

LEZIONE 3

La Cassetta degli Attrezzi del C



Dispense di Programmazione 1 – Canale O-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania

Lezione 1

La Cassetta degli Attrezzi del C

Nelle lezioni precedenti abbiamo costruito le fondamenta del C: le variabili `int` e `double`, i formati base di `printf` e `scanf`, gli operatori aritmetici e la logica condizionale con `if`. In questa lezione completiamo la nostra **cassetta degli attrezzi** fondamentale: impareremo a scegliere il tipo più adatto per ogni situazione grazie ai modificatori (`unsigned`, `short`, `long`), a formattare l'output con precisione chirurgica grazie al padding di `printf`, a nominare i valori fissi con `const` e `\{\}\#define`, a convertire dati tra tipi diversi in modo consapevole e ad accedere alle funzioni matematiche avanzate della libreria `<math.h>`. Chiuderemo con una demo sui limiti dei tipi numerici: overflow, underflow, `inf` e `NAN`.

1.1 Il Sistema dei Tipi: Modificatori e Limiti

1.1.1 Riepilogo Rapido

Ricordate che un tipo di dato definisce due cose: **quanta memoria** viene allocata e **come vengono interpretati i bit** al suo interno. Abbiamo già incontrato `int` (4 byte, numeri interi) e `double` (8 byte, numeri reali in doppia precisione). Ma il C mette a disposizione una famiglia più ampia di tipi interi.

1.1.2 Modificatori di tipo, il tipo `char` (ASCII) e la dipendenza dalla macchina

- **La dipendenza dalla macchina (Machine Dependency)** Una delle caratteristiche più importanti del C è che **la dimensione in byte dei tipi di dato non è fissa o universale**. Lo standard C definisce solo dei requisiti minimi e delle relazioni d'ordine (es. `sizeof(short) <= sizeof(int) <= sizeof(long)`). La dimensione effettiva dipende dall'architettura del processore (16, 32 o 64 bit) e dal compilatore (il sistema operativo e le sue convenzioni di ABI). Il tipo `char` è l'unico garantito occupare **sempre 1 byte (8 bit)** in qualsiasi sistema compatibile.
- **I modificatori degli interi** Permettono di adattare il tipo alle proprie esigenze senza sprecare memoria:
 - `short int` (o `short`): tipicamente 2 byte, valori da -32.768 a +32.767. Utile per piccoli numeri.
 - `int`: tipicamente 4 byte, valori da -2 miliardi a +2 miliardi.

- `long int` (o `long`): tipicamente 8 byte su sistemi moderni a 64 bit, per numeri molto grandi.
- `unsigned int`/`unsigned long`: rimuovono il segno, consentendo di rappresentare solo valori positivi e raddoppiando il range positivo.
- **Il tipo `char` è un intero piccolo sotto la maschera!** Spesso si pensa al tipo `char` come a qualcosa di radicalmente diverso perché serve a memorizzare lettere o simboli (es. `'a'`, `'?'`). **Sotto il cofano del calcolatore, un `char` è semplicemente un numero intero a 8 bit (1 byte).** Può memorizzare valori numerici compresi tra -128 e $+127$ (se con segno) o da 0 a 255 (se `unsigned char`).

L'associazione tra il numero binario memorizzato in RAM e il carattere visualizzato a schermo è governata da una convenzione di codifica: la **tabella ASCII** (American Standard Code for Information Interchange). Ad esempio, memorizzare il carattere `'A'` significa scrivere in RAM il valore intero 65 (pari al binario `01000001`).

L'equivalenza in pratica:

```
1 #include <stdio.h>
2 int main(void) {
3     char c = 'A';
4     printf("Visualizzato come carattere: %c\n", c); // Stampa: A
5     printf("Visualizzato come intero:      %d\n", c); // Stampa: 65
6
7     // Possiamo fare operazioni matematiche sui caratteri!
8     c = c + 1;
9     printf("Nuovo carattere:              %c\n", c); // Stampa: B (66
10    ↪ in ASCII)
11    return 0;
12 }
```

- **La Cultura dello Sviluppatore: Consultare la tabella ASCII** Non bisogna imparare i codici a memoria. Su qualsiasi sistema Unix/macOS basta digitare nel terminale:

```
1 man ascii
```

In alternativa, si può consultare la risorsa web di riferimento su devdocs.io/c/language/ascii.

- **Tabella dei range tipici e dipendenza dall'implementazione** Ecco i range tipici su una macchina a 64 bit moderna:

Tabella

Tipo	Byte (tipici)	Intervallo di Valori	Specificatore printf
<code>char</code>	1	da -128 a $+127$	<code>\{%c</code> (carattere) o <code>\{%d</code> (intero)
<code>unsigned char</code>	1	da 0 a 255	<code>\{%d</code> o <code>\{%u</code>

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 $\approx$ -9.22×10^{18} a $\approx$ $+9.22 \times 10^{18}$	\{\}%ld
unsigned long	8	da 0 a $\approx$ 1.84×10^{19}	\{\}%lu
float	4	da $\approx$ 1.17×10^{-38} a $\approx$ 3.40×10^{38}	\{\}%f (o \{\}%e / \{\}%g)
double	8	da $\approx 2.22 \times$ 10^{-308} a $\approx$ 1.80×10^{308}	\{\}%f (o \{\}%e / \{\}%g)

