

LEZIONE 1

Introduzione alla Programmazione in C



Dispense di Programmazione 1 – Canale O-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania

Lezione 1

Introduzione alla Programmazione in C

Benvenuti a **Programmazione 1**! Il nostro obiettivo in questa lezione è demistificare la “scatola nera” del computer, comprendere l’evoluzione della programmazione nell’era dell’Intelligenza Artificiale, collegare l’hardware ai concetti di astrazione del C e muovere i primi passi pratici al terminale con la compilazione ed l’ispezione dei valori di ritorno (echo \$?).

1.1 Dalla Fisica all’Astrazione: Il Ruolo del Programmatore Oggi

1.1.1 Definizione di Programmazione, Rilevanza e Applicazioni

La **programmazione** non consiste nel memorizzare sintassi o comandi a memoria. È l’arte di modellare il **pensiero algoritmico**: la capacità di prendere problemi complessi, disordinati e reali della scienza e della società e tradurli in strutture logiche trasparenti ed istruzioni deterministiche comprese dalle macchine.

La programmazione è oggi il motore invisibile di ogni disciplina moderna: dalle simulazioni dei cambiamenti climatici alla bioinformatica avanzata, dai sistemi aerospaziali alla medicina di precisione, fino alla domotica ed alla finanza.

1.1.2 L’Epoca dell’IA ed i Dati dell’Industria

Iniziamo il nostro percorso affrontando apertamente il contesto tecnologico in cui ci troviamo. L’Intelligenza Artificiale Generativa ha trasformato radicalmente il panorama del software engineering.

Analizziamo i dati reali ed ufficiali dell’industria:

- **GitHub Octoverse (2023)**: Il **92%** degli sviluppatori usa o sperimenta strumenti di coding basati su AI, anche fuori dal lavoro.
- **World Economic Forum (Future of Jobs Report 2023)**: Basato su 803 aziende e oltre 11,3 milioni di lavoratori, il report stima che il **44%** delle competenze sarà trasformato nei prossimi cinque anni.
- **Gartner (2024)**: Entro il 2028, il **75%** dei programmatori in ambito enterprise userà assistenti AI per il codice.



