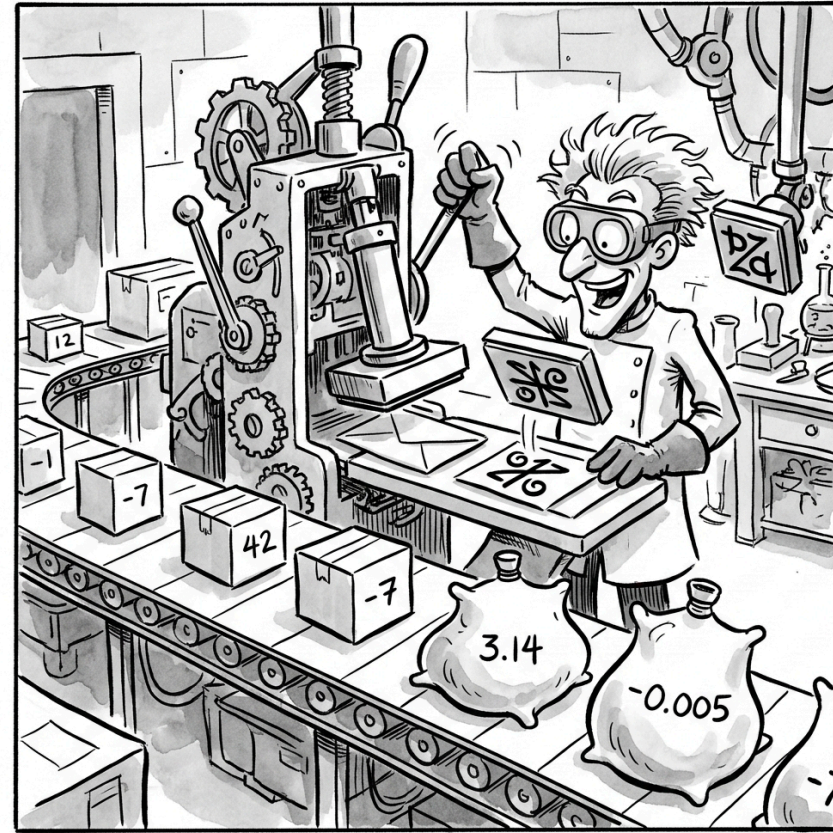
Variabili in RAM: Sotto il Cofano

Che cos'è una variabile?

`int age = 19;` ordina al compilatore di:

- **Riservare** una cella di RAM (nello Stack) di dimensione fissa: **4 byte** per un `int` sui sistemi a 64 bit.
- **Associare** il nome `age` all'indirizzo di partenza della cella.
- **Scrivere** nella cella il valore in binario:

```
00000000 00000000 00000000 00010011
```



I Nomi Sono un'Illusione

- I nomi delle variabili servono **solo a noi umani**.
- In Assembly e codice macchina i nomi (`age`, `raggio`, ...) **scompaiono**.
- Al loro posto: indirizzi di memoria o scostamenti da registri di base (es. `[rbp-4]`).

? Ricordate l'Assembly della Lezione 1? Dove erano finite `x` e `y`?

Il Concetto di Tipo di Dato

Dimensione e Regole di Interpretazione

- Il tipo dice al compilatore due cose:
 - **Dimensione**: quanti byte riservare.
 - **Codifica**: la regola con cui interpretare i bit.
- Esempio: i bit dell'intero `65` sono **identici** a quelli del carattere `'A'` in ASCII: solo il tipo dice come leggerli.
- **Perché dichiarare i tipi esplicitamente?** Il compilatore genera subito le istruzioni macchina corrette ed efficienti: la somma tra interi (`add`) usa circuiti della CPU diversi dalla somma in virgola mobile.