1.1.3 Scoprire le Dimensioni con sizeof

L'operatore `sizeof` restituisce a tempo di compilazione il numero di byte occupato da un tipo o da una variabile. È molto utile per verificare il comportamento del compilatore sulla propria macchina:

```

1  #include <stdio.h>
2
3  int main(void) {
4      printf("char:      %zu byte\n", sizeof(char));
5      printf("short:     %zu byte\n", sizeof(short));
6      printf("int:        %zu byte\n", sizeof(int));
7      printf("long:       %zu byte\n", sizeof(long));
8      printf("float:      %zu byte\n", sizeof(float));
9      printf("double:    %zu byte\n", sizeof(double));
10     return 0;
11 }
```

Nota

Il formato `\{%zu` per `sizeof` restituisce un valore di tipo `size_t` (un intero senza segno, sui sistemi a 64 bit grande quanto un `unsigned long`). Lo specificatore corretto per stamparlo è `\{%zu`, non `\{%d`.

1.1.4 Accedere ai Limiti Esatti: `<limits.h>` e `<float.h>`

Non è necessario ricordare i valori limite a memoria. La libreria standard mette a disposizione costanti predefinite che il compilatore calcola automaticamente per la piattaforma corrente:

```

1  #include <stdio.h>
2  #include <limits.h>
3  #include <float.h>
4
5  int main(void) {
6      printf("INT_MAX = %d\n", INT_MAX);    // +2.147.483.647
7      printf("INT_MIN = %d\n", INT_MIN);    // -2.147.483.648
8      printf("UINT_MAX = %u\n", UINT_MAX);  // +4.294.967.295
9      printf("LONG_MAX = %ld\n", LONG_MAX);
10     printf("DBL_MAX = %e\n", DBL_MAX);    // il pi? grande double
        ↪ positivo
11     printf("DBL_MIN = %e\n", DBL_MIN);    // il pi? piccolo double
        ↪ positivo normalizzato
12     return 0;
13 }
```

La competenza chiave non è memorizzare questi numeri: è sapere *dove cercarli*. Aprire `<limits.h>` dalla documentazione di sistema è una delle prime abitudini da coltivare.

1.2 Codifica IEEE 754: Perché $0.1 + 0.2 \neq 0.3$

Un `double` ha 64 bit. Lo standard **IEEE 754** li divide così:

Tabella

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

Il valore rappresentato è:

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

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

1.2.1 Perché esiste il bias 1023?

L'esponente reale può essere **negativo** (numeri molto piccoli) o **positivo** (numeri molto grandi).

Gli 11 bit dell'esponente possono rappresentare valori da 0 a 2047. Per rappresentare esponenti negativi, il *bias* 1023 "centra" la gamma:

- $e_{\text{mem}} = 0 \rightarrow e_{\text{reale}} = -1023$ (numeri minuscoli)
- $e_{\text{mem}} = 2047 \rightarrow e_{\text{reale}} = +1024$ (numeri enormi)

Così un double può rappresentare valori che vanno da circa (10^{-308}) a (10^{308}) .

1.2.2 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 (mi sono "spostato" di 4 posizioni dopo la virgola)
- esponente memorizzato = $-4 + 1023 = 1019$
- mantissa = primi 52 bit dell'espansione infinita

1.2.3 Tabella dei campi del double per 0.1

Tabella			
Campo	Bit (valore binario)	Significato	Valore decimale
Segno	0	Numero positivo	+
Esponente	01111111011 (1019)	Esponente memorizzato	Esponente reale = $1019 - 1023 = -4$

1.3 Formati di Stampa printf – La Guida Completa

1.3.1 Tutti i Formati Essenziali

In L2 abbiamo usato principalmente `\{%d` e `\{%f`. Ecco il quadro completo dei formati che userete più spesso in questo corso:

Tabella

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

1.3.2 Controllo della Precisione Decimale

In L2 abbiamo visto `\{%f` con il formato di default (6 decimali). Possiamo controllare esattamente quante cifre mostrare usando il modificatore `.<N>` tra `\{%` e la lettera di formato:

- `\{%.2f` -> 2 cifre dopo la virgola (es. 3.14)
- `\{%.0f` -> solo la parte intera, senza virgola (es. 3)
- `\{%.3e` -> notazione scientifica con 3 cifre decimali (es. 3.142e+00)
- `\{%.17f` -> 17 cifre: utile per esporre gli errori di arrotondamento IEEE 754

```

1  #include <stdio.h>
2
3  int main(void) {
4      double pi = 3.14159265358979;
5      printf("Default:    %f\n", pi); // 3.141593
6      printf("2 decimali: %.2f\n", pi); // 3.14
7      printf("0 decimali: %.0f\n", pi); // 3
8      printf("Scientifica: %.3e\n", pi); // 3.142e+00
9      return 0;
10 }
```

1.3.3 Ampiezza Minima e Padding

Quando dobbiamo stampare dati in forma tabellare allineata, possiamo specificare l'**ampiezza minima** del campo aggiungendo un numero intero tra `\{ }%` e la lettera del formato:

Tabella

Specificatore	Effetto
<code>\{ }%10d</code>	Occupi 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