Figura 1.1: La Pervasività della Programmazione

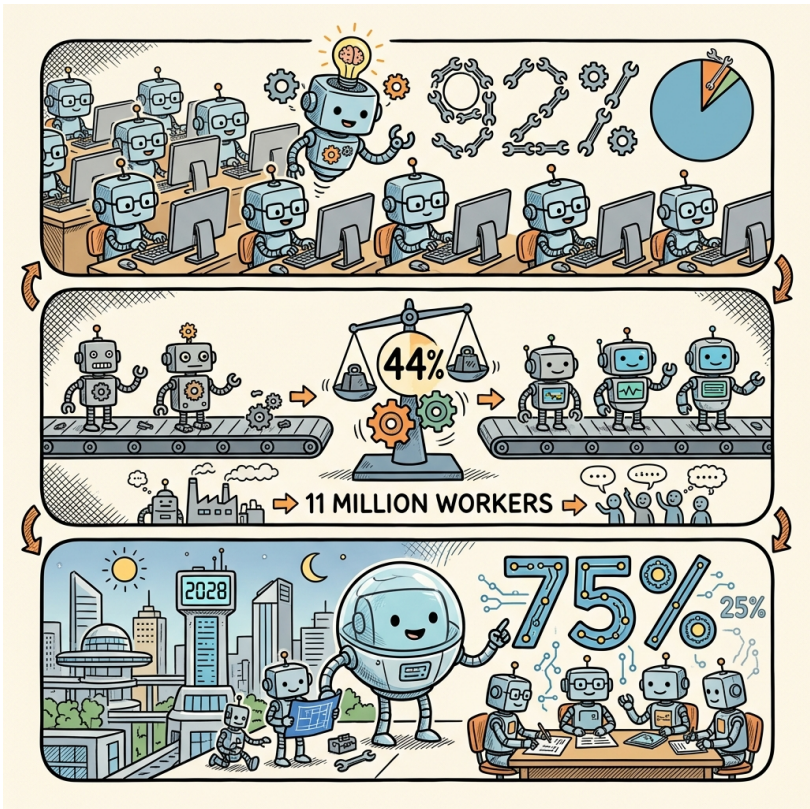


Figura 1.2: Infografica Dati IA



Figura 1.3: Premi Nobel 2024

1.1.3 Ma allora, perché imparare a programmare nell'era dell'IA?

Se gli strumenti IA possono generare codice in pochi millisecondi, qual è il valore dello studio dell'informatica?

L'Intelligenza Artificiale non è magia: è costruita interamente su strutture dati, allocazione di memoria, matrici, algoritmi ed equazioni algebriche. Imparare a programmare significa padroneggiare la grammatica ed la sintassi fondamentale di questo linguaggio per guidare la rivoluzione tecnologica anziché subirla.

1.1.4 L'Informatica è la Nuova Fisica (Nobel 2024)

L'informatica è diventata il nuovo telescopio e microscopio della scienza moderna. A conferma di ciò, i **Premi Nobel 2024** hanno consacrato il ruolo centrale degli algoritmi:

- **Nobel per la Fisica:** Assegnato a John Hopfield e Geoffrey Hinton per le scoperte fondamentali che consentono l'apprendimento automatico tramite reti neurali artificiali basate sulla fisica statistica.
- **Nobel per la Chimica:** Assegnato a Demis Hassabis, John Jumper e David Baker per la progettazione computazionale delle proteine e per **AlphaFold**, l'algoritmo che ha risolto il problema cinquantennale del ripiegamento proteico.



Figura 1.4: La Metafora del Direttore d'Orchestra

1.1.5 La Metafora del Direttore d'Orchestra (Agentic Engineering)

L'IA non sostituisce il programmatore, ma ne eleva il ruolo verso l'**Ingegneria Agantica** (*Agentic Engineering*). Il programmatore moderno coordina e supervisiona reti di agenti IA autonomi.

La Metafora: Il direttore d'orchestra non suona ogni singolo strumento dal vivo durante un concerto, ma possiede una conoscenza impeccabile della musica, del tempo, dell'armonia e degli spartiti. Senza la visione rigorosa del direttore, l'orchestra di robot e strumenti automatici suona in modo disarmonico e cade nel caos.

1.2 L'Evoluzione del Calcolo e l'Architettura di Von Neumann

1.2.1 La Pascalina e l'Informatica Meccanica (1642)

Agli albori del calcolo, l'algoritmo era letteralmente fuso nel metallo. Nella **Pascalina** inventata da Blaise Pascal, le ruote dentate e gli ingranaggi rappresentavano le operazioni aritmetiche. Per cambiare il calcolo o l'algoritmo, occorreva smontare e riprogettare fisicamente i ruotismi meccanici.

1.2.2 ENIAC e la Fisicità del Ricablaggio (Anni '40)

Con l'avvento dell'elettronica (le valvole termoioniche), l'**ENIAC** consentì calcoli infinitamente più veloci. Tuttavia, la riprogrammazione era ancora un'attività fisica estenuante: per cambiare



Figura 1.5: La Pascalina

programma, le ingegnere pioniere dovevano staccare e riattaccare fisicamente centinaia di cavi in enormi pannelli telefonici. *Riprogrammare significava letteralmente ricablare!*

1.2.3 John von Neumann e lo *Stored Program Concept* (1945)

La gigantesca intuizione del matematico John von Neumann fu il concetto di **Programma Memorizzato** (*Stored Program Concept*): memorizzare le istruzioni del programma nello stesso spazio fisico di memoria (la RAM) che contiene i dati su cui il programma opera.