¹²₃₄ Tipi Fondamentali: `int` vs `double`

`int`: interi con segno

- Esempi: `0`, `42`, `-1053`
- Dimensione: **4 byte** (32 bit)
- Intervallo: da circa -2 miliardi a +2 miliardi

`double`: numeri reali

- Esempi: `3.14`, `-0.005`, `2.0`
- Dimensione: **8 byte** (64 bit), *double precision*
- Codifica: standard **IEEE 754** (virgola mobile)
- È il tipo di default delle costanti decimali nel codice



Dichiarazione e Inizializzazione

Dichiarazione con inizializzazione (stessa riga):

```
int age = 19; // Alloca la memoria e vi scrive subito 19
```

Dichiarazione disgiunta dall'assegnamento:

```
int score; // Dichiarazione: spazio riservato, ma il valore è indeterminato  
score = 100; // Assegnamento successivo: viene scritto 100
```

? Cosa contiene `score` tra le due righe?

Pericolo: Variabili Non Inizializzate!

- Dichiarare una variabile riserva la cella ma **non azzerà i bit**.
- Leggerla prima di scriverci **non dà errore di compilazione**, ma restituisce un **valore spazzatura** (*garbage value*): i residui di operazioni precedenti in quell'area di RAM.
- **Inizializzate sempre le vostre variabili!**

```
int score;  
printf("%d\n", score); // ERRORE LOGICO: stampa dati spazzatura!
```

 **Live:** compilare con `gcc -Wall garbage.c -o garbage`: `-Wall` ci avvisa con warning: `'score' is used uninitialized`. Eseguire con `./garbage` e guardare il valore stampato.

Lvalue vs Rvalue: Identità di Assegnamento

Capire gli errori del compilatore

- **Lvalue (Left-value)**: una cella di memoria persistente e **scrivibile**. Sta a **sinistra** di `=`. Esempio: il nome di una variabile (`x`).
- **Rvalue (Right-value)**: un valore temporaneo, costante o calcolato, **senza** una cella persistente. Sta solo a **destra** di `=`. Esempi: `5`, `x + 3`.

```
5 = x;           // ERRORE: 5 è un rvalue, non potete assegnargli una cella di memoria!  
x + y = 10;      // ERRORE: l'espressione a sinistra non è un lvalue scrivibile.
```

 **Live:** compilare `5 = x;` e leggere l'errore: `error: lvalue required as left operand of assignment.`

Regole per i Nomi delle Variabili

Identificatori Validi in C

Caratteri ammessi: lettere, cifre, `_`

Primo carattere: lettera o `_` (mai una cifra)

Nome	Valido?
<code>var_123</code>	✓
<code>_var2</code>	✓
<code>2var</code>	✗ (inizia con cifra)
<code>var-2</code>	✗ (- non consentito)

Parole chiave riservate (non usabili come nomi):

`int` `double` `float` `char` `void`
`if` `else` `for` `while` `do`
`return` `struct` `typedef` `const` ...

⚠ C è case sensitive

`var`, `Var` e `VAR` sono tre identificatori
distinti.

Output: La Funzione `printf`

Modelli di Stampa (Format String)

- `printf` è una **stampante di modelli**: converte la rappresentazione binaria in RAM (il numero `5`) nel carattere testuale corrispondente (il codice ASCII di `'5'`).
- Riceve una **Stringa di Formato** con segnaposto, seguita dagli argomenti che li riempiono **in ordine**:

```
printf("L'utente %s ha ottenuto %d punti.\n", "Alice", 150);
```

```
L'utente Alice ha ottenuto 150 punti.
```

- `%s` riceve `"Alice"` (stringa), `%d` riceve `150` (intero).

Specificatori di Formato Essenziali

Specificatore	Tipo	Esempio di stampa
<code>%d</code>	<code>int</code> in base 10 con segno	<code>150</code>
<code>%f</code>	<code>double</code> / <code>float</code> in virgola fissa	<code>3.141593</code> (6 decimali di default)

 **Live:** nessuno ricorda tutti i formati a memoria: aprire `man 3 printf` al terminale (la sezione 3 del manuale è quella della libreria standard C).