Esempio pratico – generare una tabella ben allineata:

```
1 #include <stdio.h>
2
3 int main(void) {
4     printf("%-20s %8s %10s\n", "Prodotto", "Quantit?", "Prezzo");
5     printf("%-20s %8d %10.2f\n", "Mela", 12, 0.89);
6     printf("%-20s %8d %10.2f\n", "Cioccolato", 3, 2.50);
7     printf("%-20s %8d %10.2f\n", "Acqua minerale", 6, 0.45);
8     return 0;
9 }
```

Output:

```
1 Prodotto          Quantit?      Prezzo
2 Mela              12          0.89
3 Cioccolato        3           2.50
4 Acqua minerale    6           0.45
```

Suggerimento

Documentazione completa di `printf` Non è necessario memorizzare tutti i flag. Consultate man 3 `printf` nel terminale oppure devdocs.io/c/io/fprintf per la documentazione interattiva completa.

1.4 Costanti: Tre Modi per Dare un Nome a un Valore Fisso

Immaginate di scrivere il valore 3.14159 cinque volte in uno stesso programma. Cosa succede se vi siete sbagliati di una cifra in un solo punto? Le costanti nominali risolvono questo problema.

1.4.1 Letterali Tipizzati

I valori che inserisco direttamente nelle istruzioni del programma vengono chiamati **letterali**. Un valore letterale come 75 o 3.14 ha un tipo implicito. Il compilatore lo determina automaticamente, ma a volte dobbiamo essere espliciti:

```
1 75          // int
2 75u         // unsigned int
3 75l         // long int
4 3.14        // double (attenzione: non float!)
5 3.14f       // float
6 3.14159L    // long double
```

Questo è importante quando si passano argomenti a funzioni che si aspettano un tipo specifico, o quando si sommano tipi diversi.

1.4.2 const – Variabile Immutabile

Il modo più diretto e moderno per definire una costante nominale è la keyword `const`:

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

Il compilatore garantisce che nessun'altra parte del codice possa modificare il valore. Se tentate di farlo, otterrete un errore di compilazione:

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

Nota

Regola fondamentale per `const` Dichiarazione e inizializzazione devono avvenire sulla stessa riga. `const int x;` seguito da `x = 5;` è un errore: una costante deve essere inizializzata al momento della sua definizione.

Quando usare `const`: ogni volta che un valore ha un significato semantico preciso e non deve mai cambiare nel programma (aliquote fiscali, costanti fisiche, dimensioni di array, etc.).

1.4.3 #define – Sostituzione del Preprocessore

Un'alternativa storica a `const` è la **macro del preprocessore**:

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

A differenza di `const`, `\{\}#define` non crea una variabile: il **preprocessore** sostituisce meccanicamente ogni occorrenza del nome con il testo corrispondente *prima ancora che il compilatore veda il codice*.

Possiamo osservare questo comportamento direttamente usando il flag `-E` di GCC, che ferma il processo dopo la fase di preprocessing e scrive il risultato in un file `.i`:

```
1 /* demo_define.c ? senza nessuna #include per vedere l'output pulito */
2 #define VOTO_MINIMO 18
3
4 int main(void) {
5     return VOTO_MINIMO;
6 }
```

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

Output (semplificato):

```
1 # "demo_define.c"
2 # "<built-in>" 1
3 ...
4
5 int main(void) {
6     return 18;
7 }
```

Il nome `VOTO\_{}\_MINIMO` è completamente scomparso: il compilatore vedrà solo 18. Per convenzione i nomi di macro si scrivono in **MAIUSCOLO** per distinguerli visivamente dalle variabili.

1.4.4 Costanti Predefinite dalla Libreria Standard

La libreria standard offre già molte costanti pronte all'uso, definite tramite `\{}\#define`:

- `M\{}\_PI` (in `<math.h>`) -> pi con massima precisione da `double`
- `M\{}\_E` (in `<math.h>`) -> il numero di Eulero *e*
- `INT\{}\_MAX`, `INT\{}\_MIN` (in `<limits.h>`) -> limiti degli interi
- `DBL\{}\_MAX`, `DBL\{}\_EPSILON` (in `<float.h>`) -> limiti e precisione dei `double`

1.5 Conversioni di Tipo: Implicite ed Esplicite

1.5.1 Conversione Implicita (Promozione)

Quando un'espressione mescola tipi diversi, il C promuove automaticamente al tipo "più capiente" per evitare la perdita di informazioni:

- `int + double` -> il compilatore converte l'`int` in `double` prima di sommare
- `double` assegnato a `int` -> la parte decimale viene **troncata** (non arrotondata!)

```

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

```

Questo troncamento silenzioso è una fonte comune di bug. Attenzione: né `-Wall` né `-Wextra` lo segnalano; con gcc serve il flag `-Wconversion` (warning: conversion from 'double' to 'int' may change value). Se il warning compare, non ignoratelo.