1.2.4 La Macchina Universale (CPU vs RAM)

Grazie a questa architettura, l'hardware del computer è diventato immutabile e generico. La **CPU** (Central Processing Unit) esegue istruzioni in sequenza, mentre la **RAM** (Random Access Memory) ospita sia i dati che i programmi. Per trasformare il computer da calcolatrice a videogioco o elaboratore di testo, non serve più modificare i cavi: basta caricare sequenze di numeri differenti in memoria.

1.3 I Livelli di Astrazione: Dal Silicio al C



Figura 1.6: L'ENIAC ed il Ricablaggio

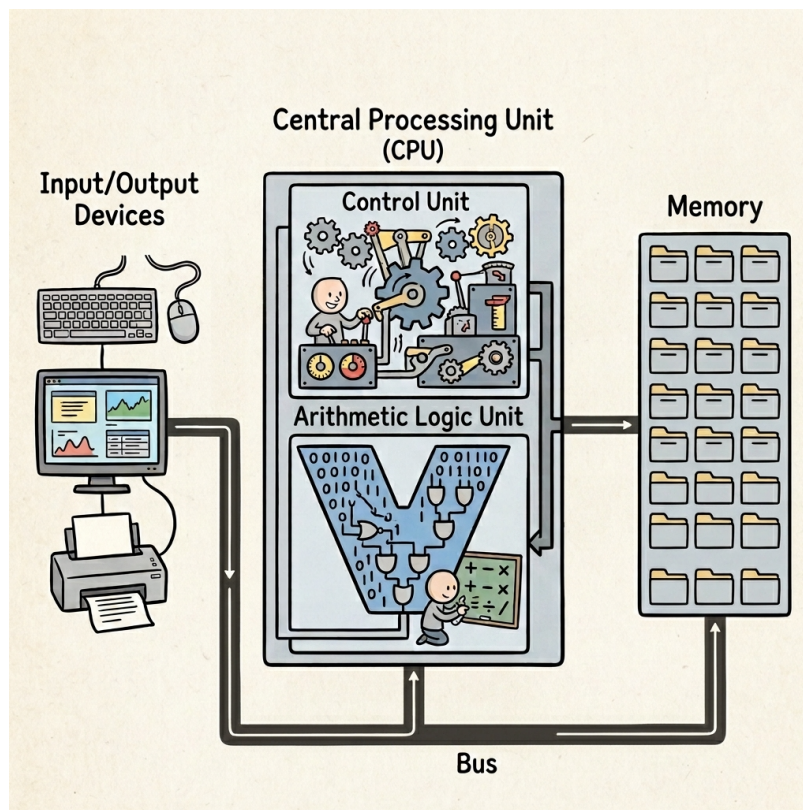


Figura 1.7: Von Neumann Architecture



Figura 1.8: I Livelli di Astrazione

1.3.1 Dal Codice Macchina al Linguaggio C

La macchina comprende ed esegue esclusivamente istruzioni binarie in **Codice Macchina** (sequenze di 0 ed 1 che corrispondono a livelli di tensione elettrica). Per non dover scrivere direttamente in binario, l'informatica ha creato diversi **livelli di astrazione**:

- **Livello Basso (Codice Macchina & Assembly):** Istruzioni binarie o mnemoniche rigide legate all'architettura specifica del chip (es. mov, add).
- **Livello Alto (Linguaggio C):** Esprime gli algoritmi in forma algebrica e strutturata, vicina al pensiero umano, astenendosi dai dettagli minuti del singolo processore.

1.3.2 La Metafora dell'Automobile e del Meccanico

Per comprendere l'importanza dei livelli di astrazione, pensiamo alla guida di un'automobile.

Per guidare un'auto ed andare da un punto A ad un punto B non ci serve sapere come funziona la camera di combustione o la fasatura dei pistoni: ci basta usare l'astrazione del volante e dei pedali (guidatore). Tuttavia, se la macchina si guasta o dobbiamo progettare un motore ad alte prestazioni, l'astrazione del volante non basta più: dobbiamo aprire il cofano e capire esattamente come interagiscono ingranaggi e pistoni (meccanico). Per risolvere un determinato problema in informatica serve sempre il giusto livello di astrazione!