Input da Tastiera: `scanf`

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

Tre Regole d'Oro per `scanf`:

1. **Formato corretto:** `%d` per interi, `%lf` (*long float*) **obbligatorio** per i `double` (con `%f` scriverebbe 4 byte anziché 8, corrompendo la memoria).
2. **Indirizzo con `&`:** `scanf` deve scrivere nella variabile, quindi le serve l'indirizzo RAM della cella. Per ora: `&` ci dà accesso **in scrittura** alla variabile.
3. **Consumo del whitespace:** ignora spazi, tab e invii iniziali fino al primo carattere utile.

Esempio Completo: Area del Cerchio

```
#include <stdio.h>

int main(void) {
    double raggio = 0.0;
    double area = 0.0;
    double PI = 3.1415926535;

    printf("Inserisci il raggio del cerchio (numero reale): ");
    scanf("%Lf", &raggio); // Chiede il raggio all'utente

    area = PI * raggio * raggio;

    printf("\n--- Risultati ---\n");
    printf("Raggio inserito: %f cm\n", raggio);
    printf("Area calcolata:  %f cm2\n", area);

    return 0;
}
```

Area del Cerchio: Esecuzione

```
gcc -Wall cerchio.c -o cerchio
./cerchio
```

```
Inserisci il raggio del cerchio (numero reale): 2
```

```
--- Risultati ---
```

```
Raggio inserito: 2.000000 cm
```

```
Area calcolata: 12.566371 cm2
```

- Il valore digitato viene scritto da `scanf` all'indirizzo `&raggio`.
- `%f` stampa **6 cifre decimali** di default.

 **Live:** provare a sostituire `%lf` con `%f` nella `scanf` e ricompilare con `-Wall`: leggere il warning.

+ Espressioni e Operatori

- **Unari**: un solo operando (es. `-x`, `!x`).
- **Binari**: due operandi (es. `+`, `-`, `*`, `/`, `%`).
- Il compilatore applica regole rigide di **Precedenza** (quale operatore prima) e **Associatività** (in che direzione, a parità di priorità).

Ogni espressione restituisce un valore!

- Anche l'assegnamento: `x = 5` scrive `5` in `x` e restituisce `5`.
- Per questo si può scrivere `y = x = 5`; : prima `x = 5` (vale `5`), poi `y = 5`.

+ Precedenza e Associatività

Priorità	Operatore	Descrizione	Valore di Output Restituito	Associatività
1 (Massima)	+ - !	Unari (Segno, NOT)	Valore invertito o stato logico (0/1)	Destra → Sinistra
2 (Media)	* / %	Molt. / Div. / Mod.	Risultato aritmetico	Sinistra → Destra
3 (Bassa)	+ -	Add. / Sott.	Risultato aritmetico	Sinistra → Destra
4 (Minima)	=	Assegnamento	Il valore scritto nella variabile a sx	Destra → Sinistra

? Quanto valgono `x` e `y` dopo `y = (x = 2 * 3) + 4; ?` ➡ tracciamo passo per passo

Tracciamento di un'Espressione (1/3)

```
int x;  
int y;  
y = (x = 2 * 3) + 4;
```

Passo 1: si risolve la parentesi più interna, `x = 2 * 3`.

Passo 2: la moltiplicazione `2 * 3` restituisce `6`.

```
y = (x = 6) + 4;
```

x	y
?	?

Tracciamento di un'Espressione (2/3)

Passo 3: l'assegnamento `x = 6` scrive `6` in `x` e restituisce `6`.

Passo 4: la parentesi `(x = 2 * 3)` viene sostituita dal suo valore di output.

```
y = 6 + 4;
```

x	y
6	?

Tracciamento di un'Espressione (3/3)

Passo 5: la somma `6 + 4` restituisce `10`.

Passo 6: l'assegnamento finale scrive `10` in `y`.

```
y = 10;
```

x	y
6	10

- 💡 Usate sempre le parentesi `()` per forzare l'ordine desiderato ed evitare espressioni ambigue.



