

# C: Intro to C

# C cos'è?



Il linguaggio C è un linguaggio di programmazione di medio livello, ideato per offrire un'efficace combinazione tra controllo a basso livello sull'hardware e astrazione a livello di struttura dati.

È caratterizzato da una sintassi compatta: il programma si basa su funzioni e blocchi delimitati dalle parentesi graffe { }; i tipi di dato fondamentali sono interi (int), caratteri (char), numeri in virgola mobile (float, double) e puntatori, che permettono di gestire direttamente indirizzi di memoria.

Il flusso di controllo include costrutti standard quali if/else, switch, for, while e do/while, e il caricamento dinamico di moduli avviene tramite direttive di compilazione (#include) e di definizione (#define).

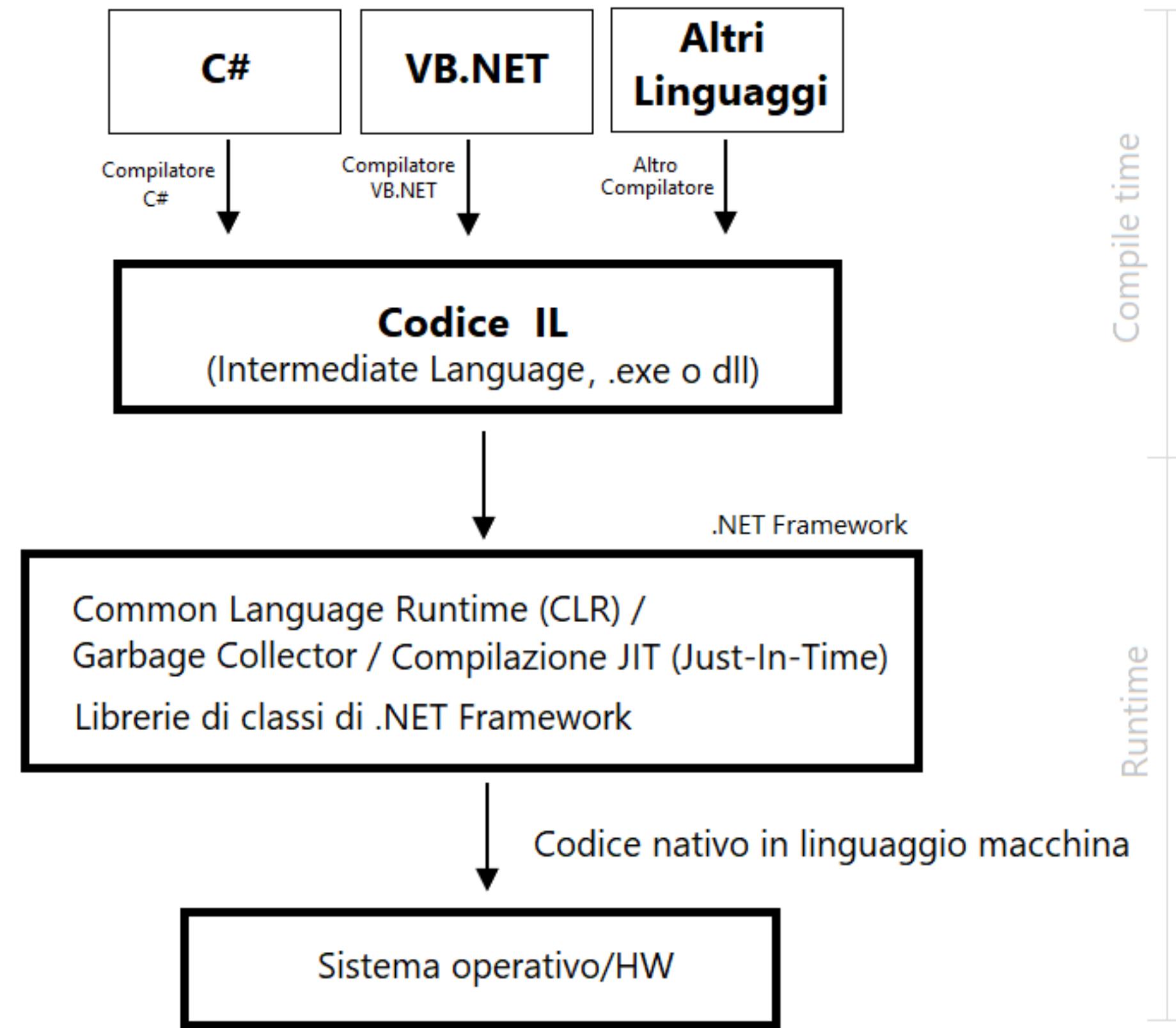
# C cos'è?

La gestione della memoria è manuale: l'allocazione e la deallocazione si effettuano rispettivamente con le funzioni malloc/calloc e con free.

La modularità e il riutilizzo del codice sono supportati dalle unità di traduzione (file sorgente .c e header .h) e dallo scope dei simboli a livello di file o globale.

L'efficienza nel runtime e la portabilità del codice sono tra i suoi punti di forza, grazie a compilatori ottimizzanti che traducono il C in codice macchina vicino a quello scritto a mano, pur mantenendo un grado di indipendenza dall'architettura sottostante.

# C cos'è?



# C cos'è?

C nasce all'inizio degli anni '70 presso i Bell Laboratories come evoluzione del linguaggio B, sviluppato da Ken Thompson.

Dennis Ritchie lo progettò per supportare lo sviluppo del sistema operativo UNIX, con l'obiettivo di superare i limiti prestazionali e strutturali che B mostrava sui primi computer DEC PDP.

Ritchie introdusse in C il concetto di tipi di dato e puntatori, una grammatica più ricca e sistemi di macro per la preprocessed, mantenendo però semplicità di implementazione dei compilatori

# C cos'è?

Il risultato fu un linguaggio sufficientemente vicino all'assemblatore per permettere un controllo fine dell'hardware, ma con costrutti di alto livello per la scrittura di codice modulare e leggibile.

La prima specifica formale comparve nel 1978 con il libro *The C Programming Language* di Brian Kernighan e Dennis Ritchie, che definì quella che oggi chiamiamo “versione K&R”.

In seguito, lo standard ANSI C del 1989 (ISO C90) codificò le funzionalità chiave, garantendo portabilità e interoperabilità tra sistemi diversi, e da allora il linguaggio ha continuato ad evolversi attraverso revisioni internazionali.

# C cos'è?

```
#include <iostream> ← File di intestazione
```

```
using namespace std; ← Namespace standard
```

```
int main() {
```

```
    cout << "Hello World" << endl;
```

```
    return 0;
```

```
}
```

L'output che viene stampato

Corpo della funzione

# Principali librerie del linguaggio C

- `<stdio.h>`

Fornisce le **funzioni** per l'**input/output standard**, quali `printf` per la **stampa su schermo** e `scanf` per la **lettura da tastiera**, oltre alla **gestione di file** tramite `fopen`, `fread`, `fwrite` e `fclose`.

- `<stdlib.h>`

Contiene **funzioni** per la **gestione della memoria dinamica** (`malloc`, `calloc`, `realloc`, `free`), il **controllo di processo** (`exit`, `system`) e **conversioni di stringhe a numeri** (`atoi`, `strtol`).

- `<string.h>`

Offre operazioni su **stringhe e memoria**: **copia** (`strcpy`, `memcpy`), **concatenazione** (`strcat`), **confronto** (`strcmp`, `memcmp`) e **ricerca di sotto-stringhe** (`strstr`).

# Principali librerie del linguaggio C

## <math.h>

- Include funzioni matematiche standard come sin, cos, exp, log, pow e sqrt, utili per calcoli scientifici e ingegneristici.

## <time.h>

- Permette di manipolare e formattare date e orari: time, difftime, gmtime, localtime e strftime per generare stringhe con formati personalizzati.

## <ctype.h>

- Fornisce macro e funzioni per il test e la conversione di caratteri: isalpha, isdigit, toupper, tolower, che operano sui valori int rappresentanti caratteri.

# Principali librerie del linguaggio C

## <errno.h>

- Definisce la variabile globale `errno` e costanti di errore per interpretare i codici di ritorno delle funzioni di libreria in caso di fallimento.

## <limits.h> e <float.h>

- Specificano i limiti dei tipi fondamentali (`INT_MAX`, `CHAR_BIT`, `DBL_MAX`, etc.), garantendo portabilità delle costanti relative a interi e virgola mobile.

## <assert.h>

- Contiene la macro `assert`, che verifica condizioni in fase di esecuzione e abortisce il programma se una verifica logica fallisce, utile per il debug.

# Introduzione pratica

**Il programma “Hello World” è tradizionalmente il primo passo nell’apprendimento di un nuovo linguaggio di programmazione.**

**Il suo scopo principale è mostrare il flusso di base di compilazione ed esecuzione di un programma C: dall’inclusione delle librerie necessarie, alla definizione della funzione main(), fino alla stampa di un messaggio su schermo e al corretto termine del programma con un valore di ritorno.**

**Pur essendo estremamente semplice, questo esempio mette in luce i concetti fondamentali del paradigma procedurale in C.**

# Introduzione pratica

- **#include <stdio.h>**: direttiva del preprocessore per **includere la libreria standard di input/output**.
- **int main(void)**: punto di ingresso del programma; restituisce un **intero al sistema operativo**.
- **printf(...)**: funzione per **stampare testo sullo stdout**.
- **Punto e virgola ()**: termina ogni istruzione in C.
- **Valore di ritorno**: **return 0**; indica che il programma è **terminato con successo**.
- **Compilazione**: trasformare il codice sorgente in eseguibile, tipicamente con **gcc hello.c -o hello**.
- **Esecuzione**: avviare l'eseguibile generato, ad esempio **./hello**.

# Introduzione pratica

- `#include <stdio.h>`

*Fa sì che il preprocessore inserisca il codice necessario per usare funzioni di input/output come printf.*

- `int main(void)`

*Definisce la funzione main, che non prende argomenti (void) e restituisce un int. È il primo codice eseguito all'avvio del programma.*

- `printf("Hello, World!\n");`

*Chiama la funzione printf per visualizzare la stringa. \n è un carattere speciale che provoca un "a capo" dopo il testo.*

- `return 0;`

*Termina la funzione main restituendo 0 al sistema operativo, convenzionalmente sinonimo di esecuzione riuscita.*

```
1. #include <stdio.h>
2. // 1) Includo la libreria per printf
3. int main(void) {
4.     // 2) Funzione principale: punto di ingresso
5.     // 3) Stampa "Hello, World!" seguito da un ritorno a capo
6.     printf("Hello, World!\n");
7.     // 4) Ritorno 0 per indicare al sistema che è tutto ok
8.     return 0;
9. }
```

# Linguaggio naturale vs artificiale

Un linguaggio naturale è la forma di comunicazione che si è evoluta spontaneamente tra gli esseri umani (ad es. italiano, inglese, cinese).

È ricco di ambiguità, sfumature e dipende dal contesto culturale.

Un linguaggio artificiale (o formale), come il C, è stato invece progettato ad hoc per scopi specifici - nel caso dei linguaggi di programmazione, per impartire istruzioni chiare e non ambigue al computer.

I linguaggi artificiali possiedono una sintassi rigorosa e una semantica ben definita, in modo che ogni programma venga interpretato o compilato sempre nello stesso modo.

# Linguaggio naturale vs artificiale

## Origine

- **Naturali:** evolvono culturalmente e storicamente.
- **Artificiali:** creati deliberatamente da persone o comitati.

## Ambiguità

- **Naturali:** molteplici significati possibili.
- **Artificiali:** non ammettono ambiguità.

## Regole

- **Naturali:** grammatica e lessico meno rigidi.
- **Artificiali:** sintassi e semantica formalmente specificate.

## Scopo

- **Naturali:** comunicazione tra persone.
- **Artificiali:** comunicazione “persona-macchina” (o macchina-macchina).

## Esempi

- **Naturali:** italiano, inglese, spagnolo.
- **Artificiali:** C, Java, Python, linguaggi di markup come HTML.

# Introduzione pratica

- **Commento**

*Le righe che iniziano con // sono scritte in linguaggio naturale e servono a chiarire lo scopo del codice per chi lo legge.*

- **Separazione di livelli**

*I commenti in italiano non influiscono sull'esecuzione, mentre il codice formale (C) viene tradotto in istruzioni macchina senza interpretazioni soggettive.*

```
1. #include <stdio.h> // 1) Includo la libreria per le funzioni di input/output
2.
3. int main(void) {
4.     // 2) Punto di ingresso del programma
5.     // 3) Questo è un commento in linguaggio naturale (italiano),
6.     //     che descrive a parole cosa farà la riga successiva.
7.
8.     printf("Differenza tra linguaggi naturali e artificiali\n");
9.     // 4) Questa è una stringa formale, interpretata senza ambiguità dalla macchina.
10.
11.    return 0;
12.    // 5) Restituisco 0 al sistema operativo: esecuzione avvenuta con successo
13.}
```

# Il linguaggio macchina

Il linguaggio macchina è il livello più basso di rappresentazione di un programma, composto da sequenze di bit (0 e 1) che il processore interpreta direttamente come istruzioni: operazioni aritmetiche, spostamenti di dati, controlli di flusso, ecc.

Ogni CPU ha il proprio set di istruzioni macchina (instruction set architecture, ISA).

A differenza dei linguaggi di alto livello, qui non esistono parole chiave leggibili: ogni istruzione è un codice binario che attiva circuiti hardware precisi.

Il compilatore traduce i sorgenti C in assembly e poi in linguaggio macchina, producendo un file eseguibile.

# Il linguaggio macchina

## Bit e Byte

- Il linguaggio macchina è una sequenza di bit raggruppati in byte (tipicamente 8 bit).

## Instruction Set Architecture (ISA)

- Specifica il formato delle istruzioni (opcode + operand) e i registri disponibili.

## Opcode

- Porzione di bits che identifica l'operazione (es. ADD, MOV, JMP).

## Operand

- Registri o indirizzi di memoria su cui opera l'instruction code.

## Endianness

- Ordine di disposizione dei byte in memoria (little- vs big-endian).

## Traduzione

- C → Assembly → Machine code tramite il compilatore e l'assembler.

## Eseguibilità

- Il processore legge e decodifica ciclicamente le istruzioni macchina dalla memoria.

# Programmazione di alto livello

I linguaggi di alto livello (High-Level Programming Languages) sono progettati per essere facilmente comprensibili dagli umani: utilizzano parole chiave leggibili, strutture astratte come funzioni e oggetti, e gestiscono automaticamente dettagli di basso livello (memoria, registri, istruzioni macchina).

Grazie a astrazione, portabilità e gestione semplificata delle risorse, permettono allo sviluppatore di concentrarsi sulla logica applicativa e non sui dettagli hardware.

Il compilatore o l'interprete traducono poi il codice in linguaggio macchina o in un bytecode intermedio.

# Processo di compilazione

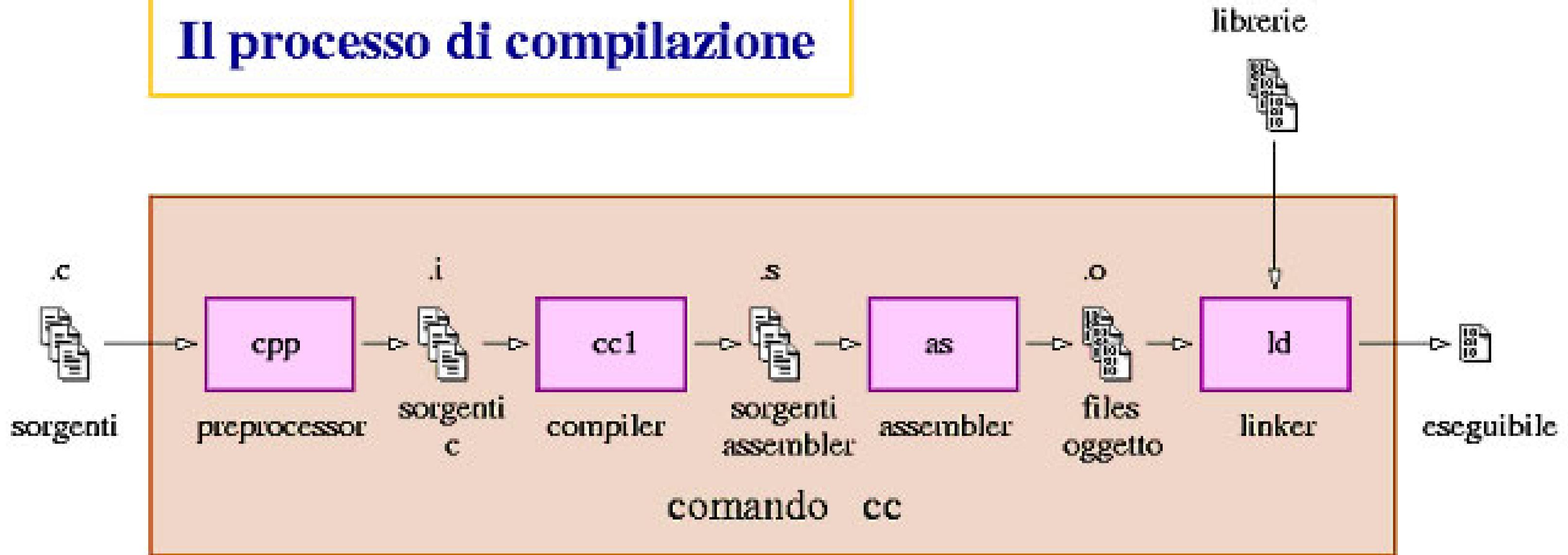
Il processo di compilazione trasforma il codice sorgente C in un eseguibile pronto per il processore.

Si articola in più fasi distinte: il preprocessore espande direttive come `#include` e `#define`; il compilatore analizza la sintassi e genera codice assembly; l'assembler traduce l'assembly in codice macchina producendo file oggetto (.o); il linker risolve riferimenti fra più file oggetto e librerie, creando l'eseguibile finale.