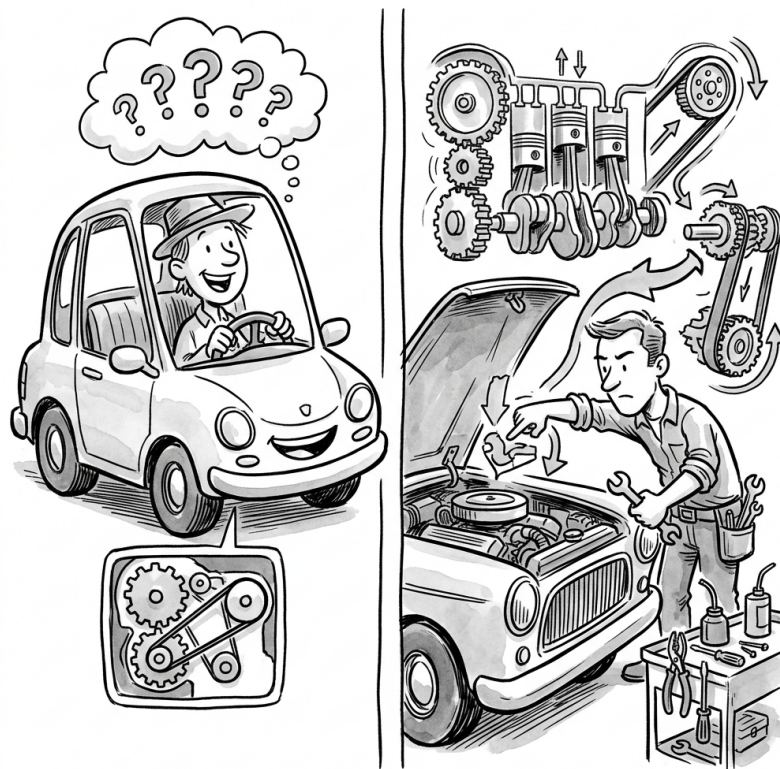


Figura 1.9: Metafora Guidatore e Meccanico

1.4 Sotto il Cofano: Registri e Traduzione Assembly

1.4.1 Il Collo di Bottiglia di Von Neumann ed i Registri CPU

Sebbene la CPU sia in grado di eseguire miliardi di operazioni al secondo, la RAM è un componente esterno collegato da linee fisiche (il **Bus**). Inviare e pescare continuamente dati dalla RAM ad ogni singola operazione crea il cosiddetto **Collo di Bottiglia di Von Neumann** (*Von Neumann Bottleneck*), che rallenterebbe l'elaborazione di oltre il 90%.

Per superare questo limite, i progettisti hanno saldato direttamente *all'interno* della CPU una manciata di celle di memoria velocissime chiamate **Registri**.

La CPU funziona come un operaio con le tasche: copia i dati necessari dalla RAM nelle proprie tasche (i registri), esegue il calcolo aritmetico istantaneo tramite la ALU (Arithmetic Logic Unit) e infine deposita il risultato in RAM.

1.4.2 Dal Codice Umano all'Assembly: L'illusione delle Variabili

Quando scriviamo in C un frammento come:

```
1 int main(void) {  
2     int x = 3;  
3     int y = 5;  
4     return x + y;  
5 }
```