1.5.2 Cast Esplicito

Per convertire un tipo in modo deliberato e controllato, si usa il **cast esplicito**: si scrive il tipo di destinazione tra parentesi tonde prima dell'espressione.

```

1 int a = 5, b = 2;
2
3 double risultato = (double)a / b;   // 2.5 ? cast prima della divisione
4 double sbagliato = (double)(a / b); // 2.0 ? divisione intera, poi cast
5
6 printf("%.1f vs %.1f\n", risultato, sbagliato);

```

Attenzione

L'ordine del cast è fondamentale `(double)a / b` e `(double)(a / b)` producono risultati diversi! Nel primo caso il cast avviene prima della divisione, quindi il compilatore esegue una divisione reale. Nel secondo, la divisione intera è già avvenuta (perdendo il resto) quando il cast viene applicato.

1.5.3 Arrotondamento con `<math.h>`: `round`, `floor`, `ceil`

Il troncamento implicito di `(int)x` non è sempre il comportamento desiderato. La libreria `<math.h>` offre tre funzioni di arrotondamento con semantiche diverse:

- `round(x)` -> arrotondamento tradizionale: 2.4 -> 2.0, **2.5 -> 3.0**
- `floor(x)` -> sempre verso il basso (*pavimento*): 2.9 -> 2.0, **-2.1 -> -3.0**
- `ceil(x)` -> sempre verso l'alto (*soffitto*): 2.1 -> 3.0, **-2.9 -> -2.0**

Tutte e tre restituiscono un `double`. Per ottenere un `int`, combinate con il cast:

```

1 #include <math.h>
2
3 int n = (int)round(7.6);   // n = 8
4 int m = (int)floor(7.6);   // m = 7
5 int p = (int)ceil(7.2);    // p = 8

```

1.6 La Libreria <math.h>

1.6.1 Perché una Libreria Separata?

Le funzioni trigonometriche, le potenze e le radici richiedono istruzioni macchina specializzate (FPU — Floating Point Unit). Il C non le include nel nucleo del linguaggio: si importano esplicitamente con `\{\}\#include <math.h>`.

Nota

Compilare con `-lm` su Linux/GCC Su sistemi Linux con GCC, la libreria matematica deve essere linkata esplicitamente aggiungendo `-lm` al comando di compilazione: `gcc programma.c -lm -o programma`. Su macOS e con Clang questo non è necessario.

1.6.2 Le Funzioni Principali

Tabella

Funzione	Descrizione	Esempio
<code>sqrt(x)</code>	Radice quadrata	<code>sqrt(9.0) -> 3.0</code>
<code>pow(x, y)</code>	x^y	<code>pow(2, 10) -> 1024.0</code>
<code>fabs(x)</code>	Valore assoluto per <code>double</code>	<code>fabs(-3.5) -> 3.5</code>
<code>sin(x), cos(x), tan(x)</code>	Funzioni trigonometriche (in radianti)	<code>sin(M\{\}_PI) ~ 0.0</code>
<code>log(x)</code>	Logaritmo naturale ($\ln$)	<code>log(M\{\}_E) -> 1.0</code>
<code>log10(x)</code>	Logaritmo in base 10	<code>log10(100.0) -> 2.0</code>
<code>round(x), floor(x), ceil(x)</code>	Arrotondamenti (vedi §5)	

La costante `M\{\}_PI` definisce pi con la massima precisione disponibile nel tipo `double`.

1.6.3 Demo Pratica: Calcolatore di Moto Uniformemente Accelerato

Il seguente programma utilizza tutto ciò che abbiamo visto in questa lezione: tipi, formati con padding, costanti, <math.h> e casting:

```

1  #include <stdio.h>
2  #include <math.h>
3
4  int main(void) {
5      double v0, a, t;
6
7      printf("Velocit? iniziale (m/s): ");
8      scanf("%lf", &v0);

```

```

9   printf("Accelerazione (m/s2):  ");
10  scanf("%lf", &a);
11  printf("Tempo (s):                ");
12  scanf("%lf", &t);
13
14  double s = v0 * t + 0.5 * a * t * t; // spazio percorso
15  double vf = v0 + a * t;              // velocit? finale
16  double ec = 0.5 * vf * vf;           // energia cinetica
    ↪ specifica (massa=1 kg)
17
18  printf("\n--- Risultati ---\n");
19  printf("Spazio percorso:      %10.3f m\n", s);
20  printf("Velocit? finale:      %10.3f m/s\n", vf);
21  printf("Energia cinetica:     %10.3f J/kg\n", ec);
22
23  return 0;
24 }

```

Suggerimento

Dove trovare la documentazione completa di `<math.h>` devdocs.io/c/numeric/math elenca tutte le funzioni con firma, descrizione ed esempi.

1.7 Overflow, Underflow, inf e NAN

1.7.1 Cosa Succede Quando Superiamo i Limiti degli Interi?

Un `int` a 32 bit ha un valore massimo di `INT_MAX` = 2.147.483.647. Cosa succede aggiungendo 1?

```

1  #include <stdio.h>
2  #include <limits.h>
3
4  int main(void) {
5      int x = INT_MAX;
6      printf("INT_MAX      = %d\n", x);
7      printf("INT_MAX + 1 = %d\n", x + 1); // overflow!
8      return 0;
9  }

```

Output:

```

1  INT_MAX      = 2147483647
2  INT_MAX + 1 = -2147483648

```

Il risultato è **negativo**: siamo tornati al valore minimo rappresentabile, come un orologio che dopo le 23:59 torna alle 00:00. Questo fenomeno si chiama **overflow**. Attenzione: qui il compilatore **non** emette alcun warning, nemmeno con `-Wall -Wextra`, perché il valore di `x` è noto solo durante l'esecuzione. `gcc` avvisa (`-Woverflow`, attivo di default) solo quando l'overflow avviene in un'espressione costante calcolata in compilazione, come `INT_MAX`

+ 1 scritto direttamente. Per scoprire l'overflow durante l'esecuzione si può compilare con `-fsanitize=undefined`, che stampa runtime error: signed integer overflow.