Questo workflow garantisce separazione di responsabilità, riusabilità di moduli e ottimizzazioni localizzate in ciascuna fase.

# Processo di compilazione

## Il processo di compilazione



# Processo di compilazione

## Preprocessore

- Gestisce `#include`, macro e condizioni (`#ifdef`, `#ifndef`).

## Analisi lessicale e sintattica

- Il compilatore verifica la correttezza del codice e costruisce l'albero sintattico (AST).

## Ottimizzazione

- Il compilatore può applicare ottimizzazioni come eliminazione di codice morto, inlining, loop unrolling.

## Generazione di assembly

- Traduzione dell'AST in istruzioni di assembly specifiche per l'architettura.

## Assemblaggio

- Conversione da assembly in bytecode macchina nei file oggetto (.o).

## Linking

- Unione di file oggetto e librerie statiche/dinamiche in un unico eseguibile.

## Strumenti comuni

- `gcc` o `clang` gestiscono internamente tutte le fasi; si possono invocare separatamente `-E`, `-S`, `-c`.
- Flag come `-g` inseriscono informazioni di debug; `-O` controlla livello di ottimizzazione.

# Processo di compilazione

*Prendiamo questo codice come esempio.*

```
1.// file: sum.c
2.#include <stdio.h>
3.
4.int sum(int a, int b) {
5.    return a + b; // Somma due interi
6.}
7.
8.int main(void) {
9.    int x = 10, y = 20;
10.   int s = sum(x, y);
11.   printf("La somma di %d e %d è %d\n", x, y, s);
12.   return 0;
13.}
```

# Processo di compilazione

- **Preprocessore**

1. `gcc -E sum.c -o sum.i`

*Espande #include <stdio.h> e macro, generando sum.i.*

- **Compilatore → Assembly**

1. `gcc -S sum.i -o sum.s`

*sum.s contiene istruzioni assembly (ad es. mov, add, call printf, ret).*

- **Assembler → Oggetto**

1. `gcc -c sum.s -o sum.o`

*sum.o è un file oggetto con codice macchina non ancora collegato.*

- **Linker → Eseguibile**

1. `gcc sum.o -o sum`

*Risolve simboli come printf usando librerie standard, producendo ./sum.*

- **Esecuzione**

1. `./sum`

- **Stampa:**

- **La somma di 10 e 20 è 30**



# C: Elementi pratici

# Elementi pratici

Un programma in C è composto da poche righe che illustrano il flusso fondamentale: includere librerie, definire la funzione `main()`, dichiarare variabili, eseguire operazioni elementari e restituire un valore al sistema operativo.

Anche se minimalista, questo tipo di programma mostra la struttura essenziale di ogni applicazione C: le direttive del preprocessore, la sequenza di istruzioni, l'uso di funzioni base per input/output e il termine con `return`.

Imparare a scrivere e compilare questi esempi getta le basi per affrontare programmi più complessi in cui si introducono decisioni, cicli e strutture dati.

# Elementi pratici - printf()

La funzione `printf()` fa parte della libreria standard `<stdio.h>` ed è il principale mezzo per produrre output testuale su `stdout` (tipicamente il terminale).

Accetta una stringa di formato contenente speciatori (es. `%d`, `%f`, `%s`) che vengono sostituiti dai valori degli argomenti successivi, formattati in modo appropriato.

Internamente `printf()` gestisce buffering e conversioni di tipo, consentendo di presentare numeri e stringhe con precisione e controllo sulla larghezza, i decimali e l'allineamento.

# Elementi pratici - printf()

## Stringa di formato

- Testo in chiaro + **specificatori % per inserire valori.**

## Specificatori di conversione

- **%d, %i** → interi decimali
- **%U** → interi unsigned
- **%f, %.2f** → floating-point (con precisione)
- **%c** → singolo carattere
- **%s** → stringa terminata da \0
- **%%** → stampa il carattere %

## Flag e larghezza

- Es. **%5d** (minimo 5 caratteri, padding a sinistra), **%-5d** (allineato a sinistra), **%05d** (zero-padding).

# Elementi pratici - printf()

**%d**

- **Stampa l'intero a in decimale.**

**%5d e %-5d**

- **Riservano 5 spazi per l'intero, con padding di spazi a sinistra o a destra.**

**%05d**

- **Riempie di zeri a sinistra fino a 5 caratteri.**

**%.2f, %.4f**

- **Controllano il numero di cifre decimali del floating-point.**

**%c, %.3s**

- **Stampa un singolo carattere e una stringa troncata alle prime 3 lettere.**

**%%**

- **Sequenza per visualizzare il carattere % stesso.**

```
1. #include <stdio.h>
2.
3. int main(void) {
4.     int a = 42;
5.     float pi = 3.14159f;
6.     char c = 'X';
7.     char str[] = "Ciao";
8.
9.     // Stampa di interi con diversi flag e larghezze
10.    printf("Valore intero base decimale: %d\n", a);
11.    printf("Intero con padding: [%5d]\n", a);
12.    printf("Intero allineato a sinistra: [%-5d]\n", a);
13.    printf("Intero con zero-padding: [%05d]\n\n", a);
14.
15.    // Stampa di float con precisione variabile
16.    printf("Pi con due decimali: %.2f\n", pi);
17.    printf("Pi con quattro decimali: %.4f\n\n", pi);
18.
19.    // Stampa di char e stringa
20.    printf("Carattere: %c\n", c);
21.    printf("Stringa limitata a 3 caratteri: %.3s\n", str);
22.
23.    // Stampare il simbolo '%' a video
24.    printf("Percentuale di completamento: 100%%\n");
25.
26.    return 0;
27.}
```

# Elementi pratici - Variabili

Le variabili in C sono contenitori nominati che vengono riservati in memoria per conservare valori di un determinato tipo (interi, caratteri, floating point, ecc.).

Ogni variabile ha un tipo, che ne definisce la dimensione e l'insieme di valori possibili, e un identificatore, che è il nome con cui il programmatore vi accede.

Prima di usarle, le variabili devono essere dichiarate, indicando tipo e nome; facoltativamente si possono anche inizializzare assegnando subito un valore.

Il corretto uso delle variabili è fondamentale per gestire dati, passare parametri alle funzioni e mantenere lo stato o anche detto momentum del programma.

# Elementi pratici - Variabili

## Tipo

- Specifica la natura del dato: **int, char, float, double, ecc.**

## Identificatore

- Nome scelto dal programmatore: deve iniziare con lettera o underscore e può contenere lettere, cifre e underscore.

## Dichiarazione

- Sintassi: tipo nome; (es. **int x;**).

## Inizializzazione

- Assegnazione di valore alla dichiarazione: **int x = 5;.**

# Elementi pratici - Variabili

## Dichiarazione e inizializzazione

- **int a = 10;** riserva spazio per un intero e lo inizializza.

## Dichiarazione senza inizializzazione

- **char c; e float f;** riservano memoria ma contengono valori indeterminati finché non assegnati.

## Assegnazione successiva

- Si possono assegnare valori dopo la dichiarazione: **c = 'Z';, f = 3.14f; .**

## Uso in espressioni

- Le variabili possono essere combinate in operazioni: **int somma = a + b; .**

## Stampa formattata

- **%d per int, %u per unsigned int, %c per char, %.2f per float con due decimali.**

```
1. #include <stdio.h>
2.
3. int main(void) {
4.     // 1) Dichiarazione e inizializzazione di variabili intere
5.     int a = 10;      // variabile 'a' di tipo int, inizializzata a 10
6.     int b = 10;
7.
8.     // 2) Dichiarazione senza inizializzazione
9.     char c;
10.    float f;
11.
12.    // 3) Assegnazione di valori successiva alla dichiarazione
13.    c = 'Z';        // variabile char che contiene il carattere 'Z'
14.    f = 3.14f;     // variabile float che contiene +/- π
15.
16.    // 4) Utilizzo delle variabili in un'operazione
17.    int somma = a + b; // sommo a e b, risultato in 'somma'
18.
19.    // 5) Stampa dei valori
20.    printf("a = %d, b = %u\n", a, b);
21.    printf("c = %c, f = %.2f\n", c, f);
22.    printf("somma = %d\n", somma);
23.
24.    return 0;       // fine del programma
25.}
```

# Elementi pratici - Variabili

## Ambito (scope)

- **Locale:** variabili dichiarate dentro una funzione o un blocco {}; accessibili solo lì.
- **Globale:** dichiarate fuori da tutte le funzioni; visibili in tutto il file (o programmate con extern).

## Allocazione

- **Statica:** per variabili globali e locali su stack.
- **Dinamica:** usando puntatori e funzioni come malloc() per ottenere memoria da heap.

## Modificatori di tipo

- **signed, unsigned, short, long** per variare intervallo e segno.

# Elementi pratici - Variabili

Nella vita reale gli interi rappresentano quantità discrete (numero di persone, oggetti, passi...), mentre in C gli integer sono tipi di dato a larghezza fissa, codificati in bit secondo il formato two's-complement per i signed, e semplici binari per gli unsigned.

Un “valore intero” in C non è altro che una sequenza di bit interpretata come un numero, con un intervallo dipendente dalla dimensione e dal segno.

I letterali interi sono le rappresentazioni testuali dei valori costanti che scriviamo nel codice: possono essere in base decimale, ottale o esadecimale, e con suffissi per specificare tipo e segno (es. U per unsigned, L per long).

# Elementi pratici - Variabili

## Interi nella vita reale

- Quantità discrete, non frazionarie.
- Spesso limitate da contesti (es. posti in un teatro).

## Tipi interi in C

- **signed**: possono rappresentare numeri negativi e positivi.
- **unsigned**: solo valori non negativi, raddoppiano il valore massimo.
- **Modificatori di larghezza**: short, int, long, long long

# Elementi pratici - Variabili

```
1. #include <stdio.h>
2.
3. int main(void) {
4.     // 1) Letterali decimali
5.     int dec = 123;      // valore decimale 123
6.
7.     // 2) Letterali ottali (prefisso 0)
8.     unsigned int oct = 075; //  $075_8 = 7 \cdot 8 + 5 = 61_{10}$ 
9.
10.    // 3) Letterali esadecimali (prefisso 0x)
11.    long hex = 0xFF;      //  $0xFF_{16} = 15 \cdot 16 + 15 = 255_{10}$ 
12.
13.    // 4) Letterali con suffissi
14.    unsigned long big = 4294967295UL;
15.    // valore massimo di 32-bit unsigned
16.
17.    // 5) Dimostrazione di overflow (signed 8-bit simulato)
18.    signed char a = 127;  // massimo di signed char
19.    a = a + 1;          // overflow: da +127 → -128 in two's-complement
20.
21.    // 6) Stampa dei risultati
22.    printf("dec = %d\n", dec);
23.    printf("oct (075) = %u\n", oct);
24.    printf("hex (0xFF) = %ld\n", hex);
25.    printf("big = %lu\n", big);
26.    printf("overflow signed char: 127 + 1 = %d\n", a);
27.    return 0;
28.}
```

# Elementi pratici - Variabili

Nella vita reale i valori floating point rappresentano quantità non discrete, con parte frazionaria (ad es. temperatura, peso, distanza).

In C questi valori sono codificati secondo lo standard IEEE 754, che prevede un formato a mantissa ed esponente per rappresentare numeri molto grandi o molto piccoli, ma introduce errori di arrotondamento e limiti di precisione.

Esistono due tipi principali: float (precisione singola, circa 7 cifre decimali) e double (precisione doppia, circa 15 cifre decimali).

I letterali floating point sono costanti numeriche scritte nel codice: possono essere in notazione decimale con punto (es. 3.14), in notazione esponenziale (es. 1.2e-3) e accettano suffissi per specificare il tipo (f per float, l per long double).

# Elementi pratici - Variabili

## Tipi in C

- **float**: 32 bit, ~7 cifre significative.
- **double**: 64 bit, ~15 cifre significative.
- **long double**: almeno 80 bit su molte piattaforme, precisione maggiore.

## Errori di arrotondamento

- Somme successive possono accumulare imprecisioni.
- Confronti diretti (==) spesso inaffidabili.

## Letterali decimali

- Con punto: 3.14 (di default **double**).
- Con punto e suffisso: 3.14f (**float**), 3.14l (**long double**).

# Elementi pratici - Variabili

```
1. #include <stdio.h>
2. #include <float.h> // Per i limiti di precisione e range
3.
4. int main(void) {
5.     // 1) Dichiarazione e inizializzazione con letterali decimali
6.     float f1 = 3.14f;           // suffisso 'f' ⇒ tipo float
7.     double d1 = 3.14;          // default doppia precisione
8.
9.     // 2) Notazione esponenziale
10.    float f2 = 1.0e3f;         // 1·103 = 1000.0f
11.    double d2 = 5.0E-2;        // 5·10-2 = 0.05
12.
13.    // 3) Limiti di un float
14.    printf("Range float: [%e ... %e]\n", FLT_MIN, FLT_MAX);
15.    // 4) Dimostrazione di arrotondamento
16.    float sum = 0.0f;
17.    for(int i = 0; i < 1000000; i++) {
18.        sum += 0.000001f;       // accumulo un milionesimo un milione di volte
19.    }
20.
21.    // 5) Stampa dei risultati
22.    printf("f1 = %.6f, d1 = %.6f\n", f1, d1);
23.    printf("f2 = %.1f, d2 = %.2f\n", f2, d2);
24.    printf("Somma approssimata = %.6f (invece di 1.000000)\n", sum);
25.
26.    return 0;
27.}
```

# Elementi pratici - Variabili

In C, il concetto di booleano non è un tipo di dato nativo fino allo standard C99: originariamente si usava un intero (int) con valore 0 per il “falso” e qualsiasi valore diverso da 0 per il “vero”.

Con C99 è stato introdotto il tipo `_Bool`, accessibile tramite l'header `<stdbool.h>` che definisce l'alias `bool` e le costanti `true` e `false`.

Internamente `_Bool` occupa almeno un byte e assicura che ogni valore diverso da 0 venga ridotto a 1. I booleani migliorano la chiarezza del codice, rendendo esplicite le condizioni logiche e riducendo l'ambiguità rispetto all'uso di interi generici.

# Elementi pratici - Variabili

- **Dichiariamo bool e usiamo true/false grazie a <stdbool.h>.**
- **La funzione e\_pari restituisce un booleano che indica se un numero è pari.**
- **Nel main, l'if valuta il valore booleano e stampa il risultato.**
- **Infine mostriamo come stampare direttamente lo stato di un flag booleano.**

```
1. #include <stdio.h>
2. #include <stdbool.h>
3.
4. int main(void) {
5.     bool e_pari(int x) {
6.         return (x % 2) == 0;
7.     }
8.
9.     int numero = 7;
10.    if (e_pari(numero)) {
11.        printf("%d è pari\n", numero);
12.    } else {
13.        printf("%d è dispari\n", numero);
14.    }
15.
16.    // Uso diretto di true/false
17.    bool flag = false;
18.    printf("Flag è %s\n", flag ? "vero" : "falso");
19.
20.    return 0;
21.}
```

# Elementi pratici - Variabili

In C il tipo `char` rappresenta un singolo carattere e occupa esattamente un byte di memoria (di solito 8 bit).

Anche se il suo scopo principale è memorizzare caratteri ASCII, in realtà `char` è un tipo intero di ampiezza ridotta, e può essere `signed` o `unsigned` a seconda dell'implementazione (definito dallo standard).

Un `char signed` può rappresentare tipicamente valori da -128 a 127, mentre un `unsigned char` va da 0 a 255.

Oltre che per i caratteri, `char` è usato per operare a basso livello su dati grezzi (buffer di byte) e per costruire stringhe terminate dal carattere nullo '\0'. Le operazioni aritmetiche sul `char` ne manipolano il codice numerico (ad esempio incrementare un carattere ne restituisce il successivo nel codice ASCII).

# Elementi pratici - Variabili

- Usiamo `getchar()` per leggere un singolo char da input.
- Con le funzioni di `<ctype.h>` verifichiamo se il carattere è maiuscolo o minuscolo.
- Applichiamo `tolower` o `toupper` per convertirlo, sfruttando il fatto che `char` è in fondo un valore numerico corrispondente al codice ASCII.
- Stampiamo il risultato, mostrando come il tipo `char` possa essere manipolato sia come carattere sia come valore numerico.

```
1. #include <stdio.h>
2. #include <ctype.h> // per isupper, tolower
3.
4. int main(void) {
5.     char c;
6.     printf("Inserisci un carattere: ");
7.     c = getchar();           // legge un singolo carattere da stdin
8.
9.     if (isupper((unsigned char)c)) {
10.         // se è una lettera maiuscola, la converto in minuscola
11.         char lower = tolower((unsigned char)c);
12.         printf("Hai inserito una maiuscola. In minuscolo diventa: %c\n", lower);
13.     } else if (islower((unsigned char)c)) {
14.         // se è una lettera minuscola, la converto in maiuscola
15.         char upper = toupper((unsigned char)c);
16.         printf("Hai inserito una minuscola. In maiuscolo diventa: %c\n", upper);
17.     } else {
18.         printf("Il carattere '%c' non è una lettera alfabetica.\n", c);
19.     }
20.
21.     return 0;
22. }
```

# Elementi pratici - Variabili

In C le stringhe non sono un tipo primitivo, ma vengono rappresentate come vettori di char terminati dal carattere nullo '\0'.

Ogni stringa è quindi un array in cui ogni elemento corrisponde a un carattere, e la fine della stringa è indicata dal byte 0.

Questo approccio "a basso livello" permette di manipolare direttamente il buffer di caratteri (lettura, scrittura, concatenazione), ma impone anche di gestire manualmente la dimensione dell'array e di prestare attenzione a non superare i limiti (rischio di buffer overflow).