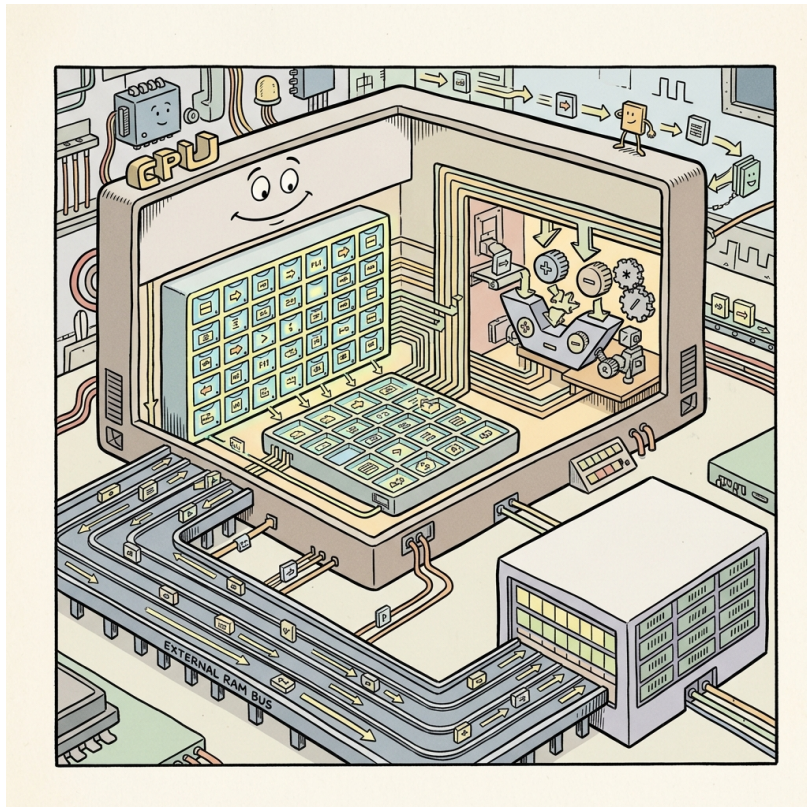


Figura 1.10: Processore basato su Registri Generali

Per la CPU le lettere *x* e *y* **non esistono**. Esse sono un'illusione ad alto livello creata dal linguaggio C per permetterci di ragionare in modo algebrico.

Guardiamo cosa genera il compilatore traducendo questo frammento in codice Assembly (sintassi Intel x86-64):

```

1  push    rbp                ; Salva il frame pointer precedente
   ↪ sullo stack
2  mov     rbp, rsp           ; Imposta il nuovo frame pointer per la
   ↪ funzione attuale
3  mov     DWORD PTR [rbp-4], 3 ; Copia il valore 3 nello spazio RAM di
   ↪ 'x'
4  mov     DWORD PTR [rbp-8], 5 ; Copia il valore 5 nello spazio RAM di
   ↪ 'y'
5  mov     edx, DWORD PTR [rbp-4] ; Carica 'x' dalla RAM al registro CPU
   ↪ edx
6  mov     eax, DWORD PTR [rbp-8] ; Carica 'y' dalla RAM al registro CPU
   ↪ eax
7  add     eax, edx            ; Somma il valore di edx nel registro
   ↪ eax
8  pop     rbp                ; Ripristina il frame pointer
   ↪ precedente
9  ret     0                  ; Restituisce il valore in eax al
   ↪ Sistema Operativo

```

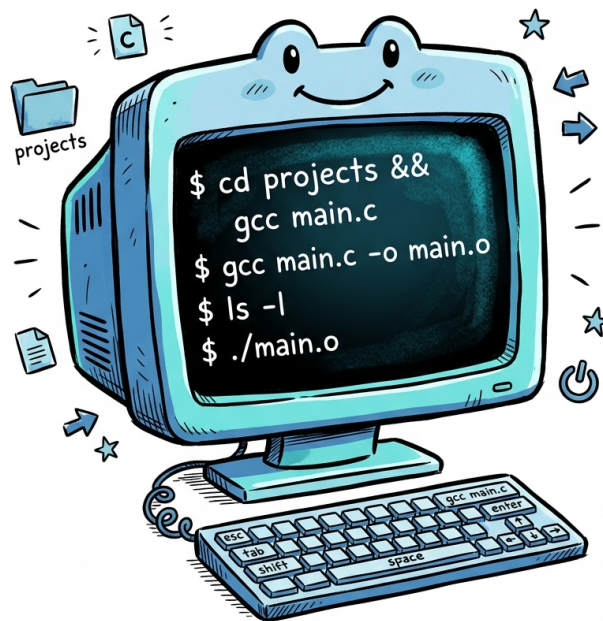


Figura 1.11: La riga di comando come intermediaria con il Sistema Operativo

Suggerimento

Strumento Utile: Compiler Explorer Potete sperimentare ed ispezionare in tempo reale come qualsiasi compilatore (gcc, clang) traduce il vostro codice C in Assembly utilizzando il celebre sito web interattivo [Compiler Explorer \(godbolt.org\)](https://godbolt.org).