Overflow degli interi: comportamento indefinito

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

1.7.2 Overflow e Valori Speciali nei Numeri in Virgola Mobile

I double si comportano diversamente dagli interi: invece di "wrapare", producono **valori speciali** definiti dallo standard IEEE 754:

```
1 #include <stdio.h>
2
3 int main(void) {
4     float troppo_grande = 10e10f * 10e30f;
5     printf("%f\n", troppo_grande); // inf
6     printf("%f\n", 0.0 / 0.0);    // nan
7     printf("%f\n", 1.0 / 0.0);    // inf
8     printf("%f\n", -1.0 / 0.0);   // -inf
9     return 0;
10 }
```

- **inf**: il risultato è troppo grande per essere rappresentato (overflow floating point).
- **nan** (Not A Number): risultato di un'operazione matematicamente indefinita ($0.0/0.0$, $\text{sqrt}(-1.0)$, etc.).
- **-inf**: infinito negativo.

La caratteristica più pericolosa di questi valori è che si **propagano**: qualsiasi operazione che coinvolge nan produce nan. Un nan che entra in un calcolo complesso può silenziosamente corrompere tutti i risultati successivi senza generare errori.

1.7.3 La Regola d'Oro: Compilate Sempre con `-Wall -Wextra`

```
1 gcc -Wall -Wextra -o programma programma.c
```

Questi flag attivano i principali avvisi del compilatore (ad esempio formati `printf` errati, variabili inutilizzate o non inizializzate). Non coprono tutto: le conversioni implicite con perdita di dati richiedono `-Wconversion` e l'overflow su variabili si scopre solo in esecuzione (`-fsanitize=undefined`). Un programma che compila con warnings non è un programma corretto.

1.8 Quiz di Autovalutazione

Mettetevi alla prova con queste 10 domande per verificare la comprensione dei concetti della lezione. Cliccate su ogni domanda per scoprire la risposta corretta e la spiegazione.

Quiz 1

Domanda

1. Qual è la differenza fondamentale tra `int` e `unsigned int` in termini di rappresentazione in memoria?

- A) `unsigned int` occupa il doppio dei byte di `int`.
- B) `int` e `unsigned int` occupano lo stesso numero di byte; cambia solo l'interpretazione del bit più significativo.
- C) `unsigned int` non può essere usato in espressioni aritmetiche.
- D) `int` è più preciso di `unsigned int` nei calcoli in virgola mobile.

Soluzione
Risposta corretta: B
Spiegazione: Entrambi i tipi occupano tipicamente 4 byte (32 bit). In `int`, il bit più significativo è il bit del segno (0=positivo, 1=negativo) che dimezza il range positivo. In `unsigned int`, quel bit contribuisce al valore numerico, portando il massimo da ~2 miliardi a ~4 miliardi.

Quiz 2

Domanda

2. Quale specificatore di formato va usato in `printf` per stampare correttamente un valore di tipo `long int`?

- A) `\{%d`
- B) `\{%ld`
- C) `\{%li`
- D) Sia `\{%ld` che `\{%li` sono corretti.

Soluzione
Risposta corretta: D
Spiegazione: Per `long int`, sia `\{%ld` (long decimal) che `\{%li` (long integer) sono corretti e producono lo stesso output. La scelta convenzionale è `\{%ld`.

Quiz 3

Domanda

3. Cosa produce il codice `printf("\{%%.17f\}\n", 0.1 + 0.2);` in C?

- A) `0.300000000000000000`
- B) Un errore di compilazione perché la precisione richiesta è troppo alta.
- C) `0.300000000000000004` (o un valore simile diverso da 0.3 esatto).

- D) 0.30000000 (7 cifre decimali, il massimo per double).

Soluzione
Risposta corretta: C
Spiegazione: Per lo standard IEEE 754, né 0.1 né 0.2 si rappresentano esattamente in base 2. La loro somma introduce un errore di arrotondamento che diventa visibile aumentando la precisione di stampa. Questo è il motivo fondamentale per cui non si devono confrontare due double con ==.

Quiz 4

Domanda

4. Cosa stampa il seguente codice?

```
1 int a = 5, b = 2;  
2 double r = (double)(a / b);  
3 printf("%.1f\n", r);
```

- A) 2.5
- B) 2.0
- C) 0.0
- D) Un errore di compilazione.

Soluzione
Risposta corretta: B
Spiegazione: Il cast (double) viene applicato dopo la divisione. Poiché sia a che b sono int, la divisione a / b è una divisione intera che produce 2 (troncando il resto). Solo successivamente il valore 2 viene convertito in 2.0. Per ottenere 2.5 bisogna scrivere (double) a / b.