La libreria standard `<string.h>` fornisce funzioni utili come `strlen` (lunghezza), `strcpy/strncpy` (copia), `strcat/strncat` (concatenazione) e `strcmp` (confronto), tutte basate sul riconoscimento del terminatore nullo.

# Elementi pratici - Variabili

- Dichiariamo un array char nome[50] abbastanza grande per contenere l'input e il terminatore nullo.
- Usiamo fgets per leggere fino a 49 caratteri da stdin, evitando buffer overflow.
- Rimuoviamo il carattere di newline ('\n') eventualmente lasciato da fgets usando strcspn.
- Calcoliamo la lunghezza vera della stringa con strlen.
- Creiamo un buffer saluto inizializzato con "Ciao, ", poi concatenamo il nome con strncat specificando il massimo spazio residuo.
- Stampiamo il messaggio completo e la lunghezza del nome. Questo esempio mostra come le stringhe in C richiedano attenzione alla terminazione e alla gestione delle dimensioni.

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main(void) {
5.     char nome[50];           // buffer per la stringa
6.     printf("Come ti chiami? ");
7.     if (fgets(nome, sizeof(nome), stdin) != NULL) {
8.         // rimuovo l'eventuale newline finale
9.         nome[strcspn(nome, "\n")] = '\0';
10.
11.    // calcolo la lunghezza
12.    size_t len = strlen(nome);
13.
14.    // preparo un saluto concatenando due stringhe
15.    char saluto[100] = "Ciao, ";
16.    strncat(saluto, nome, sizeof(saluto) - strlen(saluto) - 1);
17.
18.    printf("%s! Il tuo nome contiene %zu caratteri.\n", saluto, len);
19. } else {
20.     printf("Errore nella lettura del nome.\n");
21. }
22.
23. return 0;
24. }
```

# Elementi pratici - `scanf()`

La funzione `scanf()` è il complemento di `printf()` e consente di leggere input formattato dallo `stdin` (tipicamente la tastiera).

Utilizza una stringa di formato simile a `printf()`, in cui specificatori come `%d`, `%f`, `%s` indicano il tipo di dato da leggere. Per ognuno, `scanf()` richiede l'indirizzo di una variabile dove memorizzare il valore acquisito.

È essenziale verificare il valore di ritorno di `scanf()`, che indica quante conversioni hanno avuto successo, per gestire correttamente errori di input.

# Elementi pratici - `scanf()`

## Stringa di formato

- Specificatori % analoghi a quelli di `printf()`.

## Argomenti

- Puntatori alle variabili (`&a`, `&pi`, `str`).
- Return value
- Numero di input letti con successo; EOF in caso di fine file o errore.

## Whitespace handling

- `%d`, `%f`, `%s` saltano automaticamente spazi bianchi e newline.

## Limitatori

- Per stringhe, `%Ns` per leggere al massimo N caratteri, evitando overflow del buffer.

# Elementi pratici - scanf()

**scanf("%d", &age)**

- Legge un intero, lo memorizza in age; controlla che sia stato letto correttamente.

**scanf("%f", &weight)**

- Legge un numero floating-point in weight; verifica il successo della conversione.

**scanf("%19s", name)**

- Legge una parola (fino al primo whitespace), massimo 19 caratteri + \0, per evitare overflow di name[20].

**Controllo del valore di ritorno**

- Se diverso da 1, significa che la conversione non è andata a buon fine o c'è stato EOF.

**Uso di %s**

- Non legge spazi interni: se servono frasi è necessario usare fgets().

```
1. #include <stdio.h>
2.
3. int main(void) {
4.     int age;
5.     float weight;
6.     char name[20];
7.
8.     // 1) Lettura di un intero
9.     printf("Inserisci la tua età: ");
10.    if (scanf("%d", &age) != 1) {
11.        printf("Input non valido per l'età.\n");
12.        return 1;
13.    }
14.
15.    // 2) Lettura di un float
16.    printf("Inserisci il tuo peso (es. 70.5): ");
17.    if (scanf("%f", &weight) != 1) {
18.        printf("Input non valido per il peso.\n");
19.        return 1;
20.    }
21.
22.    // 3) Lettura di una stringa con limite
23.    printf("Inserisci il tuo nome: ");
24.    if (scanf("%19s", name) != 1) { // %19s per evitare overflow
25.        printf("Input non valido per il nome.\n");
26.        return 1;
27.    }
28.
29.    // 4) Stampa dei dati letti
30.    printf("\nDati inseriti:\n");
31.    printf("Età : %d anni\n", age);
32.    printf("Peso : %.2f kg\n", weight);
33.    printf("Nome : %s\n", name);
34.
35.    return 0;
36.}
```

# Elementi pratici - scanf()

## Esercizio:

**Lettura di due numeri e di una stringa (nome), stampa della somma, scrivi un programma in C che:**

- **Chiede all'utente di inserire il proprio nome (una stringa).**
- **Chiede all'utente di inserire due numeri interi.**
- **Usa la funzione scanf per leggere il nome e i numeri inseriti dall'utente.**
- **Calcola la somma dei due numeri.**
- **Stampa a schermo il nome dell'utente e il risultato della somma.**

```
1. #include <stdio.h>
2.
3. int main() {
4.     char nome[50]; // Variabile per memorizzare il nome
5.     int num1, num2; // Variabili per i due numeri
6.     int somma; // Variabile per la somma
7.
8.     // Chiede all'utente di inserire il nome
9.     printf("Inserisci il tuo nome: ");
10.    scanf("%s", nome); // Legge una parola (senza spazi) come nome
11.
12.    // Chiede all'utente di inserire il primo numero intero
13.    printf("Inserisci il primo numero intero: ");
14.    scanf("%d", &num1);
15.
16.    // Chiede all'utente di inserire il secondo numero intero
17.    printf("Inserisci il secondo numero intero: ");
18.    scanf("%d", &num2);
19.
20.    // Calcola la somma
21.    somma = num1 + num2;
22.
23.    // Stampa il risultato
24.    printf("%s, la somma dei due numeri è: %d\n", nome, somma);
25.
26.    return 0;
27.}
```

# Elementi pratici - scanf()

**Esercizio:**

**Scrivi un programma in C che:**

- **Chiede all'utente di inserire il nome della propria città (una stringa).**
- **Chiede all'utente di inserire tre numeri decimali (float).**
- **Usa la funzione scanf per leggere la città e i numeri inseriti dall'utente.**
- **Calcola la media dei tre numeri.**
- **Stampa a schermo il nome della città e il risultato della media.**

```
1. #include <stdio.h>
2.
3. int main() {
4.     char citta[50]; // Variabile per la città
5.     float num1, num2, num3; // Variabili per i tre numeri decimali
6.     float media; // Variabile per la media
7.
8.     // Chiede all'utente di inserire il nome della città
9.     printf("Inserisci il nome della tua città: ");
10.    scanf("%s", citta); // Legge una parola (senza spazi)
11.
12.    // Chiede il primo numero decimale
13.    printf("Inserisci il primo numero decimale: ");
14.    scanf("%f", &num1);
15.
16.    // Chiede il secondo numero decimale
17.    printf("Inserisci il secondo numero decimale: ");
18.    scanf("%f", &num2);
19.
20.    // Chiede il terzo numero decimale
21.    printf("Inserisci il terzo numero decimale: ");
22.    scanf("%f", &num3);
23.
24.    // Calcola la media
25.    media = (num1 + num2 + num3) / 3;
26.
27.    // Stampa il risultato
28.    printf("La media dei tre numeri a %s è: %.2f\n", citta, media);
29.
30.    return 0;
31.}
```

# **Elementi pratici - Operatori**

# Elementi pratici - Operatori

In C un operatore è un simbolo (o sequenza di simboli) che indica al compilatore di eseguire una specifica operazione su uno o più operandi (espressioni, variabili o valori letterali): ad esempio somme, confronti, assegnamenti o manipolazioni di bit.

Ogni operatore ha un preciso dominio di applicabilità (tipi di dato ammessi), una precedenza (ordine di valutazione rispetto agli altri operatori in un'espressione complessa) e un'associatività (direzione di valutazione in caso di operatori di pari precedenza).

In fase di compilazione l'espressione viene scomposta secondo tali regole, infine vengono generate le istruzioni macchina che realizzano il calcolo o l'azione richiesta.

Gli operatori sono dunque il meccanismo sintattico e semantico che trasforma frammenti di codice in operazioni computazionali reali.

# Elementi pratici - Operatori

## Operatori aritmetici

- Eseguono le operazioni matematiche di base su operandi numerici: somma (+), sottrazione (-), moltiplicazione (\*), divisione (/) e resto (%).

## Operatori di incremento e decremento

- Modificano di 1 il valore di una variabile, in forma prefissa (++x, --x) o postfissa (x++, x--), con leggi di precedenza diverse.

## Operatori relazionali

- Confrontano due operandi e restituiscono un valore booleano (int 0 o 1): uguale (==), diverso (!=), maggiore (>), minore (<), maggiore o uguale (>=), minore o uguale (<=).

## Operatori logici

- Combinano espressioni booleane secondo la logica proposizionale: AND logico (&&), OR logico (||), negazione (!).

## Operatori di assegnamento

- Assegnano valori a variabili; includono l'assegnamento semplice (=) e le forme combinate con operazioni aritmetiche (es. +=, -=, \*=).

# Elementi pratici - Operatori aritmetici

Gli operatori aritmetici in C permettono di eseguire operazioni matematiche di base sui tipi numerici (interi e floating-point).

Essi sono incorporati nel linguaggio con una sintassi infissa, e ciascuno corrisponde a uno o più codici macchina che svolgono l'operazione sul processore.

Oltre alle operazioni fondamentali (somma, sottrazione, moltiplicazione, divisione, modulo), esistono operatori di incremento e decremento per modifiche snelle di variabili.

Conoscere la precedenza (priorità) e l'associatività di questi operatori è fondamentale per scrivere espressioni corrette e prevedibili.

# Elementi pratici - Operatori aritmetici

- Somma (+)
- Sottrazione (-)
- Moltiplicazione (\*)

## Divisione (/)

- Tra interi: divisione troncante.
- Tra floating-point: risultato reale.

## Modulo (%)

- Restituisce il resto della divisione intera.

## Incremento (++) e Decremento (--)

- Prefisso (++a, --a): modifica il valore prima di usarlo.
- Postfisso (a++, a--): usa il valore, poi lo modifica.

# Elementi pratici - operatori

- **a + b, a - b, a \* b:** eseguono le classiche operazioni matematiche tra interi.
- **a / b su interi:** il risultato viene troncato verso zero ( $10/3 = 3$ ).
- **a % b:** calcola il resto della divisione intera ( $10\%3 = 1$ ).
- **x / y su float:** divisione reale con parte decimale ( $\approx 3.333333$ ).
- **++a (prefisso):** prima incrementa a (da 10 a 11), poi restituisce 11.
- **b++ (postfisso):** restituisce prima il valore originale di b (3), poi incrementa b a 4.
- **Stampe:** usa **%d** per interi e **%.6f** per float con sei decimali; per stampare il simbolo **%** nel testo si scrive **%%**.

```
1. #include <stdio.h>
2.
3. int main(void) {
4.     int a = 10, b = 3;
5.     float x = 10.0f, y = 3.0f;
6.
7.     // Operazioni intere
8.     int sum    = a + b;    // somma: 13
9.     int diff   = a - b;    // sottrazione: 7
10.    int prod   = a * b;   // moltiplicazione: 30
11.    int quot   = a / b;   // divisione intera: 3 (troncamento)
12.    int mod    = a % b;   // modulo: 1
13.
14.    // Operazioni floating-point
15.    float fquot = x / y;  // divisione reale: circa 3.333333
16.
17.    // Incremento e decremento
18.    int pre_inc = ++a;    // a diventa 11, poi assegnato a pre_inc
19.    int post_inc = b++;   // post_inc = 3, poi b diventa 4
20.
21.    // Stampa dei risultati
22.    printf("Interi: %d + %d = %d\n", 10, 3, sum);
23.    printf("Interi: %d - %d = %d\n", 10, 3, diff);
24.    printf("Interi: %d * %d = %d\n", 10, 3, prod);
25.    printf("Interi: %d / %d = %d (troncato)\n", 10, 3, quot);
26.    printf("Interi: %d %% %d = %d (resto)\n", 10, 3, mod);
27.    printf("Float: %.6f / %.6f = %.6f\n", x, y, fquot);
28.    printf("Pre-incremento: ++10 = %d\n", pre_inc);
29.    printf("Post-incremento: 3++ = %d (b ora = %d)\n", post_inc, b);
30.
31.    return 0;
32.}
```

# Elementi pratici - Operatori

In C, quando un'espressione contiene più operatori, l'ordine di valutazione è determinato da due regole fondamentali: la priorità (precedenza) stabilisce quali operatori vengono calcolati per primi, mentre l'associazione (o binding) definisce in quale direzione si valuta un gruppo di operatori con la stessa precedenza (da sinistra a destra o viceversa).

Conoscere queste regole è cruciale per scrivere espressioni corrette e per evitare risultati inattesi; in caso di dubbi, è sempre consigliabile utilizzare le parentesi per esplicitare l'ordine voluto.

# Elementi pratici - Operatori

Tabella di precedenza principale (dalla più alta alla più bassa):

1. **Operatori postfissi:** (), [], ->, .
2. **Prefisso e unari:** ++, --, !, ~, &, \*, (type)
3. **Moltiplicazione, divisione e modulo:** \*, /, %
4. **Addizione e sottrazione:** +, -
5. **Operatori di spostamento bit a sinistra/destra:** <<, >>
6. **Confronti:** <, <=, >, >=
7. **Uguaglianza:** ==, !=
8. **AND bit a bit:** &
9. **XOR bit a bit:** ^
10. **OR bit a bit:** |
11. **AND logico:** &&
12. **OR logico:** \|\|
13. **Operatore condizionale ternario:** ?:
14. **Assegnamento:** =, +=, -=, ...
15. **Virgola:** ,

# Elementi pratici - Operatori

```
1. #include <stdio.h>
2.
3. int main(void) {
4.     int a = 2, b = 3, c = 4;
5.     int result1, result2, result3;
6.
7.     // 1) Senza parentesi: * ha precedenza su + e -
8.     result1 = a + b * c;      // equivale a a + (b * c) = 2 + 12 = 14
9.
10.    // 2) Override con parentesi
11.    result2 = (a + b) * c;   // (2 + 3) * 4 = 20
12.
13.    // 3) Associazione di assegnamenti (right-to-left)
14.    a = b = c + 1;          // equivale a b = (c + 1); a = b; => b = 5, a = 5
15.
16.    // 4) Combinazione di operatori logici e di confronto
17.    result3 = (a > b) && (b < c) || (c == 4);
18.    // valutazione step by step:
19.    // (a > b) -> (5 > 5) = false
20.    // (b < c) -> (5 < 4) = false
21.    // false && false = false
22.    // (c == 4) -> true
23.    // false || true = true => result3 = 1
24.
25.    // Stampa dei risultati
26.    printf("result1 (a + b * c) = %d\n", result1);
27.    printf("result2 ((a + b) * c) = %d\n", result2);
28.    printf("after a = b = c + 1: a = %d, b = %d\n", a, b);
29.    printf("result3 = %d (true = 1, false = 0)\n", result3);
30.
31.    return 0;
32.}
```

# Elementi pratici - Operatori relazionali

Gli operatori relazionali in C servono a confrontare due valori (numerici o espressioni) e restituiscono un valore di tipo `_Bool` (in C99) o un intero (0 per "falso", 1 per "vero").

Gli operatori disponibili sono:

- `==` : uguale a
- `!=` : diverso da
- `>` : maggiore di
- `<` : minore di
- `>=` : maggiore o uguale a
- `<=` : minore o uguale a

Ogni confronto produce 1 se la relazione è vera, altrimenti 0. Nell'ambito di un `if`, `while` o in qualsiasi contesto booleano, il risultato viene valutato come condizione.

La precedenza di questi operatori è inferiore a quella aritmetica, ma superiore a quella logica, e sono tutti associativi da sinistra a destra.

# Elementi pratici - Operatori relazionali

- Dichiariamo due interi a e b.
- Stampiamo i risultati dei diversi operatori relazionali usando l'operatore ternario per convertire 0/1 in "falso"/"vero".
- Mostriamo l'utilizzo di  $a < b$  all'interno di un if, che esegue il ramo "vero" se la condizione è soddisfatta. Questo dimostra come i confronti guidino il flusso di controllo.

```
1. #include <stdio.h>
2. #include <stdbool.h>
3.
4. int main(void) {
5.     int a = 10, b = 20;
6.
7.     printf("a = %d, b = %d\n", a, b);
8.
9.     // Esempi di confronto
10.    printf("a == b --> %s\n", (a == b) ? "vero" : "falso");
11.    printf("a != b --> %s\n", (a != b) ? "vero" : "falso");
12.    printf("a < b --> %s\n", (a < b) ? "vero" : "falso");
13.    printf("a > b --> %s\n", (a > b) ? "vero" : "falso");
14.    printf("a <= b --> %s\n", (a <= b) ? "vero" : "falso");
15.    printf("a >= b --> %s\n", (a >= b) ? "vero" : "falso");
16.
17.    // Uso in un if
18.    if (a < b) {
19.        puts("a è minore di b");
20.    } else {
21.        puts("a non è minore di b");
22.    }
23.
24.    return 0;
25.}
```

# Elementi pratici - Operatori logici

Gli operatori logici in C permettono di combinare o invertire il risultato di espressioni booleane (o di confronto) e restituiscono anch'essi un valore booleano (0 per "falso", 1 per "vero"):

- **&& (AND logico)**: dà "vero" solo se entrambe le espressioni sono vere; sfrutta short-circuit, perciò la seconda non viene valutata se la prima è falsa.
- **|| (OR logico)**: dà "vero" se almeno una delle espressioni è vera; anche qui c'è short-circuit, quindi la seconda non viene valutata se la prima è vera.
- **!** (NOT logico): nega il valore dell'espressione, da "vero" se l'espressione è falsa e viceversa.

La precedenza è tale che ! viene valutato prima di &&, che a sua volta viene valutato prima di ||.

# Elementi pratici - Operatori logici

- Dichiariamo due flag booleani (haPatente, haAssicurazione) e un intero eta.
- Nel primo if usiamo **&&** per richiedere entrambe le condizioni, e **!** per negare haAssicurazione.
- Nel secondo if verifichiamo se eta è compresa tra 18 e 65 (due confronti uniti da **&&**).
- Nel terzo controlliamo se è weekend o festa con **||**.
- Ogni stampa illustra il risultato della combinazione logica.