Assegnamenti Composti

Sintassi abbreviata ed efficiente

Per rendere il codice più compatto, C mette a disposizione operatori combinati:

- `x += 5;` equivale a `x = x + 5;`
- `x -= 1;` equivale a `x = x - 1;`
- `x *= 2;` equivale a `x = x * 2;`
- `x /= 3;` equivale a `x = x / 3;`
- `x %= 2;` equivale a `x = x % 2;`

⚡ Divisione Intera vs Divisione Reale

Il Trabocchetto della Divisione

- Il comportamento di `/` dipende **solo dal tipo degli operandi**.
- **Divisione Intera**: entrambi `int` ➡ la parte decimale viene **troncata** (nessun arrotondamento). È il **quoziente**.
 - `5 / 2` ➡ `2`
- **Divisione Reale**: almeno un operando reale ➡ l'altro viene promosso e il calcolo è in virgola mobile.
 - `5.0 / 2` ➡ `5.0 / 2.0` ➡ `2.5`
- **Modulo (%)**: il **resto** della divisione intera, **solo** tra interi.
 - `5 % 2` ➡ `1`. Uso tipico: `n` è pari se `n % 2 == 0`.

Divisione: Proviamolo

```
printf("%d %f %d\n", 5 / 2, 5.0 / 2, 5 % 2);
```

```
2 2.500000 1
```

 **Live:** compilare con `gcc -Wall divisione.c -o divisione && ./divisione`. Poi provare `printf("%f\n", 5 / 2);` e leggere il warning di `-Wall` sul formato.

? Quanto vale `9 / 5` in C? E se ci serve `1.8`?

Come il C Rappresenta la Verità

- Nel C originale **non esiste** un tipo booleano nativo: la verità si rappresenta con gli interi.
 - `0` ➡ Falso
 - **Qualsiasi valore diverso da `0`** (di solito `1`) ➡ Vero
- Gli operatori relazionali confrontano due espressioni e restituiscono `1` (Vero) o `0` (Falso).

? Se `a = 5` e `b = 10`, quanto vale l'espressione `a < b`? ➡ slide successiva

Operatori Relazionali (Confronto)

Operatore	Significato	Esempio	Output (se <code>a=5</code> , <code>b=10</code>)
<code>=</code>	Uguale a	<code>a = b</code>	<code>0</code> (Falso)
<code>≠</code>	Diverso da	<code>a ≠ b</code>	<code>1</code> (Vero)
<code>></code>	Maggiore di	<code>a > b</code>	<code>0</code> (Falso)
<code><</code>	Minore di	<code>a < b</code>	<code>1</code> (Vero)
<code>≥</code>	Maggiore o uguale	<code>b ≥ 10</code>	<code>1</code> (Vero)
<code>≤</code>	Minore o uguale	<code>a ≤ 5</code>	<code>1</code> (Vero)

-  `=` (confronto) da non confondere con `=` (assegnamento)!

Operatori Logici Booleani

AND, OR e NOT in C

Operatore	Nome	Sintassi	Regola di Output
<code>&&</code>	AND (Congiunzione)	<code>A && B</code>	1 solo se sia <code>A</code> che <code>B</code> sono diversi da zero
<code> </code>	OR (Disgiunzione)	<code>A B</code>	1 se almeno uno tra <code>A</code> e <code>B</code> è diverso da zero
<code>!</code>	NOT (Negazione)	<code>!A</code>	1 se <code>A</code> vale zero, 0 se <code>A</code> è diverso da zero

Tabelle di Verità (Truth Tables)

Output logico: 0 = Falso, 1 = Vero

Operando A	Operando B	A && B	A B	!A
0 (Falso)	0 (Falso)	0	0	1
0 (Falso)	1 (Vero)	0	1	1
1 (Vero)	0 (Falso)	0	1	0
1 (Vero)	1 (Vero)	1	1	0