Quiz 5

Domanda

5. Quale delle seguenti affermazioni su const è corretta?

- A) Una variabile const può essere modificata usando il cast (int).
- B) Una variabile const può essere dichiarata senza inizializzarla e inizializzata in seguito.
- C) Il compilatore emette un errore se si tenta di modificare una variabile const.
- D) const è equivalente a \{\}\#define e produce lo stesso codice macchina.

Soluzione
Risposta corretta: C
Spiegazione: Il compilatore C garantisce l'immutabilità di una variabile `const` emettendo un errore di compilazione per qualsiasi tentativo di assegnamento successivo. A differenza di `\{\}\#define, const` ha un tipo e uno scope ben definiti, ed è soggetto al type checking del compilatore.

Quiz 6

Domanda

6. Il flag `-E` di GCC serve a:

- A) Attivare tutti i warning di errore (`-E` sta per "Errors enabled").
- B) Fermare la compilazione dopo la fase di preprocessing e produrre un file `.i`.
- C) Ottimizzare il codice generato per la massima efficienza.
- D) Abilitare le estensioni GNU non standard del linguaggio.

Soluzione
Risposta corretta: B
Spiegazione: Con `gcc -E sorgente.c -o sorgente.i`, GCC esegue solo il preprocessing (che risolve `\{\}\#include, \{\}\#define` e le direttive condizionali) e scrive il risultato nel file `.i`. Questo è utile per capire cosa "vede" effettivamente il compilatore dopo che le macro sono state espanse.

Quiz 7

Domanda

7. Cosa restituisce `floor(-2.1)` in C?

- A) `-2.0`
- B) `-3.0`
- C) `2.0`
- D) `-2` (di tipo `int`).

Soluzione
Risposta corretta: B
Spiegazione: `floor` restituisce il numero intero più grande che sia minore o uguale all'argomento (in italiano: arrotonda sempre verso il basso, cioè verso $-\infty$). Per `-2.1`, il numero intero più grande che è ancora ≤ -2.1 è `-3`. Nota: tutte le funzioni di arrotondamento di `<math.h>` restituiscono un `double`.

Quiz 8

Domanda

8. Quale dei seguenti formati `printf` stampa il numero 42 occupando esattamente 8 caratteri con zeri a sinistra?

- A) `\{\}%8d`
- B) `\{\}%-8d`
- C) `\{\}%08d`
- D) `\{\}%z8d`

Risposta corretta: C
Spiegazione: `\{\}%08d` specifica un'ampiezza minima di 8 caratteri con padding tra-
 mite zeri (flag 0). Produce `00000042`. `\{\}%8d` produce 42 (spazi a sinistra). `\{\}%-8d`
 produce 42 (allineato a sinistra con spazi a destra).

Soluzione

Quiz 9

Domanda

9. Cosa stampa questo codice?

```
1 #include <stdio.h>
2 #include <limits.h>
3 int main(void) {
4     int x = INT_MAX;
5     printf("%d\n", x + 1);
6     return 0;
7 }
```

- A) 2147483648
- B) 0
- C) -2147483648
- D) Il programma non compila.

Risposta corretta: C
Spiegazione: Aggiungere 1 a `INT_MAX` causa un overflow: il valore "wrapa" e torna al valore minimo rappresentabile da un `int` a 32 bit, che è -2147483648. Il compilatore non emette alcun warning (nemmeno con `-Wall -Wextra`), perché l'overflow avviene su una variabile durante l'esecuzione. Tecnicamente si tratta di comportamento indefinito (undefined behavior), ma in pratica su architetture comuni si osserva questo comportamento.

Soluzione

Quiz 10

Domanda

10. Su Linux con GCC, perché è necessario aggiungere `-lm` al comando di compilazione quando si usa `<math.h>`?

- A) Per abilitare le estensioni GNU non standard.
- B) Per istruire il linker a includere la libreria matematica (`libm`) che contiene il codice delle funzioni come `sqrt` e `sin`.
- C) Perché `-lm` attiva la modalità "long math" che usa numeri a 128 bit.
- D) Non è necessario: `gcc` include automaticamente tutte le librerie standard.

Soluzione
Risposta corretta: B
Spiegazione: In C, la libreria matematica (`libm`) è separata dalla libreria standard C (`libc`). Il flag `-lm` dice al linker di collegare `libm` all'eseguibile finale, rendendo disponibili le implementazioni di `sqrt`, `sin`, `pow`, etc. Senza questo flag, il linker non troverà queste funzioni e la compilazione fallirà con errori "undefined reference". Su macOS e con Clang questo non è necessario perché `libm` è inclusa automaticamente.

1.9 Esercizi Pratici

Esercizio 1: Tabella dei Limiti

Domanda

Scrivete un programma che stampi una tabella formattata con tre colonne (TIPO, DIMENSIONE, VALORE MASSIMO) per i tipi `int`, `unsigned int`, `long` e `unsigned long`, usando `<limits.h>`. Assicuratevi che le colonne siano allineate correttamente usando i modificatori di ampiezza di `printf`.