```
1. #include <stdio.h>
2. #include <stdbool.h>
3.
4. int main(void) {
5.     bool haPatente = true;
6.     bool haAssicurazione = false;
7.     int eta = 20;
8.
9.     // Uso di && e ||
10.    if (haPatente && haAssicurazione) {
11.        printf("Puoi guidare legalmente.\n");
12.    } else if (haPatente && !haAssicurazione) {
13.        printf("Hai la patente ma manca l'assicurazione.\n");
14.    } else if (!haPatente && haAssicurazione) {
15.        printf("Hai l'assicurazione ma non hai la patente.\n");
16.    } else {
17.        printf("Non puoi guidare.\n");
18.    }
19.
20.    // Controllo di un intervallo con AND logico
21.    if (eta >= 18 && eta <= 65) {
22.        printf("Sei in età lavorativa.\n");
23.    }
24.
25.    // Uso di OR logico
26.    bool weekend = false;
27.    bool festa = true;
28.    if (weekend || festa) {
29.        printf("È giorno di riposo.\n");
30.    }
31.
32.    return 0;
33.}
```

# Elementi pratici - Operatori logici

Scrivi un programma in C che:

- Chiede all'utente di inserire due numeri interi.
- Usa la funzione scanf per leggere i numeri.
- Calcola e stampa:
  - La somma
  - La differenza
  - Il prodotto
  - Il quoziente (divisione intera)
  - Il resto della divisione (operatore modulo)
- Verifica e stampa con gli operatori logici:
  - Se entrambi i numeri sono maggiori di zero (operatore AND: &&)
  - Se almeno uno dei due numeri è pari (operatore OR: ||)

```
1. #include <stdio.h>
2.
3. int main() {
4.     int num1, num2;
5.
6.     // Chiede i due numeri all'utente
7.     printf("Inserisci il primo numero intero: ");
8.     scanf("%d", &num1);
9.
10.    printf("Inserisci il secondo numero intero: ");
11.    scanf("%d", &num2);
12.
13.    // Operatori aritmetici
14.    printf("Somma: %d\n", num1 + num2);
15.    printf("Differenza: %d\n", num1 - num2);
16.    printf("Prodotto: %d\n", num1 * num2);
17.
18.    // Controllo per divisione e resto
19.    if(num2 != 0) {
20.        printf("Quoziente: %d\n", num1 / num2);
21.        printf("Resto della divisione: %d\n", num1 % num2);
22.    } else {
23.        printf("Non posso dividere per zero!\n");
24.    }
25.
26.    // Operatori logici
27.    if(num1 > 0 && num2 > 0) {
28.        printf("Entrambi i numeri sono maggiori di zero.\n");
29.    } else {
30.        printf("Almeno uno dei numeri NON è maggiore di zero.\n");
31.    }
32.
33.    if(num1 % 2 == 0 || num2 % 2 == 0) {
34.        printf("Almeno uno dei due numeri è pari.\n");
35.    } else {
36.        printf("Nessuno dei due numeri è pari.\n");
37.    }
38.
39.    return 0;
40.}
```

# Elementi pratici - Operatori di assegnamento

Gli operatori di assegnamento in C servono a trasferire un valore a una variabile; il risultato di un'operazione di assegnamento è il valore stesso assegnato, valutato come espressione.

L'operatore fondamentale è `=`, che prende a sinistra un l-value (variabile o elemento di array) e a destra un'espressione il cui valore viene copiato nella variabile.

Esistono inoltre operatori composti che combinano un'operazione aritmetica o bit-a-bit con l'assegnamento stesso, ad esempio `+=`, `-=`, `*=`, `/=`, `%=` per l'aritmetica e `&=`, `|=`, `^=`, `<<=`, `>>=` per i bit.

Questi operatori equivalgono a `x = x op y`, ma valutano `x` una sola volta, migliorando efficienza e leggibilità.

Tutti gli operatori di assegnamento hanno bassa precedenza (valgono meno di quasi tutti gli altri operatori) e associatività da destra, perciò in una catena come `a = b = c` si valuta prima `b = c` e poi `a = (b = c)`.

# Elementi pratici - Operatori logici

- L'operatore `=` realizza l'assegnamento diretto di un valore a una variabile.
- Gli operatori composti come `+=`, `\*=`, `%=` e `<=>` uniscono un'operazione aritmetica o bit-a-bit con l'assegnamento stesso, evitando di ripetere il nome della variabile e incrementando efficienza e chiarezza.
- La precedenza di assegnamento è bassa, e questi operatori sono associativi da destra, come mostrato dalla catena `a = b = c`.

```
1. #include <stdio.h>
2.
3. int main(void) {
4.     int x = 5;      // assegnamento semplice
5.     printf("x = %d\n", x);
6.
7.     x += 3;        // equivalente a x = x + 3
8.     printf("dopo x += 3 --> x = %d\n", x);
9.
10.    x *= 2;       // equivalente a x = x * 2
11.    printf("dopo x *= 2 --> x = %d\n", x);
12.
13.    x %= 4;       // equivalente a x = x % 4
14.    printf("dopo x %%= 4 --> x = %d\n", x);
15.
16.    x <= 1;        // shift a sinistra e assegnamento: x = x << 1
17.    printf("dopo x <= 1 --> x = %d\n", x);
18.
19.    // Catena di assegnamenti, associatività da destra
20.    int a, b, c = 7;
21.    a = b = c;      // prima b = c (b diventa 7), poi a = b (a diventa 7)
22.    printf("a = %d, b = %d, c = %d\n", a, b, c);
23.
24.    return 0;
25.}
```

# Elementi pratici - Operatori logici

## Esercizio: Operazioni di confronto e operatori logici

- Scrivi un programma in C che:
  - Chiede all'utente di inserire tre numeri interi.
  - Chiede all'utente di inserire un valore intero chiamato X (la soglia).
  - Usa la funzione scanf per leggere tutti i valori.
- Calcola e stampa:
  - La somma tra il primo e il secondo numero (`num1 + num2`)
  - La moltiplicazione tra il secondo e il terzo numero (`num2 * num3`)
- Verifica e stampa:
  - Se la somma è maggiore di X, stampa un messaggio, altrimenti uno diverso.
  - Se la moltiplicazione è minore di X, stampa un messaggio, altrimenti uno diverso.
- Operatori logici:
  - Se almeno uno dei numeri è negativo (usa l'operatore OR `||`)
  - Se tutti e tre i numeri sono diversi da zero (usa AND `&&`)
  - Se nessuno dei numeri è uguale a 100 (usa l'operatore NOT ! combinato con OR)

```
1. #include <stdio.h>
2.
3. int main() {
4.     int num1, num2, num3, X;
5.     int somma, moltiplicazione;
6.
7.     // Input dei tre numeri
8.     printf("Inserisci il primo numero intero: ");
9.     scanf("%d", &num1);
10.
11.    printf("Inserisci il secondo numero intero: ");
12.    scanf("%d", &num2);
13.
14.    printf("Inserisci il terzo numero intero: ");
15.    scanf("%d", &num3);
16.
17.    // Input della soglia X
18.    printf("Inserisci il valore della soglia X: ");
19.    scanf("%d", &X);
20.
21.    // Calcolo delle operazioni
22.    somma = num1 + num2;
23.    moltiplicazione = num2 * num3;
24.
25.    printf("La somma di num1 e num2 è: %d\n", somma);
26.    printf("La moltiplicazione di num2 e num3 è: %d\n", moltiplicazione);
27.
28.    // Verifica rispetto a X
29.    if(somma > X) {
30.        printf("La somma è maggiore di X.\n");
31.    } else {
32.        printf("La somma NON è maggiore di X.\n");
33.    }
34.
35.    if(moltiplicazione < X) {
36.        printf("La moltiplicazione è minore di X.\n");
37.    } else {
38.        printf("La moltiplicazione NON è minore di X.\n");
39.    }
40.
41.    // --- OPERATORI LOGICI ---
42.    // Almeno uno dei numeri è negativo (OR)
43.    if(num1 < 0 || num2 < 0 || num3 < 0) {
44.        printf("Almeno uno dei numeri è negativo.\n");
45.    } else {
46.        printf("Nessun numero è negativo.\n");
47.    }
48.
49.    // Tutti e tre i numeri sono diversi da zero (AND)
50.    if(num1 != 0 && num2 != 0 && num3 != 0) {
51.        printf("Tutti i numeri sono diversi da zero.\n");
52.    } else {
53.        printf("Almeno uno dei numeri è zero.\n");
54.    }
55.
56.    // Nessuno dei numeri è uguale a 100 (NOT combinato con OR)
57.    if(!(num1 == 100 || num2 == 100 || num3 == 100)) {
58.        printf("Nessuno dei numeri è uguale a 100.\n");
59.    } else {
60.        printf("Almeno uno dei numeri è uguale a 100.\n");
61.    }
62.
63.    return 0;
64.}
```

# Elementi pratici - Condizioni

# Elementi pratici - Condizioni

L'esecuzione condizionale in C permette di far compiere al programma scelte diverse in base al valore di un'espressione booleana (vero/falso).

L'istruzione **if** valuta una condizione e, se risulta vera (diversa da zero), esegue un blocco di codice; in alternativa, tramite il ramo **else**, è possibile specificare un secondo blocco da eseguire se la condizione è falsa (pari a zero).

Questo costrutto è fondamentale per controllare il flusso del programma e gestire situazioni diverse a runtime.

```
1. if (condizione) {  
2.  
3.     // blocco vero  
4.  
5. } else {  
6.  
7.     // blocco falso  
8.  
9. }
```

# Elementi pratici - Condizioni

## Condizione

- Qualsiasi espressione aritmetica o logica;

## Blocco di codice

- Può essere una singola istruzione o un gruppo racchiuso tra {}.

## Ramo else if

- Permette di concatenare più controlli in sequenza:  
1. `if (...) { ... } else if (...) { ... } else { ... }`

## Nesting

- È possibile annidare if dentro blocchi if/else.

## Operatori logici

- Spesso si combinano condizioni con &&, ||, !.

## Valori booleani

- In C non esiste tipo bool nativo (prima del C99); si usa int.

## Best practice

- Sempre usare le parentesi {} anche per singole istruzioni, per chiarezza e manutenzione.

# Elementi pratici - Condizioni

## Lettura del voto

- `scanf("%d", &voto);` memorizza il valore inserito in `voto`.

## Primo if...else

- `if (voto >= 60)` controlla se il voto è almeno 60;
- Se vero, stampa "Promosso"; altrimenti, "Insufficiente".

## Catena else if

- Permette di gestire più intervalli:
  - <60 → Debole
  - 60–74 → Sufficiente
  - 75–89 → Buono
  - >=90 → Ottimo

```
1. #include <stdio.h>
2.
3. int main(void) {
4.     int voto;
5.
6.     // 1) Chiedo all'utente di inserire un voto da 0 a 100
7.     printf("Inserisci il voto (0-100): ");
8.     scanf("%d", &voto);
9.
10.    // 2) Esecuzione condizionale: se voto >= 60 → passato, altrimenti no
11.    if (voto >= 60) {
12.        printf("Esito: Promosso\n");
13.    } else {
14.        printf("Esito: Insufficiente\n");
15.    }
16.
17.    // 3) Uso di else if per classificare in fasce
18.    if (voto < 60) {
19.        printf("Fascia: Debole\n");
20.    } else if (voto < 75) {
21.        printf("Fascia: Sufficiente\n");
22.    } else if (voto < 90) {
23.        printf("Fascia: Buono\n");
24.    } else {
25.        printf("Fascia: Ottimo\n");
26.    }
27.
28.    return 0;
29.}
```

# Elementi pratici - Condizioni

- **Confronto di uguaglianza e diversità:**

Puoi verificare se una variabile è uguale o diversa da un certo valore o da un'altra variabile. Ad esempio, puoi controllare se un numero inserito dall'utente è uguale a 10, oppure se una password inserita è diversa da quella richiesta.

- **Confronto di maggiore e minore:**

È possibile controllare se un valore è maggiore, minore, maggiore o uguale, oppure minore o uguale a un altro. Ad esempio, puoi verificare se una persona ha almeno 18 anni oppure se una temperatura è inferiore a zero.

- **Condizioni combinate con AND (e logico):**

Le condizioni possono essere collegate tra loro usando "e" logico. Ad esempio, puoi controllare se un numero è compreso tra due valori, come verificare che sia maggiore di 0 e minore di 100 contemporaneamente.

# Elementi pratici - Condizioni

- **Condizioni combinate con OR (o logico):**

Puoi collegare più condizioni alternative usando “o” logico. Ad esempio, puoi controllare se una variabile corrisponde a uno o a un altro valore, come controllare se un carattere inserito è una vocale oppure una consonante specifica.

- **Negazione (NOT):**

È possibile negare una condizione, cioè verificare che non sia vera. Ad esempio, puoi controllare che una variabile non sia uguale a zero oppure che una certa condizione non sia soddisfatta.

- **Confronto sul resto della divisione (modulo):**

Spesso si verifica se un numero è pari o dispari controllando se il resto della divisione per 2 è zero oppure no.

# Elementi pratici - Condizioni

- **Condizione logica (&&):**

La prima istruzione if controlla che il numero sia maggiore o uguale a 1 e minore o uguale a 100.

- **If annidato:**

All'interno del primo if, si trova un secondo if che controlla se il numero è pari o dispari usando

Se il numero è pari, viene stampato il relativo messaggio, altrimenti entra nell'else.

- **Else esterno:**

Se il numero non rispetta la condizione iniziale (non è tra 1 e 100), il programma stampa che il numero non è valido.

```
1. #include <stdio.h>
2.
3. int main() {
4.     int numero;
5.     printf("Inserisci un numero intero: ");
6.     scanf("%d", &numero);
7.
8.     // Condizione logica: il numero deve essere compreso tra 1 e
9.     // 100 (inclusi)
10.    if (numero >= 1 && numero <= 100) {
11.        printf("Il numero è compreso tra 1 e 100.\n");
12.
13.        // If annidato: verifica se il numero è pari o dispari
14.        if (numero % 2 == 0) {
15.            printf("Il numero è anche pari.\n");
16.        } else {
17.            printf("Il numero è invece dispari.\n");
18.        }
19.    } else {
20.        printf("Il numero NON è compreso tra 1 e 100.\n");
21.    }
22.    return 0;
23.}
```

# Elementi pratici - Condizioni

- Si include <string.h> per poter usare strcmp.
- L'utente inserisce una stringa (massimo 19 caratteri) e questa viene salvata nell'array password.
- Con strcmp(password, "segreto") == 0 si controlla se la stringa inserita è uguale a "segreto".
- strcmp restituisce 0 solo se le due stringhe sono identiche.
- Se la condizione è vera, il programma stampa che l'accesso è consentito.
- Altrimenti stampa che l'accesso è negato.