1.5 La Riga di Comando: Parlare Direttamente al Sistema Operativo

Prima di addentrarci nella compilazione del codice, è fondamentale familiarizzare con lo strumento che ci permetterà di interagire direttamente con il computer: la **riga di comando** (o terminale).

1.5.1 Cos'è e a cosa serve

La riga di comando è un'interfaccia testuale che funge da intermediaria tra l'utente ed il Sistema Operativo (OS). A differenza delle classiche interfacce grafiche (GUI) a cui siamo abituati (fatte di finestre, icone e mouse), la riga di comando permette di impartire ordini digitando stringhe di testo ed eseguendo programmi o script in modo diretto ed efficiente.

Nell'era dello sviluppo moderno e dell'Intelligenza Artificiale, la riga di comando sta vivendo una grandissima rinascita di popolarità: gli assistenti di programmazione IA e gli agenti di coding autonomi operano ed eseguono strumenti quasi esclusivamente tramite interfacce

a riga di comando (CLI). Imparare ad utilizzarla è quindi una competenza chiave per ogni sviluppatore oggi.

1.5.2 Le Nostre Piattaforme: Shell Unix

In questo corso faremo riferimento allo standard delle shell di tipo **Unix** (come bash o zsh), che sono quelle preinstallate e utilizzate nativamente su **macOS** e **Linux**.

- **Windows:** Gli utenti Windows useranno il terminale fornito da **WSL** (Windows Subsystem for Linux), che mette a disposizione una shell Ubuntu Unix a tutti gli effetti.
- **Nota:** Windows possiede anche delle shell proprietarie (il Prompt dei comandi cmd e PowerShell), ma queste utilizzano comandi e sintassi differenti che non tratteremo in questo corso.

1.5.3 Comandi e Scorciatoie di Base

Per orientarsi ed operare agevolmente all'interno della shell Unix, è sufficiente memorizzare pochi semplici comandi e scorciatoie fondamentali riportati nella tabella seguente:

Tabella

Comando / Tasto	Nome completo	Descrizione
pwd	<i>Print Working Directory</i>	Stampa il percorso assoluto della cartella in cui ci si trova in questo momento.
ls	<i>List</i>	Elenca i file e le sottocartelle contenute nella cartella corrente.
cd <nome\{}_cartella>	<i>Change Directory</i>	Entra all'interno della cartella specificata.
cd ..	<i>Change Directory (Parent)</i>	Torna indietro di un livello, spostandosi nella cartella genitore.
./<eseguibile>	<i>Execute local</i>	Esegue un file binario o un programma presente all'interno della cartella corrente (il prefisso ./ dice alla shell di cercare il file proprio lì).

Tab (tasto Tabulazione)	<i>Autocomplete</i>	Premuto mentre si scrive il nome di un file o di una cartella, ne autocompleta il nome risparmiando tempo ed evitando errori di battitura.
Freccia Su / Freccia Giù	<i>History</i>	Permette di scorrere la cronologia dei comandi precedentemente digitati e premere Invio per rieseguirli.
clear o Ctrl + L	<i>Clear screen</i>	Pulisce completamente l'interfaccia visiva del terminale, lasciando la riga di comando in alto.

1.6 Pratica al Terminale: La Magia del Valore di Ritorno (echo \$?)

1.6.1 Prima Compilazione e il Valore di Ritorno

Proviamo a scrivere ed eseguire questo codice. Creiamo un file chiamato `somma.c` con il nostro `main` che restituisce `x + y`. Compiliamo ed eseguiamo dal terminale:

```
1 gcc somma.c -o somma
2 ./somma
```

Dopo aver premuto Invio, notiamo che **a schermo non appare nulla**. Perché? Perché non abbiamo incluso alcuna istruzione esplicita di stampa a video (come `printf`).