Spiegazione: Usiamo `\t` per la colonna del tipo (allineata a sinistra, 20 caratteri), `\t` per la dimensione (8 caratteri, `\t` perché `sizeof` restituisce `size_t`) e l'ampiezza appropriata per il valore massimo.

```

1  #include <stdio.h>
2  #include <limits.h>
3
4  int main(void) {
5      printf("%-20s %8s %25s\n", "TIP0", "BYTE", "VALORE MASSIMO");
6      printf("%-20s %8zu %25d\n", "int", "int",
7             sizeof(int), "unsigned int", "unsigned int",
8             sizeof(long), "long", "long", "long", "long",
9             printf("%-20s %8zu %25lu\n", "unsigned long", "unsigned long",
10                  sizeof(long), ULONG_MAX);
11     return 0;

```

Soluzione

Esercizio 2: Divisione con Cast

Domanda

Scrivete un programma che legga due interi `a` e `b` da tastiera e stampi: il risultato della divisione intera, il risultato della divisione reale (usando il cast), e il resto.

Soluzione

Spiegazione: Il cast (double) a prima della divisione promuove l'operazione a divisione reale. L'operatore \{\}\% funziona solo tra interi e restituisce il resto della divisione intera.

```

1  #include <stdio.h>
2
3  int main(void) {
4      int a, b;
5      printf("Inserisci il dividendo: ");
6      scanf("%d", &a);
7      printf("Inserisci il divisore: ");
8      scanf("%d", &b);
9
10     int quoziente = a / b;
11     double reale = (double)a / b;
12     int resto = a % b;
13
14     printf("Divisione intera: %d\n", quoziente);
15     printf("Divisione reale: %.4f\n", reale);
16     printf("Resto: %d\n", resto);
17
18     return 0;
19 }
```

Esercizio 3: Area e Perimetro con**Domanda**

$M\{\}_PI$

Scrivete un programma che legga il raggio di un cerchio e stampi area e circonferenza usando la costante $M\{\}_PI$ definita in `<math.h>`. Formattate l'output con 4 cifre decimali.

Spiegazione: `M_PI` è una macro predefinita in `<math.h>` che contiene il valore di π con la massima precisione del tipo `double`. Non è necessario definire manualmente `3.14159`.

```

1  #include <stdio.h>
2  #include <math.h>
3
4  int main(void) {
5      double raggio;
6      printf("Inserisci il raggio: ");
7      scanf("%lf", &raggio);
8
9      double area = M_PI * raggio * raggio;
10     double circonferenza = 2.0 * M_PI * raggio;
11
12     printf("Area: %.4f\n", area);
13     printf("Circonferenza: %.4f\n", circonferenza);
14
15     return 0;
16 }
```

Soluzione

Esercizio 4: Calcolatore di Ipotenusa

Domanda

Scrivete un programma che legga i due cateti di un triangolo rettangolo e calcoli l'ipotenusa usando `sqrt` e `pow`.

Spiegazione: Il teorema di Pitagora si implementa direttamente con `sqrt` e `pow`. Nota-
re che `a*a` è preferibile a `pow(a, 2)` per prestazioni (meno overhead), ma entrambe
le versioni sono corrette.

```

1  #include <stdio.h>
2  #include <math.h>
3
4  int main(void) {
5      double a, b;
6      printf("Cateo a: ");
7      scanf("%lf", &a);
8      printf("Cateo b: ");
9      scanf("%lf", &b);
10
11     double ipotenusaa = sqrt(pow(a, 2) + pow(b, 2));
12     // oppure, equivalentemente: sqrt(a*a + b*b)
13     printf("Ipotenusaa: %.4f\n", ipotenusaa);
14     return 0;
15 }

```

Soluzione

Esercizio 5: Convertitore Angoli Radianti/Gradi

Domanda

Scrivete un programma che legga un angolo in gradi e stampi il suo seno, coseno e tangente, dopo averlo convertito in radianti (usando `M_PI`). Formattate l'output a 6 cifre decimali.

Spiegazione: La formula di conversione è $\text{rad} = \text{gradi} \times \pi / 180$. Le funzioni trigonometriche del C lavorano sempre in radianti.

```

1  #include <stdio.h>
2  #include <math.h>
3
4  int main(void) {
5      double gradi;
6      printf("Inserisci un angolo in gradi: ");
7      scanf("%lf", &gradi);
8
9      double radianti = gradi * M_PI / 180.0;
10
11     printf("Angolo in radianti: %.6f\n", radianti);
12     printf("Seno: %.6f\n", sin(radianti));
13     printf("Coseno: %.6f\n", cos(radianti));
14     printf("Tangente: %.6f\n", tan(radianti));
15
16     return 0;
17 }
```

Soluzione

Esercizio 6: Tabella di Arrotondamenti

Domanda

Scrivete un programma che legga un numero reale e stampi in una tabella i risultati di $(\text{int})x$ (troncamento), $\text{round}(x)$, $\text{floor}(x)$ e $\text{ceil}(x)$.

Spiegazione: Confrontare i quattro metodi sullo stesso valore (es. -2.7) evidenzia chiaramente la differenza tra troncamento, arrotondamento tradizionale, arrotondamento verso il basso e verso l'alto.

```

1 #include <stdio.h>
2 #include <math.h>
3
4 int main(void) {
5     double x;
6     printf("Inserisci un numero reale: ");
7     scanf("%lf", &x);
8
9     printf("\n%-20s %10.4f\n", "Valore originale:", x);
10    printf("\n%-20s %10d\n", "Troncamento (int):", (int)x);
11    printf("\n%-20s %10.4f\n", "round(x):", round(x));
12    printf("\n%-20s %10.4f\n", "floor(x):", floor(x));
13    printf("\n%-20s %10.4f\n", "ceil(x):", ceil(x));
14
15    return 0;
16 }
```

Soluzione

Esercizio 7: Demo Overflow

Domanda

Scrivete un programma che dimostri l'overflow degli interi: partite da `INT_MAX`, sommategli 1 e stampate il risultato. Poi risolvete il problema promuovendo l'operazione a `long`.