```
1. #include <stdio.h>
2. #include <string.h> // Serve per strcmp
3.
4. int main() {
5.     char password[20];
6.
7.     printf("Inserisci la password: ");
8.     scanf("%19s", password);
9.
10.    if(strcmp(password, "segreto") == 0) {
11.        printf("Accesso consentito.\n");
12.    } else {
13.        printf("Accesso negato.\n");
14.    }
15.
16.    return 0;
17.}
```

# Elementi pratici - Condizioni

**Esercizio: Verifica numero positivo o negativo**

**Scrivi un programma in C che:**

- **Chiede all'utente di inserire un numero intero.**
- **Usa l'istruzione if per verificare se il numero è positivo, negativo oppure zero.**
- **Stampa a schermo un messaggio che indica la natura del numero inserito.**
- **Dentro il primo if vada ad aggiungere un if annidato che controlla se il numero è superiore a 100 e stampi "wow"**

```
1. #include <stdio.h>
2.
3. int main() {
4.     int numero;
5.
6.     // Chiede un numero all'utente
7.     printf("Inserisci un numero intero: ");
8.     scanf("%d", &numero);
9.
10.    // Verifica se il numero è positivo, negativo o zero
11.    if (numero > 0) {
12.        printf("Il numero è positivo.\n");
13.        // If annidato: verifica se il numero è maggiore di 100
14.        if (numero > 100) {
15.            printf("wow\n");
16.        }
17.    } else if (numero < 0) {
18.        printf("Il numero è negativo.\n");
19.        if (numero < -100) {
20.            printf("- wow\n");
21.        }
22.    } else {
23.        printf("Il numero è zero.\n");
24.    }
25.
26.    return 0;
27.}
```

# Elementi pratici - Condizioni

**Esercizio: If, else if, else con operatori logici e if annidati**

**Scrivi un programma in C che:**

**Chiede all'utente di inserire un numero intero.**

**Usa una serie di condizioni con if, else if ed else, e operatori logici per verificare:**

**Se il numero è positivo e pari (usa &&):**

- **Se è anche maggiore di 50, stampa "Molto grande!" in un if annidato.**

**Se è negativo o maggiore di 100 (usa ||):**

- **Se è multiplo di 5, stampa "Multiplo di 5!" in un if annidato.**

**Altrimenti, stampa che il numero non soddisfa nessuna delle condizioni precedenti:**

- **Se il numero è diverso da zero, stampa anche "Numero diverso da zero".**

```
1. #include <stdio.h>
2.
3. int main() {
4.     int numero;
5.
6.     printf("Inserisci un numero intero: ");
7.     scanf("%d", &numero);
8.
9.     // Prima casistica: positivo e pari
10.    if (numero > 0 && numero % 2 == 0) {
11.        printf("Il numero è positivo e pari.\n");
12.        // If annidato
13.        if (numero > 50) {
14.            printf("Molto grande!\n");
15.        }
16.    }
17.    // Seconda casistica: negativo o maggiore di 100
18.    else if (numero < 0 || numero > 100) {
19.        printf("Il numero è negativo oppure è maggiore di 100.\n");
20.        // If annidato
21.        if (numero % 5 == 0) {
22.            printf("Multiplo di 5!\n");
23.        }
24.    }
25.    // Terza casistica: nessuna delle precedenti
26.    else {
27.        printf("Il numero non soddisfa nessuna delle condizioni precedenti.\n");
28.        // If annidato
29.        if (numero != 0) {
30.            printf("Numero diverso da zero.\n");
31.        }
32.    }
33.
34.    return 0;
35.}
```

# Elementi pratici - Condizioni - Switch

L'istruzione **switch** in C fornisce un modo più leggibile e strutturato di gestire molteplici rami condizionali basati sul valore di un'unica espressione (tipicamente un int o un char convertibile).

Al posto di annidare numerosi if...else if...else, switch confronta il risultato dell'espressione con ciascun case e trasferisce il controllo al blocco corrispondente.

Se nessun case coincide, viene eseguito il blocco default (opzionale). Per passare da un case all'altro senza interrompere il ciclo, si usa l'istruzione break; in sua assenza, si verifica il fall-through, cioè l'esecuzione continua nei case successivi.

# Elementi pratici - Condizioni - Switch

## Espressione di controllo

- Deve essere di tipo intero o enum; convertita in int.

## case

- Ogni etichetta case è un valore costante univoco.

## break

- Interrompe l'esecuzione del switch ed esce dal blocco.

## Fall-through

- Se manca break, l'esecuzione continua nel case successivo.

## default

- Cattura tutti i valori non coperti dai case; non è obbligatorio ma consigliato.

## Confronto a tempo di compilazione

- I valori di case devono essere costanti note a compilazione.

# Elementi pratici - Condizioni - Switch

**giorno = 3**

- **Variabile di controllo usata nel switch.**

**break dopo case 1**

- **Impedisce il passaggio ai case successivi; esce dal blocco switch.**

**Fall-through da case 3 a case 4**

- **Non essendoci break dopo Mercoledì, viene stampato anche "Giovedì". Utile per raggruppare comportamenti simili.**

**Fall-through in case 6 e case 7**

- **Entrambi eseguono lo stesso blocco "Weekend!" senza duplicazioni di codice.**

**default**

- **Gestisce eventuali valori di giorno non previsti (meno di 1 o più di 7).**

```
1. #include <stdio.h>
2.
3. int main(void) {
4.     int giorno = 3; // 1) Rappresentazione numerica del giorno (1=Lun, ..., 7=Dom)
5.
6.     printf("Giorno %d: ", giorno);
7.     switch (giorno) {
8.         case 1:
9.             printf("Lunedì\n");
10.            break; // 2) Esco dallo switch dopo aver eseguito case 1
11.        case 2:
12.            printf("Martedì\n");
13.            break;
14.        case 3:
15.            printf("Mercoledì\n");
16.            // 3) Nessun break: rientro voluto per mostrare metà settimana
17.        case 4:
18.            printf("Giovedì\n");
19.            break;
20.        case 5:
21.            printf("Venerdì\n");
22.            break;
23.        case 6:
24.        case 7:
25.            // 4) Fall-through per raggruppare Sabato e Domenica
26.            printf("Weekend!\n");
27.            break;
28.        default:
29.            printf("Giorno non valido\n");
30.    }
31.
32.    return 0;
33.}
```

# Elementi pratici - Condizioni - Switch

```
1. #include <stdio.h>
2.
3. int main() {
4.     char comando;
5.     printf("Inserisci un comando (a, b, c): ");
6.     scanf(" %c", &comando); // attenzione allo spazio prima di %c!
7.
8.     switch (comando) {
9.         case 'a':
10.             printf("Hai scelto il comando A.\n");
11.             break;
12.         case 'b':
13.             printf("Hai scelto il comando B.\n");
14.             break;
15.         case 'c':
16.             printf("Hai scelto il comando C.\n");
17.             break;
18.         default:
19.             printf("Comando non riconosciuto.\n");
20.     }
21.
22.     return 0;
23. }
```

```
1. #include <stdio.h>
2.
3. int main() {
4.     int giorno;
5.     printf("Inserisci un numero da 1 a 7 per il giorno della settimana: ");
6.     scanf("%d", &giorno);
7.
8.     switch (giorno) {
9.         case 1:
10.         case 7:
11.             printf("È weekend!\n");
12.             break;
13.         case 2:
14.         case 3:
15.         case 4:
16.         case 5:
17.         case 6:
18.             printf("È un giorno feriale.\n");
19.             break;
20.         default:
21.             printf("Valore non valido.\n");
22.     }
23.
24.     return 0;
25. }
```

# Elementi pratici - Condizioni - Switch

- **Nota: In C non si può usare direttamente una stringa nello switch (ad esempio, switch(nome) dove nome è un array di caratteri).**
- **Tuttavia, puoi simulare uno switch sulle stringhe usando una serie di if-else con strcmp.**
- **Come abbiamo già detto però si possono anche mescolare le strutture switch e if per usare entrambe le cose.**

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     char frutto[20];
6.     printf("Inserisci un frutto (mela, banana, pera): ");
7.     scanf("%19s", frutto);
8.
9.     if (strcmp(frutto, "mela") == 0) {
10.         printf("Hai scelto una mela.\n");
11.     } else if (strcmp(frutto, "banana") == 0) {
12.         printf("Hai scelto una banana.\n");
13.     } else if (strcmp(frutto, "pera") == 0) {
14.         printf("Hai scelto una pera.\n");
15.     } else {
16.         printf("Frutto non riconosciuto.\n");
17.     }
18.
19.     return 0;
20. }
```

# Elementi pratici - Condizioni - Switch

**Esercizio: Giorno della settimana con switch**

**Scrivi un programma in C che:**

- **Chiede all'utente di inserire parola del giorno tra lunedì e domenica e convertirlo in un numero intero da 1 a 7.**
- **Usa la struttura switch per stampare il nome del giorno della settimana corrispondente al numero inserito (1 = Lunedì, 2 = Martedì, ..., 7 = Domenica).**
- **Se il numero inserito non è compreso tra 1 e 7, stampa un messaggio di errore.**

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     char giorno[15];
6.     int numero = 0;
7.
8.     printf("Inserisci il giorno della settimana (in minuscolo): ");
9.     scanf("%s", giorno);
10.
11.    // Conversione parola in numero
12.    if (strcmp(giorno, "lunedì") == 0 || strcmp(giorno, "lunedì") == 0)
13.        numero = 1;
14.    else if (strcmp(giorno, "martedì") == 0 || strcmp(giorno, "martedì") == 0)
15.        numero = 2;
16.    else if (strcmp(giorno, "mercoledì") == 0 || strcmp(giorno, "mercoledì") == 0)
17.        numero = 3;
18.    else if (strcmp(giorno, "giovedì") == 0 || strcmp(giorno, "giovedì") == 0)
19.        numero = 4;
20.    else if (strcmp(giorno, "venerdì") == 0 || strcmp(giorno, "venerdì") == 0)
21.        numero = 5;
22.    else if (strcmp(giorno, "sabato") == 0)
23.        numero = 6;
24.    else if (strcmp(giorno, "domenica") == 0)
25.        numero = 7;
26.    else
27.        numero = 0;
28.
29.    // Switch sul numero ricavato
30.    switch (numero) {
31.        case 1:
32.            printf("1 = Lunedì\n");
33.            break;
34.        case 2:
35.            printf("2 = Martedì\n");
36.            break;
37.        case 3:
38.            printf("3 = Mercoledì\n");
39.            break;
40.        case 4:
41.            printf("4 = Giovedì\n");
42.            break;
43.        case 5:
44.            printf("5 = Venerdì\n");
45.            break;
46.        case 6:
47.            printf("6 = Sabato\n");
48.            break;
49.        case 7:
50.            printf("7 = Domenica\n");
51.            break;
52.        default:
53.            printf("Errore: giorno non valido!\n");
54.            break;
55.    }
56.
57.    return 0;
}
```

# Elementi pratici - Condizioni - Switch

**Scrivi un programma in C che:**

- Chiede all'utente di inserire il proprio ruolo (stringa: "studente", "docente", "ospite").
- Chiede all'utente di scegliere un'opzione di menu inserendo un numero intero da 1 a 3.
- Se il ruolo è studente e l'opzione è 1 , stampa "Accesso a materiali didattici".
- Se il ruolo è docente e l'opzione è 2, stampa "Accesso alla gestione corsi".
- Se il ruolo è ospite, stampa "Area informativa".
- In tutti gli altri casi, stampa "Opzione non disponibile".
- Dopo questi controlli, usa uno switch sull'opzione scelta per stampare:
  - Se 1: "Hai scelto: Visualizza"
  - Se 2: "Hai scelto: Modifica"
  - Se 3: "Hai scelto: Esci"
  - Altrimenti, "Opzione di menu non valida"
- Aggiungere una variabile segreta di tipo int che esegua uno dei comandi scelti sopra

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     char ruolo[20];
6.     int opzione;
7.
8.     // Input ruolo
9.     printf("Inserisci il tuo ruolo (studente/docente/ospite): ");
10.    scanf("%s", ruolo);
11.
12.    // Input opzione menu
13.    printf("Scegli un'opzione (1=Visualizza, 2=Modifica, 3=Esci, 99=Scelta segreta): ");
14.    scanf("%d", &opzione);
15.
16.    // Caso scelta segreta
17.    if (opzione == 99) {
18.        printf("Hai trovato la scelta segreta!\n");
19.    }
20.    // Studente: visualizza o modifica
21.    else if (strcmp(ruolo, "studente") == 0 && (opzione == 1 || opzione == 2)) {
22.        if (opzione == 1) {
23.            printf("Studente - Visualizzazione materiali\n");
24.        } else { // opzione == 2
25.            printf("Studente - Modifica note personali\n");
26.        }
27.    }
28.    // Docente: solo modifica
29.    else if (strcmp(ruolo, "docente") == 0 && opzione == 2) {
30.        printf("Docente - Modifica corsi\n");
31.    }
32.    // Ospite
33.    else if (strcmp(ruolo, "ospite") == 0) {
34.        printf("Area informativa\n");
35.    }
36.    // Altri casi
37.    else {
38.        printf("Opzione non disponibile\n");
39.    }
40.
41.    // Switch sull'opzione scelta
42.    switch (opzione) {
43.        case 1:
44.            printf("Hai scelto: Visualizza\n");
45.            break;
46.        case 2:
47.            printf("Hai scelto: Modifica\n");
48.            break;
49.        case 3:
50.            printf("Hai scelto: Esci\n");
51.            break;
52.        case 99:
53.            printf("Scelta segreta attivata!\n");
54.            break;
55.        default:
56.            printf("Opzione di menu non valida\n");
57.            break;
58.    }
59.
60.    return 0;
61.}
```

# Elementi pratici - Condizioni - El.Tern

- Cos'è l'operatore ternario:  
È una forma compatta di scrivere un'istruzione if-else. Ha la forma:

**condizione ? valore\_se\_vero : valore\_se\_falso**

- Come funziona nell'esempio:  
La condizione `numero % 2 == 0` controlla se il numero è divisibile per 2 (quindi pari). Se la condizione è vera, il risultato è la stringa "pari", altrimenti "dispari".

- Dove viene usato il risultato:  
Il valore scelto viene inserito direttamente all'interno della funzione printf, così viene stampato "pari" o "dispari" a seconda del caso.

- Vantaggio principale:  
Permette di scrivere codice più compatto e leggibile per scelte semplici, evitando la scrittura di un blocco if-else più lungo.

```
1. #include <stdio.h>
2.
3. int main() {
4.     int numero;
5.     printf("Inserisci un numero: ");
6.     scanf("%d", &numero);
7.
8.     // Operatore ternario per determinare se il numero è pari o dispari
9.     printf("%d è %s.\n", numero, (numero % 2 == 0) ? "pari" : "dispari");
10.
11.    return 0;
12.}
```

# **Elementi pratici - Cicli**

# Elementi pratici - Cicli

I cicli (o loop) in C sono costrutti di controllo del flusso che consentono di ripetere un blocco di codice fino al soddisfacimento di una condizione. Sono fondamentali per iterare su dati, ripetere operazioni fino a un determinato stato e automatizzare compiti ripetitivi.

In C esistono tre tipi principali di cicli:

- **while**
- **do ... while**
- **for**

Ognuno di essi si adatta a scenari diversi in base a quando e come si valuta la condizione, alla necessità di inizializzazione e aggiornamento delle variabili di iterazione, e alla leggibilità del codice.

# Elementi pratici - Cicli

**Condizione di continuazione:** espressione booleana che decide se ripetere o uscire dal ciclo.

**Corpo del ciclo:** blocco di codice { ... } eseguito ad ogni iterazione.

**Rischio di loop infinito:** assicurarsi che la condizione diventi mai falsa in un tempo finito.

**Istruzioni di controllo:**

- **break:** esce immediatamente dal ciclo.
- **continue:** salta il resto del corpo e passa all'iterazione successiva.

**Scope:** variabili di iterazione dichiarate nel ciclo hanno scope locale al ciclo.

# Elementi pratici - Cicli - while

**Il ciclo while valuta la condizione prima di ogni iterazione. Se la condizione è vera, esegue il corpo e poi riesegue la verifica.**

**È ideale quando non si conosce a priori il numero di iterazioni e la ripetizione dipende da un evento o da un valore che evolve durante l'esecuzione.**

1. `while (condizione) { ... }`

- Condizione valutata all'inizio (pre-test).
- Possibile esecuzione zero volte se la condizione è falsa fin da subito.
- Controllo fine-grained: si può modificare la variabile di condizione all'interno del corpo.

# Elementi pratici - Cicli - while

**int n = 5;**

- **inizializzazione del contatore.**

**while (n > 0)**

- **pre-controllo:** se n è maggiore di zero, entra nel corpo.

**n--**

- **riduce n, garantendo che la condizione diventi falsa e si esca.**

```
1. #include <stdio.h>
2.
3. int main(void) {
4.     int n = 5;      // 1) Contatore di iterazioni
5.
6.     // 2) while: ripete finché n > 0
7.     while (n > 0) {
8.         printf("Valore di n: %d\n", n);
9.         n--;        // 3) Decremento del contatore
10.    }
11.
12.    printf("Fine ciclo while\n");
13.    return 0;
14.}
15.
```

# Elementi pratici - Cicli - while

- Il ciclo continua finché la variabile booleana trovato rimane falsa (0).
- Quando l'utente indovina il numero, la variabile viene impostata a vero (1) e il ciclo termina.

```
1. #include <stdio.h>
2.
3. int main() {
4.     int numero, trovato = 0; // 0 = falso, 1 = vero
5.
6.     while (!trovato) {
7.         printf("Indovina il numero segreto (tra 1 e 5): ");
8.         scanf("%d", &numero);
9.
10.        if (numero == 3) {
11.            printf("Hai indovinato!\n");
12.            trovato = 1; // esce dal ciclo
13.        } else {
14.            printf("Ritenta.\n");
15.        }
16.    }
17.    return 0;
18.}
```

# Elementi pratici - Cicli - while

- Il **ciclo chiede all'utente di inserire un carattere.**
- Finché non viene inserito '**s**', il **ciclo continua**. Quando l'utente inserisce '**s**', il **ciclo si interrompe**.

```
1. #include <stdio.h>
2.
3. int main() {
4.     char risposta = 'n';
5.
6.     while (risposta != 's') {
7.         printf("Vuoi uscire dal programma? (s/n): ");
8.         scanf(" %c", &risposta);
9.         // spazio prima di %c per pulire il buffer
10.
11.        if (risposta != 's') {
12.            printf("Programma ancora in esecuzione...\n");
13.        }
14.    }
15.    printf("Programma terminato.\n");
16.    return 0;
17.}
```

# Elementi pratici - Cicli - while

- Si include `<stdbool.h>` per poter dichiarare variabili di tipo `bool` (booleans) che possono essere `true` o `false`.
- La variabile `continua` viene inizializzata a `true` per entrare nel ciclo.
- All'interno del ciclo, viene chiesto all'utente di inserire un numero.
- Se l'utente inserisce 0, la variabile booleana viene impostata a `false` e il ciclo si interrompe.
- In caso contrario, il ciclo continua e il numero inserito viene stampato.
- Al termine, viene stampato un messaggio di fine ciclo.

```
1. #include <stdio.h>
2. #include <stdbool.h> // Necessario per usare il tipo bool
3.
4. int main() {
5.     int numero;
6.     bool continua = true; // Variabile booleana di controllo
7.
8.     while (continua) {
9.         printf("Inserisci un numero (0 per terminare): ");
10.        scanf("%d", &numero);
11.
12.        if (numero == 0) {
13.            continua = false; // Interrompe il ciclo
14.        } else {
15.            printf("Hai inserito il numero %d\n", numero);
16.        }
17.    }
18.    printf("Ciclo terminato!\n");
19.    return 0;
20.}
21.
22.
23.
```

# Elementi pratici - Cicli - while

## Esercizio: Somma di numeri positivi

Scrivi un programma in C che:

- Chiede all'utente di inserire ripetutamente numeri interi.
- Somma solo i numeri positivi inseriti.
- Quando l'utente inserisce il valore zero, il programma termina e stampa la somma totale dei numeri positivi inseriti.

```
1. #include <stdio.h>
2.
3. int main() {
4.     int numero, somma = 0;
5.
6.     printf("Inserisci numeri positivi da sommare (0 per
terminare):\n");
7.     scanf("%d", &numero);
8.
9.     while (numero != 0) {
10.         if (numero > 0) {
11.             somma += numero;
12.         }
13.         scanf("%d", &numero);
14.     }
15.
16.     printf("La somma dei numeri positivi inseriti è: %d\n",
somma);
17.
18.     return 0;
19. }
```

# Elementi pratici - Cicli - do while

**Il ciclo do ... while esegue prima il corpo del ciclo e poi verifica la condizione.**

**Questo garantisce che il corpo venga eseguito almeno una volta, anche se la condizione iniziale è falsa. È utile quando è necessario eseguire un'operazione iniziale (ad es. un menu) prima di controllarne l'esito.**

1. do { ... } while (condizione);

- Condizione valutata alla fine (post-test).
- Esecuzione minima di una iterazione.
- Sintassi richiede punto e virgola dopo la parentesi di chiusura.