Decisioni Condizionali: `if`

Permette di deviare il flusso lineare del programma in base a una decisione logica.

```
if (condizione) {  
    // Eseguito solo se la condizione restituisce un valore Vero (diverso da 0)  
}
```

? Come usiamo `&&` e `!` dentro un `if`? ➡ esempio completo



if in Azione: Esempio Completo

```
#include <stdio.h>

int main(void) {
    int haPatente = 1; // 1 rappresenta Vero, 0 rappresenta Falso
    int eta = 20;

    // L'operatore logico && combina i due controlli
    if (eta ≥ 18 && haPatente) {
        printf("Autorizzato alla guida dell'autovettura.\n");
    }

    // Il NOT logico inverte la condizione
    int pioggia = 1; // 1 rappresenta Vero (Piove)
    if (!pioggia) {
        printf("Puoi fare una passeggiata all'aperto.\n");
    } else {
        printf("Prendi l'ombrello!\n");
    }

    return 0;
}
```



if in Azione: Esecuzione

```
gcc -Wall patente.c -o patente  
./patente
```

Autorizzato alla guida dell'autovettura.
Prendi l'ombrello!

- `eta ≥ 18` vale `1` e `haPatente` vale `1` ➡ `1 && 1` vale `1`: il primo blocco viene eseguito.
- `pioggia` vale `1` ➡ `!pioggia` vale `0`: si esegue il ramo `else`.

 **Live:** cambiare `eta` in `16` e `pioggia` in `0`, ricompilare e rieseguire: cosa cambia nell'output?

Logica di Cortocircuito (Short-Circuit)

- Il C valuta le espressioni logiche **da sinistra a destra** e si **ferma** appena il risultato è matematicamente certo.
- **AND (A && B)**: se A è Falso (0), il risultato è sicuramente 0 ➡ B non viene valutato.
- **OR (A || B)**: se A è Vero (1), il risultato è sicuramente 1 ➡ B non viene valutato.
- Utilissimo per un **controllo di sicurezza** prima di un'operazione potenzialmente fatale.

Cortocircuito: AND (&&)

Prevenire operazioni illegali a runtime

```
int divisore = 0;
int dividendo = 10;

// Grazie al cortocircuito, poiché "divisore ≠ 0" è Falso (0),
// la divisione per zero non viene eseguita e il programma non crasha.
if (divisore ≠ 0 && (dividendo / divisore) > 2) {
    printf("Condizione superata in sicurezza!\n");
}
```

Cortocircuito: OR (||)

Bypass e scorciatoie di sicurezza

```
int bypassaControlli = 1; // 1 rappresenta Vero
int divisore = 0;
int dividendo = 10;

// Poiché "bypassaControlli" è Vero (1), la condizione è già soddisfatta:
// la seconda parte non viene valutata, niente divisione per zero!
if (bypassaControlli || (dividendo / divisore) > 2) {
    printf("Accesso consentito (controlli bypassati o superati).\n");
}
```

 **Live:** invertire l'ordine, `if ((dividendo / divisore) > 2 || bypassaControlli)`, ricompilare ed eseguire: il programma si arresta con Floating point exception.

Cosa Abbiamo Imparato Oggi

- Una variabile è un **nome simbolico** per una cella di RAM; il **tipo** ne fissa dimensione e codifica (`int` 4 byte, `double` 8 byte).
- **Inizializzate sempre** le variabili: altrimenti contengono valori spazzatura.
- `printf` con `%d / %f`; `scanf` con `%d / %lf` e l'operatore `&`.
- Ogni espressione **restituisce un valore**, anche l'assegnamento.
- `5 / 2` vale `2`, `5.0 / 2` vale `2.5`, `5 % 2` vale `1`.
- In C `0` è Falso, il resto è Vero; `&&` e `||` sono **cortocircuitati**.

 **Per casa:** esercizi della Lezione 2 sul sito del corso.

Grazie per l'attenzione!

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