Spiegazione: `x + 1` produce un overflow perché l'addizione avviene tra due `int` (il letterale 1 è un `int`). `(long)x + 1` promuove prima `x` a `long`, poi la somma avviene in aritmetica a 64 bit e il risultato è corretto.

```

1 #include <stdio.h>
2 #include <limits.h>
3
4 int main(void) {
5     int x = INT_MAX;
6     printf("INT_MAX\n", x);
7     printf("INT_MAX + 1\n", x + 1);
8     printf("INT_MAX + 1 = %ld\n", (long)x + 1);
9     printf("promozione a long\n", (long)x + 1);
10    return 0;
11 }
```

Soluzione

Esercizio 8: Cedolino Stipendio con

Domanda

const e Padding

Scrivete un programma che legga il reddito lordo mensile da tastiera e stampi un cedolino formattato che mostri: reddito lordo, imponibile IRPEF (23% della parte fino a 15.000€/anno), contributi INPS (9.19%), e netto mensile. Definite tutte le aliquote come costanti const.

Spiegazione: Le aliquote definite con const rendono il codice auto-documentante e sicuro. Il padding con \{\}\%-25s e \{\}\%10.2f produce un output tabellare allineato e professionale.

```

1  #include <stdio.h>
2  #include <math.h>
3
4  int main(void) {
5      const double ALIQUOTA_IRPEF = 0.23;
6      const double ALIQUOTA_INPS = 0.0919;
7
8      double lordo_mensile;
9      printf("Reddito lordo mensile (?) : ");
10     scanf("%lf", &lordo_mensile);
11
12     double inps = lordo_mensile * ALIQUOTA_INPS;
13     double irpef = lordo_mensile * ALIQUOTA_IRPEF;
14     double netto = lordo_mensile - inps - irpef;
15
16     printf("\n--- CEDOLINO ---\n");
17     printf("%10.2f ?\n", "Lordo mensile:", lordo_mensile);
18     printf("%10.2f ?\n", "Contributi INPS:", inps);
19     printf("%10.2f ?\n", "Ritenuta IRPEF:", irpef);
20     printf("%10.2f ?\n", "Netto mensile:", netto);
21
22     return 0;
23 }
```

Soluzione

Esercizio 9: Verifica Teorema di Pitagora con Tolleranza

Domanda

Scrivete un programma che legga tre numeri reali e verifichi se possono essere i lati di un triangolo rettangolo, usando fabs e una tolleranza di 1e-6 per confrontare i double (senza usare if/else).

Spiegazione: Non si usa `==` per confrontare i double (errori di arrotondamento IEEE 754). Si calcola la differenza assoluta con `fabs` e si verifica che sia sotto la soglia di tolleranza $1e-6$. Il risultato dell'espressione `(differenza < 1e-6)` è 1 (Vero) o 0 (Falso), utilizzabile direttamente senza `if`.

```

1 #include <stdio.h>
2 #include <math.h>
3
4 int main(void) {
5     double a, b, c;
6     printf("Inserisci i tre lati: ");
7     scanf("%lf%lf%lf", &a, &b, &c);
8
9     // c deve essere il lato pi? lungo (ipotenusa candidata)
10    double ipotenusasq = c * c;
11    double cateti_sq = a * a + b * b;
12    double differenza = fabs(ipotenusasq - cateti_sq);
13
14    int e Rettangolo = (differenza < 1e-6);
15    printf("? un triangolo rettangolo (con c come ipotenusa): %d\n",
16           e_Rettangolo);
17    return 0;
18 }
```

Soluzione

Esercizio 10: Notazione Scientifica e Formati

Domanda

Scrivete un programma che stampi il numero di Avogadro (6.022×10^{23}) e la costante di Planck (6.626×10^{-34}) in tre formati diversi: `\{ }\%f`, `\{ }\%e`, e `\{ }\%g`. Osservate le differenze nell'output.

Spiegazione: \{\} tenta di stampare tutti i decimali (inutile per numeri molto grandi o piccoli), \{\}e forza sempre la notazione scientifica, \{\}g sceglie automaticamente il formato più compatto e leggibile. Per costanti fisiche, \{\}e o \{\}g sono quasi sempre la scelta migliore.

```
1 #include <stdio.h>
2
3 int main(void) {
4     double avogadro = 6.022e23;
5     double planck = 6.626e-34;
6
7     printf("Numero di Avogadro:\n");
8     printf("%f\n", avogadro);
9     printf("%e\n", avogadro);
10    printf("%g\n", avogadro);
11
12    printf("Costante di Planck:\n");
13    printf("%f\n", planck);
14    printf("%e\n", planck);
15    printf("%g\n", planck);
16
17    return 0;
18 }
```

Soluzione