# Elementi pratici - Cicli - do while

## Corpo del do

- viene sempre eseguito almeno una volta per mostrare il menu.

**while (scelta != 0);**

- controlla l'uscita dopo l'esecuzione.

```
1. #include <stdio.h>
2.
3. int main(void) {
4.     int scelta;
5.
6.     // 1) Menu che si ripete almeno una volta
7.     do {
8.         printf("Menu:\n");
9.         printf("1. Saluta\n");
10.        printf("0. Esci\n");
11.        printf("Scelta: ");
12.        scanf("%d", &scelta);
13.
14.        if (scelta == 1) {
15.            printf("Ciao!\n");
16.        }
17.    } while (scelta != 0);
18.    // 2) Continua finché l'utente non inserisce 0
19.
20.    printf("Fine programma\n");
21.    return 0;
22.}
```

# Elementi pratici - Cicli - do while

**Esercizio: Validazione e somma con while e do...while**

**Scrivi un programma in C che:**

- **Con un ciclo do...while, chiedi all'utente se vuole continuare (rispondendo 's' per sì o 'n' per no). Il ciclo termina solo se l'utente inserisce 'n'.**
- **All'interno del ciclo, ogni volta chiedi un numero intero e conta quanti numeri pari vengono inseriti.**
- **Alla fine, stampa quanti numeri pari sono stati inseriti.**
- **Usa anche un ciclo while per validare la risposta alla domanda ("Vuoi continuare?"): se l'utente inserisce qualcosa di diverso da 's' o 'n', richiedi nuovamente la risposta finché non è valida.**

```
1. #include <stdio.h>
2.
3. int main() {
4.     char risposta;
5.     int numero, conta_pari = 0;
6.
7.     do {
8.         // Richiesta numero
9.         printf("Inserisci un numero intero: ");
10.        scanf("%d", &numero);
11.
12.        // Conta se pari
13.        if (numero % 2 == 0) {
14.            conta_pari++;
15.        }
16.
17.        // Validazione risposta con while
18.        printf("Vuoi continuare? (s/n): ");
19.        scanf(" %c", &risposta); // spazio prima di %c per consumare eventuali \n
20.        while (risposta != 's' && risposta != 'n') {
21.            printf("Risposta non valida. Vuoi continuare? (s/n): ");
22.            scanf(" %c", &risposta);
23.        }
24.
25.    } while (risposta == 's');
26.
27.    printf("Hai inserito %d numeri pari.\n", conta_pari);
28.
29.    return 0;
30.}
```

# Elementi pratici - Cicli - for

**Il ciclo for è il più compatto per iterazioni a numero noto di passi.**

**Si compone di tre parti: inizializzazione, condizione e aggiornamento, tutte nella stessa riga, rendendo esplicito il comportamento del contatore. È ideale per attraversare array, contare da un valore a un altro e cicli con passo costante.**

1. Sintassi: `for (inizializzazione; condizione; aggiornamento) { ... }`

- Inizializzazione: dichiarazione o assegnazione di variabili di controllo.
- Condizione: test di continuazione (pre-test).
- Aggiornamento: eseguito alla fine di ogni iterazione.
- Scope: variabile di controllo può avere scope locale al for.

# Elementi pratici - Cicli - for

**for (int i = 1; i <= 10; i++)**

- **inizializza i=1, verifica i<=10, poi i++ ad ogni iterazione.**

## Corpo

- **stampa i e uno spazio.**

**for (int j = 10; j > 0; j -= 2)**

- **iterazione con passo -2, dimostrando flessibilità di aggiornamento.**

```
1. #include <stdio.h>
2.
3. int main(void) {
4.     // 1) Stampa i primi 10 numeri naturali
5.     for (int i = 1; i <= 10; i++) {
6.         printf("%d ", i); // 2) Corpo del ciclo
7.     }
8.     printf("\n");
9.
10.    // 3) Iterazione con passo diverso
11.    for (int j = 10; j > 0; j -= 2) {
12.        printf("%d ", j); // stampa 10, 8, 6, 4, 2
13.    }
14.    printf("\n");
15.
16.    return 0;
17.}
```

# Elementi pratici - Cicli - for

**Esercizio: Inserimento controllato e conteggio dei numeri dispari**

**Scrivi un programma in C che:**

- Chiede all'utente quanti numeri vuole inserire (n, input intero positivo).
- Usa un ciclo for per richiedere l'inserimento di n numeri interi.
- Per ogni numero, usa un ciclo while per validare che il numero sia compreso tra 10 e 100 (estremi inclusi).
- Se il numero inserito non è valido, chiedilo di nuovo.
- Se il numero è dispari (if), aumenta un contatore.
- Alla fine, stampa quanti numeri dispari sono stati inseriti.

```
1. #include <stdio.h>
2.
3. int main() {
4.     int n, i, numero, conta_dispari = 0;
5.
6.     // 1. Input numero di valori da inserire
7.     printf("Quanti numeri vuoi inserire? ");
8.     scanf("%d", &n);
9.
10.    // 2. Ciclo for per chiedere n numeri
11.    for (i = 0; i < n; i++) {
12.        // 3. Validazione con while
13.        printf("Inserisci un numero tra 10 e 100: ");
14.        scanf("%d", &numero);
15.
16.        while (numero < 10 || numero > 100) {
17.            printf("Numero non valido! Inserisci un numero tra 10 e 100: ");
18.            scanf("%d", &numero);
19.        }
20.
21.        // 4. Controllo se dispari
22.        if (numero % 2 != 0) {
23.            conta_dispari++;
24.        }
25.    }
26.
27.    // 5. Stampa risultato
28.    printf("Hai inserito %d numeri dispari.\n", conta_dispari);
29.
30.    return 0;
31.}
```

# Elementi pratici - Arrays

# Elementi pratici - Arrays

**Un array in C è una collezione contigua di elementi dello stesso tipo, indicizzati a partire da zero.**

**Viene allocato come un blocco continuo di memoria, garantendo accesso in tempo costante a ciascun elemento tramite l'indice.**

**Gli array sono utili per gestire sequenze di dati (numeri, caratteri, strutture) quando si conosce il numero massimo di elementi a compile-time (array statici) o si desidera un buffer di dimensione fissa.**

**L'uso corretto degli array richiede attenzione al bounds checking, poiché in C non c'è verifica automatica degli indici e un accesso fuori dall'array provoca errore.**

# Elementi pratici - Arrays

## Dichiarazione

- Sintassi: **tipo nome[tam];** (es. **int v[5];**).

## Indice

- Va da **0** a **tam-1**.

## Inizializzazione

- Statica: **int v[3] = {1, 2, 3};**
- Parziale: **int v[5] = {0};** inizializza tutti a 0.

## Accesso

- **Lettura/scrittura con v[i].**

## Dimensione

- **sizeof(v) / sizeof(v[0])** calcola il numero di elementi (solo per array, non per puntatori).

# Elementi pratici - Arrays

```
int v[5] = {10, 20, 30, 40, 50};
```

- Crea un array di 5 int e assegna i valori iniziali.

## Calcolo di n

- **sizeof(v)** è la dimensione in byte dell'array; dividendo per **sizeof(v[0])** otteniamo il numero di elementi.

## Lettura con v[i]

- Cicliamo da 0 a n-1; v[i] accede all'i-esimo elemento.

## Scrittura con v[2] = 99;

- Modifica il valore in posizione 2 sfruttando l'indice.

## Copia con memcpy

- Copia il blocco di memoria dell'array v in copia, dimostrando che gli array sono blocchi contigui.

```
1. #include <stdio.h>
2.
3. int main(void){
4. // 1) Dichiarazione e inizializzazione di un array di interi
5. int v[5] = {10, 20, 30, 40, 50};
6.
7. // 2) Calcolo della dimensione (numero di elementi)
8. size_t n = sizeof(v) / sizeof(v[0]);
9.
10. // 3) Accesso e stampa di tutti gli elementi
11. printf("Contenuto dell'array:\n");
12. for (size_t i = 0; i < n; i++) {
13.     printf("v[%zu] = %d\n", i, v[i]);
14. }
15.
16. // 4) Modifica di un elemento
17. v[2] = 99; // cambia il terzo elemento (precedentemente 30) in 99
18.
19. // 5) Uso di memcpy per copiare l'array in uno nuovo
20. int copia[5];
21. memcpy(copia, v, sizeof(v)); // include <string.h> per memcpy
22.
23. printf("\nArray dopo modifica e copia:\n");
24. for (size_t i = 0; i < n; i++) {
25.     printf("copia[%zu] = %d\n", i, copia[i]);
26. }
27.
28. return 0;
29.}
```

# Elementi pratici - Arrays

**Esercizio: Somma degli elementi di array**

**Scrivi un programma in C che:**

- **Chiede all'utente quanti numeri desidera inserire (massimo 100).**
- **Usa un array per memorizzare questi numeri interi.**
- **Chiede all'utente di inserire tutti i numeri, uno alla volta.**
- **Calcola e stampa la somma di tutti gli elementi dell'array.**

```
1. #include <stdio.h>
2.
3. int main() {
4.     int numeri[100], n, i, somma = 0;
5.
6.     // 1. Input quantità
7.     printf("Quanti numeri vuoi inserire? (max 100): ");
8.     scanf("%d", &n);
9.
10.    // Controllo limite massimo
11.    if (n > 100 || n <= 0) {
12.        printf("Numero non valido!\n");
13.        return 1;
14.    }
15.
16.    // 2. Inserimento numeri nell'array
17.    for (i = 0; i < n; i++) {
18.        printf("Inserisci il numero %d: ", i + 1);
19.        scanf("%d", &numeri[i]);
20.        somma += numeri[i]; // Calcolo somma
21.    }
22.
23.    // 3. Output somma
24.    printf("La somma degli elementi è: %d\n", somma);
25.
26.    return 0;
27.}
```

# Elementi pratici - Arrays

**Esercizio: Somma degli elementi di array**

**Scrivi un programma in C che:**

- **Chiede all'utente quanti numeri vuole inserire (massimo 50).**
- **Usa un array per memorizzare questi numeri interi.**
- **Chiede all'utente di inserire i numeri uno alla volta.**
- **Chiede all'utente di inserire un valore da cercare nell'array.**
- **Conta quante volte il valore scelto appare nell'array e stampa il risultato.**

```
1. #include <stdio.h>
2.
3. int main() {
4.     int numeri[50], n, i, valore, contatore = 0;
5.
6.     // 1. Input quantità
7.     printf("Quanti numeri vuoi inserire? (max 50): ");
8.     scanf("%d", &n);
9.
10.    // Controllo validità quantità
11.    if (n > 50 || n <= 0) {
12.        printf("Numero non valido!\n");
13.        return 1;
14.    }
15.
16.    // 2. Inserimento valori nell'array
17.    for (i = 0; i < n; i++) {
18.        printf("Inserisci il numero %d: ", i + 1);
19.        scanf("%d", &numeri[i]);
20.    }
21.
22.    // 3. Scelta valore da cercare
23.    printf("Inserisci il valore da cercare: ");
24.    scanf("%d", &valore);
25.
26.    // 4. Conta quante volte appare il valore
27.    for (i = 0; i < n; i++) {
28.        if (numeri[i] == valore) {
29.            contatore++;
30.        }
31.    }
32.
33.    // 5. Output risultato
34.    printf("Il valore %d appare %d volte nell'array.\n", valore, contatore);
35.
36.    return 0;
37.}
```

# Elementi pratici - Arrays

- Dichiarazione dell'array:

Viene dichiarato un array bidimensionale char nomi[3][20], che può contenere 3 stringhe, ognuna lunga massimo 19 caratteri (più il carattere terminatore \0).

- Inserimento:

Con un ciclo for, vengono richiesti e inseriti i nomi nell'array tramite scanf.

- Stampa:

Un secondo ciclo for stampa a video tutti i nomi memorizzati nell'array.

- Nota:

In C le "stringhe" sono in realtà array di caratteri.

```
1. #include <stdio.h>
2.
3. int main() {
4.     // Array di 3 stringhe, ognuna lunga massimo 19 caratteri (+1 per '\0')
5.     char nomi[3][20];
6.
7.     // Inserimento dei nomi
8.     for (int i = 0; i < 3; i++) {
9.         printf("Inserisci il nome %d: ", i + 1);
10.        scanf("%19s", nomi[i]);
11.        // Legge una stringa (nome) e la salva nell'array
12.    }
13.
14.    // Stampa dei nomi inseriti
15.    printf("Hai inserito questi nomi:\n");
16.    for (int i = 0; i < 3; i++) {
17.        printf("%s\n", nomi[i]);
18.    }
19.
20.    return 0;
21.}
```

# Elementi pratici - Arrays - Inizializzatori

Gli **inizializzatori** in C (**initializer**) permettono di assegnare valori alle variabili (e in particolare agli array e alle strutture) nel momento stesso della loro dichiarazione.

Questo approccio garantisce che ogni elemento venga definito in modo esplicito, evitando valori “spazzatura” e rendendo il codice più conciso e leggibile.

Per gli array e le strutture, gli inizializzatori possono sfruttare liste di valori racchiuse tra {}: l’ordine degli elementi corrisponde a quello dei campi (o degli indici) e, se non vengono specificati tutti i valori, i restanti vengono automaticamente inizializzati a zero (per i tipi numerici) o al carattere '\0' (per le stringhe).

# Elementi pratici - Arrays - Inizializzatori

## Dichiarazione con inizializzatore

```
1. int x = 5;
```

## Inizializzazione di array

```
1. int a[4] = {1, 2, 3, 4};
```

## Inizializzazione parziale

```
1. int b[5] = {0};           // b = {0,0,0,0,0}
2. int c[5] = {1,2};         // c = {1,2,0,0,0}
```

## Omissione del taglio

```
1. int d[] = {10,20,30}; // dimensione dedotta = 3
```

## Inizializzazione di strutture

```
1. struct Punto { int x, y; };
2. struct Punto p = { .x = 1, .y = 2 };
```

# Elementi pratici - Arrays - Inizializzatori

**int a = 10;, float f = 3.5f;**

- **inizializzatori diretti per variabili scalari.**

**int arr1[5] = {...}**

- **tutti i valori specificati.**

**int arr2[5] = {1,2}**

- **i rimanenti elementi sono automaticamente zero.**

**char msg[] = "Ciao";**

- C deduce la lunghezza dell'array (5: 4 lettere + '\0').

## Designated initializers:

- **associa esplicitamente valori a membri o indici, indipendentemente dall'ordine.**

## Zero initialization implicita:

- **qualsiasi elemento non inizializzato viene posto a 0 o '\0'.**

```
1. #include <stdio.h>
2.
3. struct Punto {
4.     int x;
5.     int y;
6. };
7.
8. int main(void) {
9.     // 1) Inizializzazione di variabili semplici
10.    int a = 10;           // a = 10
11.    float f = 3.5f;      // f = 3.5
12.
13.    // 2) Array completamente inizializzato
14.    int arr1[5] = {1, 2, 3, 4, 5};
15.
16.    // 3) Array parzialmente inizializzato
17.    int arr2[5] = {1, 2};   // arr2 = {1,2,0,0,0}
18.
19.    // 4) Dimensione dedotta
20.    char msg[] = "Ciao";   // msg ha 5+1=5 caratteri compreso '\0'
21.
22.    // 5) Struttura con designated initializers
23.    struct Punto p = { .y = 20, .x = 10 }; // ordine esplicito dei campi
24.
25.    // 6) Stampa dei valori
26.    printf("a=%d, f=%.1f\n", a, f);
27.    printf("arr1: ");
28.    for(int i=0; i<5; i++) printf("%d ", arr1[i]);
29.    printf("\narr2: ");
30.    for(int i=0; i<5; i++) printf("%d ", arr2[i]);
31.    printf("\nmsg: %s\n", msg);
32.    printf("Punto p: (%d, %d)\n", p.x, p.y);
33.
34.    return 0;
35. }
```

# Elementi pratici - Arrays - puntatore

Un puntatore in C è una variabile che memorizza l'indirizzo di memoria di un'altra variabile.

Agendo sul puntatore, si può leggere o scrivere il valore a cui punta mediante l'operazione di dereferenziazione.

I puntatori sono essenziali per l'allocazione dinamica, la gestione di strutture dati complesse (liste, alberi), e per passare array e strutture alle funzioni senza copiare grandi blocchi di memoria.

L'operatore & restituisce l'indirizzo di una variabile, mentre l'operatore \* (prefisso) dereferenzia l'indirizzo contenuto in un puntatore.

# Elementi pratici - Arrays - puntatore

L'operatore unario & in C restituisce l'indirizzo di memoria della variabile a cui viene applicato.

È fondamentale per ottenere un puntatore che "punta" a quella variabile, permettendo di manipolare il suo contenuto indirettamente.

Usato in combinazione con l'operatore di dereferenziazione \*, consente di passare valori per riferimento, gestire strutture dati dinamiche e implementare funzioni che modificano variabili definite nel chiamante.

# Elementi pratici - Arrays

Il tipo **void** in C rappresenta l'assenza di valore o di tipo, e viene utilizzato in tre principali contesti:

- **Funzioni che non restituiscono nulla:** dichiarate come **void nome\_funzione(...)**, indicano che non hanno un valore di ritorno.
- **Parametri di funzione vuoti:** **int main(void)** esplicita che la funzione **main** non accetta argomenti.
- **Puntatori generici:** **void \*** è un puntatore “polimorfo” che può riferirsi a qualunque tipo di dato, ma va sempre convertito (cast) al tipo corretto prima della dereferenziazione.

# Elementi pratici - Arrays - puntatore

L'operatore unario `*`, quando applicato a un puntatore, effettua la dereferenziazione, ovvero accede al contenuto della locazione di memoria il cui indirizzo è memorizzato nel puntatore.

Permette di leggere o scrivere il valore della variabile "puntata".

È complementare all'operatore `&` e consente la manipolazione indiretta dei dati, base per strutture dinamiche, array e passaggio di parametri per riferimento.

- Uso: `*ptr` → valore di tipo T se ptr è di tipo T \*.

# Elementi pratici - Arrays - puntatore

## Dichiarazione

1. **int \*p;** // p è puntatore a int

## Assegnazione di indirizzo

1. **int x;**  
2. **p = &x;** // p riceve l'indirizzo di x

## Dereferenziazione

1. **\*p = 5;** // scrive 5 in x  
2. **int y = x;** // legge il valore di x in y

# Elementi pratici - Arrays - Puntatore

**int \*p = &x;**

- p memorizza l'indirizzo di x.

**\*p = 100;**

- scrive attraverso il puntatore, modificando x.

## Allocazione dinamica

- malloc restituisce un puntatore; arr[i] e \*(arr+i) sono equivalenti.

## Puntatore a puntatore

- pp punta a p; dereferenziazione doppia modifica il dato originario.

**free(arr);**

- fondamentale per evitare memory leak quando si usa heap.

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. int main(void) {
5.     // 1) Puntatore a intero
6.     int x = 42;
7.     int *p = &x;      // p punta a x
8.
9.     printf("Valore di x tramite p: %d\n", *p); // dereferenziazione
10.
11.    // 2) Modifica via puntatore
12.    *p = 100;
13.    printf("x modificato: %d\n", x);
14.
15.    // 3) Array e pointer arithmetic
16.    int *arr = malloc(5 * sizeof *arr); // allocazione dinamica
17.    if (!arr) return 1;
18.    for (int i = 0; i < 5; i++) {
19.        arr[i] = i * 10; // equivalente a *(arr + i) = i*10
20.    }
21.    printf("Array dinamico: ");
22.    for (int i = 0; i < 5; i++) {
23.        printf("%d ", *(arr + i));
24.    }
25.    printf("\n");
26.
27.    // 4) Puntatore a puntatore
28.    int **pp = &p;
29.    **pp = 200;      // modifica indiretta di x
30.    printf("x dopo **pp = %d\n", x);
31.
32.    free(arr);      // rilascio memoria
33.    return 0;
34.}
```

# Elementi pratici - Arrays - puntatore

## Uso:

- **&variabile → tipo T \*** se variabile è di tipo T.

## Puntatori:

- assegna a un puntatore l'indirizzo di una variabile.

## Passaggio per riferimento:

- permette a una funzione di modificare la variabile originale.

## Costante:

- applicabile solo a variabili (l'uso su costanti letterali è invalido).

## Allineamento:

- l'indirizzo restituito rispetta l'architettura (alignment).

## Null vs Indirizzo:

- **&x non è mai NULL;** per un puntatore “vuoto” si usa NULL.

# Elementi pratici - Arrays - Puntatore

`int *p = &x;`

- `&x` restituisce l'indirizzo di `x` (tipo `int *`), assegnato a `p`.

`incrementa(&x);`

- Passa l'indirizzo di `x` alla funzione, che riceve un puntatore e modifica `x` direttamente.

`*p = *p + 5;`

- Usa il puntatore `p` (contenente `&x`) per leggere e riscrivere il valore di `x`.

```
1. #include <stdio.h>
2.
3. void incrementa(int *ptr) {
4.     // Modifica il valore della variabile passata per riferimento
5.     (*ptr)++;
6. }
7.
8. int main(void) {
9.     int x = 10;
10.
11.    // 1) Ottengo l'indirizzo di x con &
12.    int *p = &x;
13.
14.    printf("Valore di x prima: %d\n", x);
15.
16.    // 2) Passaggio dell'indirizzo di x alla funzione
17.    incrementa(&x);
18.
19.    printf("Valore di x dopo incrementa(&x): %d\n", x);
20.
21.    // 3) Uso di p (che già contiene &x)
22.    *p = *p + 5;
23.
24.    printf("Valore di x dopo *p = *p + 5: %d\n", x);
25.
26.    return 0;
27.}
```

# Elementi pratici - Arrays - Puntatore

Esercizio: Scambio, somma e massimo con puntatori

Scrivi un programma in C che:

- Chiede all'utente di inserire due numeri interi.
- Usa due puntatori per scambiare i valori dei due numeri.
- Stampa a schermo i numeri prima e dopo lo scambio.
- Calcola la somma dei due numeri usando i puntatori e stampa il risultato.
- Determina quale dei due numeri (dopo lo scambio) è il maggiore, sempre usando i puntatori, e stampa il risultato.

```
1. #include <stdio.h>
2.
3. int main() {
4.     int a, b, temp;
5.     int *pa, *pb;
6.
7.     // 1. Inserimento valori
8.     printf("Inserisci il primo numero: ");
9.     scanf("%d", &a);
10.
11.    printf("Inserisci il secondo numero: ");
12.    scanf("%d", &b);
13.
14.    // 2. Puntatori
15.    pa = &a;
16.    pb = &b;
17.
18.    // 3. Stampa valori prima dello scambio
19.    printf("Prima dello scambio: a = %d, b = %d\n", a, b);
20.
21.    // Scambio valori tramite puntatori
22.    temp = *pa;
23.    *pa = *pb;
24.    *pb = temp;
25.
26.    // 4. Stampa valori dopo lo scambio
27.    printf("Dopo lo scambio: a = %d, b = %d\n", a, b);
28.
29.    // 5. Calcola la somma usando i puntatori
30.    int somma = *pa + *pb;
31.    printf("La somma dei due numeri (dopo lo scambio) è: %d\n", somma);
```

# **Elementi pratici - Funzioni**

# Elementi pratici - Funzioni

Le funzioni in C sono blocchi di codice autonomi che svolgono un compito specifico e possono essere richiamate (invocate) da più punti di un programma, favorendo la modularità, la riusabilità e la leggibilità.

Una funzione ha una dichiarazione (prototipo) che specifica il suo nome, il tipo di ritorno e il tipo e ordine dei parametri, seguita dalla sua definizione, che contiene il corpo vero e proprio.

Il flusso di esecuzione passa dal chiamante al corpo della funzione e ritorna al chiamante con un valore (se il tipo di ritorno non è void).

Le funzioni permettono di astrarre dettagli implementativi, suddividere il programma in unità logiche e facilitare il debug e la manutenzione.

# Elementi pratici - Funzioni

In C, una **funzione** viene dichiarata (o prototipata) per informare il compilatore del suo nome, del tipo di ritorno e del tipo/numero dei parametri, prima del suo effettivo utilizzo.

La **definizione** fornisce il corpo della funzione, ossia l'**implementazione** vera e propria.

L'**invocazione** avviene quando, all'interno di un'altra funzione (ad es. main o un'altra funzione), si chiama il nome della funzione passandole gli argomenti appropriati; a quel punto il flusso di esecuzione salta al corpo della funzione e, al termine, ritorna al punto di chiamata restituendo un valore (se previsto).

```
1. tipo_di_ritorno nome(parametri) {  
2. // corpo  
3. return valore; // se tipo_di_ritorno ≠ void  
4. }
```

# Elementi pratici - Funzioni

## Prototipo

- `int somma(int a, int b);` dice al compilatore che esiste una funzione con quel nome e firma.

## Invocazione

- In `main`, `somma(x, y)` chiama la funzione, passando i valori di `x` e `y`;
- Il risultato viene assegnato a `risultato`.

## Definizione

- `int somma(int a, int b) { ... }` contiene il corpo: calcola  $a + b$  e la restituisce.

```
1. #include <stdio.h>
2.
3. // 1) Dichiarazione (prototipo) della funzione prima di main
4. int somma(int a, int b);
5.
6. int main(void) {
7.     int x = 5, y = 7;
8.
9.     // 2) Invocazione della funzione: passo x e y come
   argomenti
10.    int risultato = somma(x, y);
11.
12.    printf("La somma di %d e %d è %d\n", x, y, risultato);
13.    return 0;
14.}
15.
16. // 3) Definizione della funzione: implementazione del
   comportamento
17. int somma(int a, int b) {
18.     int s = a + b; // calcolo della somma
19.     return s;      // restituisco il risultato
20.}
21.
```

# Elementi pratici - Funzioni

## Parametri

- **Formali:** nomi e tipi definiti nella firma;
- **Attuali (argomenti):** valori passati alla chiamata.

## Passaggio per valore

- In C tutti i parametri vengono passati per copia; modifiche interne non impattano il chiamante (a meno di puntatori).

## Tipo di ritorno

- **void** se non deve restituire nulla; altrimenti un tipo scalare o struct.

## Ambito e durata

- **Variabili locali hanno scope limitato al corpo della funzione e durata automatica (stack).**

## Ricorsione

- Le funzioni possono chiamare sé stesse per **risolvere problemi dividendo il compito in sottoproblemi.**

# Elementi pratici - Funzioni

## Prototipo

- `int massimo(int a, int b);` informa il compilatore dell'esistenza della funzione e dei tipi coinvolti.

## Definizione

- `int massimo(int a, int b) { ... }` contiene la logica: un `if...else` che restituisce il maggiore dei due parametri.

## Passaggio per valore

- In `main`, `massimo(x, y)` copia `x` in `a` e `y` in `b`; modifiche a o `b` non alterano `x` e `y`.

## Return

- Il valore restituito (`a` o `b`) è passato indietro alla chiamata e memorizzato in `m`.

```
1. #include <stdio.h>
2.
3. // Prototipo: dichiara la funzione prima di main
4. int massimo(int a, int b);
5.
6. // Definizione: implementa il comportamento
7. int massimo(int a, int b) {
8.     if (a > b) {
9.         return a; // restituisce il valore più grande
10.    } else {
11.        return b;
12.    }
13.}
14.
15. int main(void) {
16.     int x = 7, y = 12;
17.
18.     // Chiamata: i valori di x e y vengono copiati in a e b
19.     int m = massimo(x, y);
20.
21.     printf("Il massimo tra %d e %d è %d\n", x, y, m);
22.
23.     return 0;
24.}
```

# Elementi pratici - Funzioni - Main

La funzione `main` in C può essere definita in due forme standard, per ricevere argomenti dalla linea di comando.

Il primo parametro, `argc` (argument count), è un intero che indica il numero totale di stringhe passate, compreso il nome del programma; il secondo, `argv` (argument vector), è un array di puntatori a `char` che contiene ciascun argomento come stringa terminata da `\0`.

In questo modo il programma può elaborare input esterno senza ricompilazione, interpretando flag, file di input, opzioni e altri parametri.

# Elementi pratici - Funzioni - Main

## Prototipi accettati

- `int main(int argc, char *argv[])`
- `int main(int argc, char **argv)`

## argc

- $\geq 1$ : `argv[0]` è sempre il nome (o percorso) del programma.
- Il numero di parametri reali è `argc - 1`.

## argv

- Array di `argc` puntatori; ultimo elemento `argv[argc]` è `NULL` per convenzione.

## Parsing

- Spesso si confrontano `argv[i]` con stringhe (`strcmp`) per riconoscere opzioni (`-h`, `--file=...`).

# Elementi pratici - Funzioni - Main

## argv[0]

- Contiene il nome o percorso del programma, utile per messaggi di errore o help.

## argc == 1

- Quando non ci sono altri argomenti, si comunica all'utente.

## Loop di parsing

- Ciclo su argv[1]...argv[argc-1] per stampare e riconoscere flag con strcmp.

## Return value

- Restituisce 0 al sistema operativo per indicare che tutto è andato a buon fine.

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main(int argc, char *argv[]) {
5.     // 1) Stampo nome del programma
6.     printf("Programma: %s\n", argv[0]);
7.
8.     // 2) Controllo se sono stati passati argomenti
9.     if (argc == 1) {
10.         printf("Nessun argomento specificato.\n");
11.     } else {
12.         printf("Argomenti (argc = %d):\n", argc);
13.         for (int i = 1; i < argc; i++) {
14.             printf(" argv[%d] = %s\n", i, argv[i]);
15.         }
16.     }
17.
18.     // 3) Esempio di riconoscimento di un flag
19.     for (int i = 1; i < argc; i++) {
20.         if (strcmp(argv[i], "-h") == 0 || strcmp(argv[i], "--help") == 0) {
21.             printf("Uso: %s [opzioni]\n", argv[0]);
22.             printf(" -h, --help Visualizza questo aiuto\n");
23.         }
24.     }
25.
26.     return 0; // 4) Codice di uscita 0 = successo
27. }
```

# Elementi pratici - Funzioni - Memoria

In C, la memoria dinamica viene gestita manualmente dal programmatore attraverso l'heap, uno spazio di memoria distinto dallo stack delle variabili locali.

Le funzioni `malloc()`, `calloc()` e `realloc()` (definite in `<stdlib.h>`) permettono di allocare blocchi di memoria di dimensione scelta a runtime, restituendo un puntatore al primo byte dell'area ottenuta.

È fondamentale liberare la memoria non più necessaria con `free()`, altrimenti si generano memory leak, ossia perdite di memoria che possono portare a esaurimento delle risorse e crash.

# Elementi pratici - Funzioni - Memoria

- `#include <stdlib.h>`: obbligatorio per `malloc()`, `calloc()`, `realloc()`, `free()`.
- `malloc(size_t size)`: alloca `size` byte, restituisce `void *` o `NULL` se fallisce.
- `calloc(size_t nmemb, size_t size)`: alloca memoria per `nmemb` elementi di `size` byte e azzera tutti i byte a 0.
- `realloc(void *ptr, size_t new_size)`: cambia la dimensione del blocco puntato da `ptr`, spostandolo se necessario.
- Controllo di `NULL`: sempre verificare che il puntatore restituito non sia `NULL` prima di usarlo.
- `free(void *ptr)`: restituisce al sistema la memoria precedentemente allocata; passare esattamente lo stesso puntatore.
- Size calculation: usare `sizeof(tipo)` per calcolare correttamente i byte da allocare.
- Memory leak: evitare di perdere il puntatore a memoria heap non liberata.
- Dangling pointer: dopo `free()`, impostare il puntatore a `NULL` per prevenire accessi

# Elementi pratici - Funzioni - Memoria

- `malloc(n * sizeof *arr)`
  - Alloca spazio per n interi; `sizeof *arr` equivale a `sizeof(int)`.
- Controllo di NULL
  - Se l'allocazione fallisce, `malloc` restituisce NULL; terminare per evitare crash.
- Inizializzazione
  - Imposto valori logici in ciascuna cella.
- Uso
  - Accesso sequenziale con `arr[i]`.
- `realloc`
  - Ridimensiona il blocco a `new_n` interi, copiando i dati esistenti.
- Gestione dell'errore
  - Se `realloc` fallisce, lascia `arr` invariato; liberare la memoria originale.
- Inizializzazione estesa
  - Popolo la nuova parte dell'array.
- Verifica
  - Stampo tutti i valori per confermare il corretto ridimensionamento.
- `free(arr)`
  - Restituisce l'intero blocco di memoria all'heap.
- `arr = NULL;`
  - Evita che il puntatore punti a memoria libera, prevenendo accessi invalidi.

```
1. #include <stdio.h>
2. #include <stdlib.h> // per malloc, calloc, realloc, free
3.
4. int main(void) {
5.     size_t n = 5;
6.
7.     // 1) Alloco dinamicamente un array di n interi
8.     int *arr = malloc(n * sizeof *arr);
9.     if (arr == NULL){           // 2) Controllo di errore
10.         fprintf(stderr, "malloc fallita\n");
11.         return EXIT_FAILURE;
12.     }
13.
14.    // 3) Inizializzo l'array
15.    for (size_t i = 0; i < n; i++) {
16.        arr[i] = (int)(i + 1) * 10; // es. 10, 20, 30, ...
17.    }
18.
19.    // 4) Uso dell'array
20.    printf("Array iniziale:\n");
21.    for (size_t i = 0; i < n; i++) {
22.        printf("arr[%zu] = %d\n", i, arr[i]);
23.    }
24.
25.    // 5) Ridimensionamento: duplico la capacità
26.    size_t new_n = n * 2;
27.    int *tmp = realloc(arr, new_n * sizeof *arr);
28.    if (tmp == NULL){           // 6) Controllo di errore su realloc
29.        fprintf(stderr, "realloc fallita\n");
30.        free(arr);            // libero la vecchia memoria
31.        return EXIT_FAILURE;
32.    }
33.    arr = tmp;                // aggiorno il puntatore
34.
35.    // 7) Inizializzo la seconda metà dell'array
36.    for (size_t i = n; i < new_n; i++) {
37.        arr[i] = (int)(i + 1) * 10;
38.    }
39.
40.    // 8) Stampa dopo realloc
41.    printf("\nArray dopo realloc a %zu elementi:\n", new_n);
42.    for (size_t i = 0; i < new_n; i++) {
43.        printf("arr[%zu] = %d\n", i, arr[i]);
44.    }
45.
46.    // 9) Dealloco la memoria
47.    free(arr);
48.    arr = NULL;               // 10) Prevengo dangling pointer
49.
50.    return EXIT_SUCCESS;
51.}
```

# Elementi pratici - Funzioni - Linker

In C, il linker deve sapere quali simboli (funzioni o variabili) sono definiti in ogni unità di traduzione (file .c) per combinarli in un unico eseguibile.

Il declaratore `extern` dichiara un simbolo senza definirlo: informa il compilatore che la sua definizione si trova in un altro file. Questo meccanismo permette di suddividere il codice in moduli, rendendo possibile compilazioni separate e promuovendo la riusabilità.

## Dichiarazione vs Definizione

Definizione (alloca spazio o fornisce il corpo):

```
1. int counter = 0;           // variabile globale definita  
2. void foo(void) { ... }    // funzione definita
```

Dichiarazione (`extern`, senza allocazione):

```
1. extern int counter;        // solo avvisa il linker  
2. extern void foo(void);    // indica esistenza della funzione
```

# Elementi pratici - Funzioni - Linker

## Visibilità e linkage

- **extern** ha **linkage esterno**: il **simbolo** è **condiviso tra i file**.
- **static** a **livello di file** limita la **visibilità al solo file corrente**.

## Header file

- **Dichiarazioni extern vengono poste in .h per includerle ovunque serva.**

## Regole di compilazione

- **Ogni simbolo con extern deve avere esattamente una definizione in un file .c.**

## Errori comuni

- **Doppie definizioni** → errori di “**multiple definition**”.
- **Mancata definizione** → “**undefined reference**” durante il **linking**.

# Elementi pratici - Funzioni - Linker

## counter.h

- Contiene dichiarazioni **extern**, incluse da tutti i file che usano **counter** o **incrementa**.

## counter.c

- Definisce **counter** (alloca memoria) e fornisce il corpo di **incrementa()**.

## main.c

- Include **counter.h**, vede le dichiarazioni e può chiamare **incrementa()** e accedere a **counter**.

```
1.#ifndef COUNTER_H
2.#define COUNTER_H
3.
4.extern int counter; // dichiarazione della variabile globale
5.void incrementa(void); // dichiarazione della funzione
6.
7.#endif
```

---

```
1.#include "counter.h"
2.
3.int counter = 0; // definizione e inizializzazione
4.
5.void incrementa(void) {
6.    counter++;
7.}
```

---

```
1.#include <stdio.h>
2.#include "counter.h" // include le dichiarazioni extern
3.
4.int main(void) {
5.    printf("Counter iniziale: %d\n", counter);
6.    incrementa();
7.    printf("Counter dopo incrementa(): %d\n", counter);
8.    return 0;
9.}
```

# **Elementi pratici - Struct**

## Elementi pratici - Struct

Le strutture in C (struct) consentono di raggruppare in un'unica entità più variabili di tipi diversi, creando nuovi tipi complessi.

Sono fondamentali per rappresentare oggetti reali (es. un punto con coordinate x e y, un libro con titolo, autore e anno), facilitando l'organizzazione dei dati e la loro gestione.

Una volta definito un struct, è possibile dichiarare variabili di quel tipo, passarle per valore o tramite puntatore, e inizializzarle sia in modo posizionale sia con i designated initializers (C99).

# Elementi pratici - Struct

- **Definizione**

```
1. struct NomeStruct {  
2.     tipo1 campo1;  
3.     tipo2 campo2;  
4.     ... };
```

- **Passaggio alle funzioni**

- Per valore: copia dell'intera struttura (potenzialmente costoso).
- Per puntatore: void fn(struct NomeStruct \*s).

- **Array di struct**

- struct Libro biblioteca[100];

- **Nesting**

- I campi possono essere a loro volta struct o array.

- **Allineamento e padding**

- Il compilatore può aggiungere byte di padding tra i campi per allineamento; usare sizeof per conoscere la dimensione effettiva.

# Elementi pratici - Struct

## Dichiarazione di variabili

```
1. typedef struct NomeStruct {
2.     ...
3. } Nome;
4. Nome var2;
```

## Accesso ai campi

- Con . se si ha la variabile: var.campo1
- Con -> se si ha un puntatore: ptr->campo2

## Inizializzazione

- **Posizionale:**

```
1. struct Punto p = { 10, 20 };
```

- **Designated:**

```
1. struct Punto p = { .y = 20, .x = 10 };
```

# Elementi pratici - Struct

- `typedef struct { ... } Libro;`

Definisce il tipo Libro senza dover scrivere struct ad ogni variabile.

- `void stampa_libro(const Libro *l)`

Riceve un puntatore a Libro, usa `->` per accedere ai campi senza copiare la struct.

- Inizializzazione posizionale di l1

I valori corrispondono all'ordine dei campi definiti.

- Designated initializers per l2

Imposta ogni campo specificando il nome, indipendentemente dall'ordine.

- Array biblioteca

Copia l1 e l2 nell'array; utile per collezioni di struct.

- Stampa con ciclo

Passa sempre l'indirizzo (`&biblioteca[i]`) a `stampa_libro` per efficienza e chiarezza.

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. // 1) Definizione di una struct per rappresentare un libro
5. typedef struct {
6.     char titolo[50];
7.     char autore[30];
8.     int anno_pubblicazione;
9.     float prezzo;
10.} Libro;
11.
12. // 2) Funzione che stampa i dati di un Libro passato per puntatore
13. void stampa_libro(const Libro *l) {
14.     printf("Titolo : %s\n", l->titolo);
15.     printf("Autore : %s\n", l->autore);
16.     printf("Anno : %d\n", l->anno_pubblicazione);
17.     printf("Prezzo : %.2f EUR\n", l->prezzo);
18. }
19.
20. int main(void) {
21.     // 3) Inizializzazione posizionale
22.     Libro l1 = {
23.         "Il Nome della Rosa",
24.         "Umberto Eco",
25.         1980,
26.         12.50f
27.     };
28.
29.     // 4) Inizializzazione con designated initializers
30.     Libro l2 = {
31.         .autore = "J.R.R. Tolkien",
32.         .titolo = "Lo Hobbit",
33.         .prezzo = 15.00f,
34.         .anno_pubblicazione = 1937
35.     };
36.
37.     // 5) Array di struct
38.     Libro biblioteca[2];
39.     biblioteca[0] = l1;
40.     biblioteca[1] = l2;
41.
42.     // 6) Stampa dei libri
43.     for (int i = 0; i < 2; i++) {
44.         printf("\n--- Libro %d ---\n", i+1);
45.         stampa_libro(&biblioteca[i]); // passo per puntatore per evitare copia
46.     }
47.
48.     return 0;
49. }
```

**Elementi pratici - Stream / File**

# Elementi pratici - File

La gestione dei file in C avviene principalmente tramite l'astrazione dei file stream definita in `<stdio.h>`.

Un file stream (`FILE *`) rappresenta un canale sequenziale con un file su disco o con uno degli stream predefiniti (`stdin`, `stdout`, `stderr`).

Le operazioni di apertura e chiusura (`fopen`, `fclose`) garantiscono l'inizializzazione e il rilascio delle risorse; la lettura e la scrittura possono avvenire in modalità testuale (funzioni basate su caratteri e stringhe) o binaria (blocchi di memoria con `fread/fwrite`).

Il buffering automatico ottimizza le prestazioni, ma va gestito (`fflush`, `setvbuf`) in situazioni critiche. Per muoversi all'interno del file si utilizzano funzioni di posizionamento (`fseek`, `ftell`, `rewind`).

È fondamentale controllare sempre lo stato dello stream con `feof` e `ferror`, e gestire correttamente gli errori ottenuti tramite la variabile globale `errno` e la funzione `perror`.

# Elementi pratici - File

La struttura opaca FILE, definita in `<stdio.h>`, rappresenta internamente uno stream di I/O in C.

Essa incapsula tutte le informazioni necessarie per gestire sia file su disco sia flussi standard (`stdin`, `stdout`, `stderr`): puntatori al buffer, contatori di posizione, modalità di apertura (lettura, scrittura, append), flag di stato (EOF, errori), e dati di sincronizzazione con il sistema operativo.

Ogni chiamata a `fopen()` restituisce un puntatore a un oggetto FILE opportunamente inizializzato; le funzioni di lettura/scrittura (`fread`, `fwrite`, `fgetc`, `fputs`, ecc.) operano su questo oggetto, gestendo automaticamente il buffering per ottimizzare l'accesso ai dati.

Alla chiusura con `fclose()`, tutte le risorse associate (buffer, descriptor di sistema) vengono rilasciate e, se necessario, le scritture rimanenti nel buffer vengono flushate su disco.

# Elementi pratici - File

- **Apertura/chiusura**
  - `FILE *f = fopen("file.txt", "r");`
  - `int r = fclose(f);`
- **Lettura/scrittura binaria**
  - `size_t n = fread(ptr, size, count, f);`
  - `size_t m = fwrite(ptr, size, count, f);`
- **Buffering**
  - **Default:** line-buffered su terminale, fully-buffered su file.
  - **Controllo:** `fflush(f)`, `setvbuf(f, buf, _IOLBF/_IOFBF/_IONBF, size)`.
- **Posizionamento nel file**
  - `fseek(f, offset, SEEK_SET/SEEK_CUR/SEEK_END)`
  - `long pos = ftell(f);`
  - `rewind(f);`
- **Controllo di fine file ed errori**
  - `while (!feof(f)),`
  - `if (ferror(f)),`
  - `perror("messaggio")` (va inclusa `<errno.h>`).
- **Rimozione e rinomina**
  - `remove("file.txt");`
  - `rename("old", "new");`

# Elementi pratici - File

- Includo `<stdio.h>`, `<errno.h>`, `<stdlib.h>`
  - `<stdio.h>` per gestire stream e funzioni di I/O.
  - `<errno.h>` per leggere `errno` in caso di errore.
  - `<stdlib.h>` per costanti di esito e allocazione.
- `fopen("input.txt", "r")`

Apre il file in modalità lettura;  
restituisce `NULL` e setta `errno` se fallisce.
- `fgetc / fputc`

Funzioni per leggere e scrivere caratteri singoli attraverso stream.
- `ferror(fp)`

Controlla se si è verificato un errore di lettura sullo stream.
- `fclose(fp)`

Chiude lo stream associato al file,  
rilasciando risorse.

```
1. #include <stdio.h> // definisce FILE, fopen, fclose, fgetc, fputc, perror, stdin, stdout
2. #include <errno.h> // definisce errno
3. #include <stdlib.h> // definisce EXIT_FAILURE, EXIT_SUCCESS
4.
5. int main(void) {
6.     // 1) Apro un file per lettura
7.     FILE *fp = fopen("input.txt", "r");
8.     if (!fp) {
9.         // 2) Se fopen fallisce, errno è settato
10.        perror("Errore apertura file");
11.        return EXIT_FAILURE;
12.    }
13.
14.    // 3) Copio carattere per carattere da fp a stdout
15.    int ch;
16.    while ((ch = fgetc(fp)) != EOF) {
17.        fputc(ch, stdout);
18.    }
19.
20.    // 4) Controllo errori di lettura
21.    if (ferror(fp)) {
22.        perror("Errore durante la lettura");
23.        fclose(fp);
24.        return EXIT_FAILURE;
25.    }
26.
27.    // 5) Chiudo lo stream
28.    fclose(fp);
29.    return EXIT_SUCCESS;
30.}
```

# Elementi pratici - Stream

In C, tutte le funzioni e i tipi per gestire i file stream—ossia la lettura e scrittura attraverso streams come file su disco o input/output standard—sono definiti principalmente nell'header `<stdio.h>`.

Per gestire gli errori delle funzioni di I/O, in particolare leggere il codice di errore, serve anche `<errno.h>`, che definisce la variabile globale `errno` e le macro associate.

In alcuni casi, per convertire stringhe in numeri dopo la lettura da stream, si usa `<stdlib.h>`, ma specificamente per aprire, leggere, scrivere, chiudere e manipolare stream in stile C è essenziale includere `<stdio.h>` (e optionalmente `<errno.h>`).

# Elementi pratici - Stream

Il C fornisce tre stream predefiniti accessibili senza doverli aprire esplicitamente:

- **stdin**: standard input, di default collegato alla tastiera.
- **stdout**: standard output, solitamente legato al terminale; è line-buffered, quindi svuota il buffer ad ogni newline (\n).
- **stderr**: standard error, anch'esso collegato al terminale ma unbuffered, per consentire la visualizzazione immediata dei messaggi di errore.

# Elementi pratici - Stream

## <stdio.h>

- **FILE**: definisce il tipo opaco per rappresentare un stream.
- **fopen**, **fclose**: aprono e chiudono uno stream su file.
- **fread**, **fwrite**: lettura/scrittura binaria.
- **fgetc**, **fputc**, **fgets**, **fputs**: lettura/scrittura di caratteri e stringhe.
- **fprintf**, **fscanf**: I/O formattato su stream.
- **stdin**, **stdout**, **stderr**: stream predefiniti.

## <errno.h>

- **errno**: variabile globale che indica l'ultimo errore di libreria.
- **perror()**: stampa messaggio di errore compatibile con **errno**.

## <stdlib.h> (opzionale)

- **EXIT\_SUCCESS**, **EXIT\_FAILURE**: costanti per **return** in **main**.
- **malloc**, **free**: per buffer dinamici se necessario.

## Ruolo

- Separazione tra interfaccia (dichiarazioni in header) e implementazione (in libreria).
- Inclusion guard: evita multiple inclusioni accidentali.

# Elementi pratici - Stream vs File

In C, un file è una risorsa di memoria secondaria (es. un file su disco) organizzata in byte e identificata da un nome nel filesystem; rappresenta l'archiviazione permanente di dati.

Un stream, invece, è un'astrazione di flusso di dati "in movimento" tra il programma e una sorgente o destinazione (file, tastiera, schermo, pipe, socket).

Mentre il file è un'entità fisica e statica sul supporto di memorizzazione, lo stream incapsula un canale di comunicazione sequenziale, gestito internamente dalla libreria standard attraverso buffer, puntatori di lettura/scrittura e controlli di stato (EOF, errori).

Questa distinzione consente di usare le stesse funzioni di I/O (`fread`, `fprintf`, `fgetc`, ecc.) sia per file su disco sia per input/output standard, rendendo il codice più portatile e flessibile: ad esempio, `stdin` e `stdout` sono semplici stream, non file fisici, ma vengono manipolati con le stesse API.

# Elementi pratici - Stream vs File

- **File**

- **Identificato da un pathname.**
- **Accesso casuale (seek), dimensione nota.**
- **Persistente dopo la terminazione del programma.**

- **Stream**

- **Accesso sequenziale, gestito da buffer.**
- **Può rappresentare file, dispositivi o pipe.**
- **Stato dinamico: EOF, errori, buffering.**

Grazie a questa separazione, un programma C può leggere o scrivere indifferentemente su file, terminale o altri dispositivi utilizzando le medesime chiamate di libreria, delegando alla gestione degli stream la complessità del buffering e della sincronizzazione con l'hardware o il sistema operativo.

# Elementi pratici - Stream

- `fopen("input.txt", "r")`

Apre in modalità testo; NULL se fallisce, errno spiega il motivo.

- `fopen("output.bin", "wb")`

Apre in modalità binaria per scrittura; crea o tronca il file.

- `fgets + printf`

Legge una riga per volta (incluso il \n) e la stampa su stdout.

- `fread + fwrite`

Copia blocchi di byte, gestendo eventuali scritture parziali e controllando fwrite.

- `fseek + fread`

Sposta il puntatore all'offset desiderato e legge un valore int.

- `fflush`

Svuota il buffer di out sul disco prima di chiudere.

- `rename / remove (commentate)`

Mostrano come rinominare o cancellare file nel filesystem.

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <errno.h>
4.
5. int main(void) {
6.     // 1) Apertura in lettura testuale
7.     FILE *in = fopen("input.txt", "r");
8.     if (!in) {
9.         perror("Errore apertura input.txt"); // errno setta il perché
10.        return EXIT_FAILURE;
11.    }
12.
13.    // 2) Apertura in scrittura binaria
14.    FILE *out = fopen("output.bin", "wb");
15.    if (!out) {
16.        perror("Errore apertura output.bin");
17.        fclose(in);
18.        return EXIT_FAILURE;
19.    }
20.
21.    // 3) Lettura riga per riga e stampa a stdout
22.    char buffer[256];
23.    while (fgets(buffer, sizeof buffer, in)) {
24.        printf("Letto: %s", buffer);
25.    }
26.    if (ferror(in)) {
27.        perror("Errore durante la lettura di input.txt");
28.    }
29.    rewind(in); // ritorno all'inizio per la fase binaria
30.
31.    // 4) Copia binaria da input a output
32.    unsigned char chunk[128];
33.    size_t n;
34.    while ((n = fread(chunk, 1, sizeof chunk, in)) > 0) {
35.        size_t written = fwrite(chunk, 1, n, out);
36.        if (written < n) {
37.            perror("Errore durante la scrittura in output.bin");
38.            break;
39.        }
40.    }
41.    if (ferror(in)) {
42.        perror("Errore di lettura binaria");
43.    }
44.
45.    // 5) Posizionamento e lettura di un int in mezzo al file binario
46.    if (fseek(out, sizeof chunk * 2, SEEK_SET) == 0) {
47.        int val;
48.        if (fread(&val, sizeof val, 1, out) == 1) {
49.            printf("Valore int letto da output.bin a offset %ld: %d\n",
50.                   (long)(sizeof chunk * 2), val);
51.        }
52.    }
53.
54.    // 6) Flush e chiusura
55.    fflush(out); // assicuro che tutto sia scritto su disco
56.    fclose(in);
57.    fclose(out);
58.
59.    // 7) Rinomino e rimozione file temporaneo di esempio
60.    // rename("output.bin", "data.bin");
61.    // remove("input.txt");
62.
63.    return EXIT_SUCCESS;
64.}
```