1.6.2 Interrogare il Sistema Operativo con echo \$?

Dove è finito il risultato della nostra somma? L'istruzione `return x + y;` (ovvero l'istruzione `ret` in Assembly) ha consegnato il valore 8 direttamente nelle mani del **Sistema Operativo**! Possiamo interrogare il Sistema Operativo digitando immediatamente questo comando speciale nel terminale:

```
1 echo $?
```

Il terminale risponderà stampando a video il numero 8!

Suggerimento

Exit Status Il comando `echo $?` stampa l'ultimo **Exit Status** (codice di uscita) restituito dal processo appena terminato. Questo dimostra in modo tangibile che l'istruzione `return` nel `main` comunica l'esito dell'esecuzione direttamente al Sistema Operativo.



Figura 1.12: Linguaggi Interpretati

1.7 Dai Traduttori all'Hello World

1.7.1 Compilatori vs Interpreti

I linguaggi di programmazione si dividono principalmente in due grandi famiglie di traduzione:

Linguaggi Interpretati (Traduzione Simultanea)

Linguaggi come Python o JavaScript sono **interpretati**. Un programma chiamato *interprete* legge il codice sorgente riga per riga e lo traduce in comandi macchina *mentre* il programma è in esecuzione.

Linguaggi Compilati (La Fabbrica del C)

Il C è un linguaggio **compilato**. Un programma specializzato chiamato **compilatore** (`gcc` o `clang`) analizza e traduce l'intero codice sorgente in anticipo, producendo un file binario eseguibile nativo e velocissimo.

1.7.2 Anatomia della Toolchain del Compilatore (Le 4 Fasi)

La compilazione C non è un atto unico, ma un processo articolato in 4 fasi distinte:

Ipotizziamo di avere un file sorgente minimale chiamato `main.c` (che non include nemmeno `stdio.h` per evitare di importare migliaia di righe esterne) contenente questo codice:

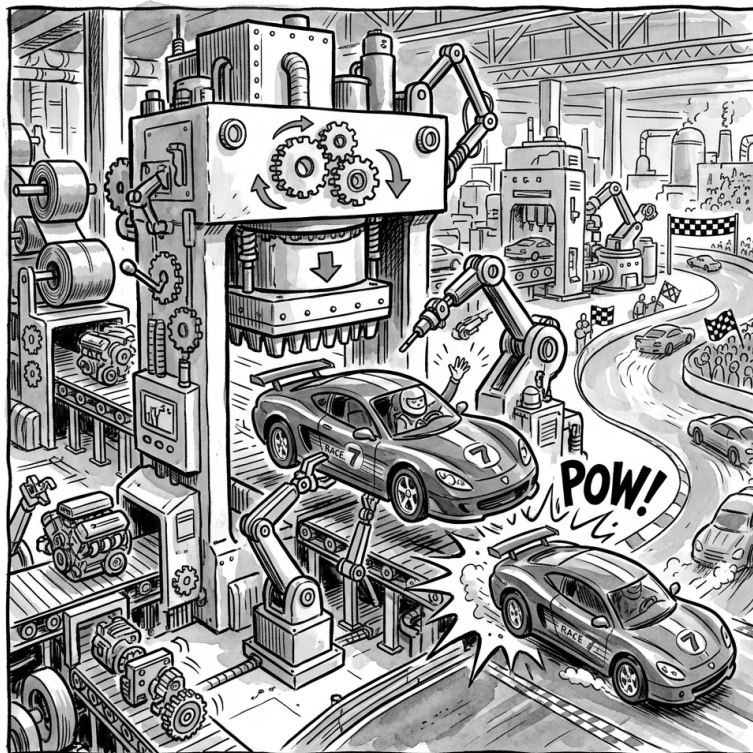


Figura 1.13: Linguaggi Compilati

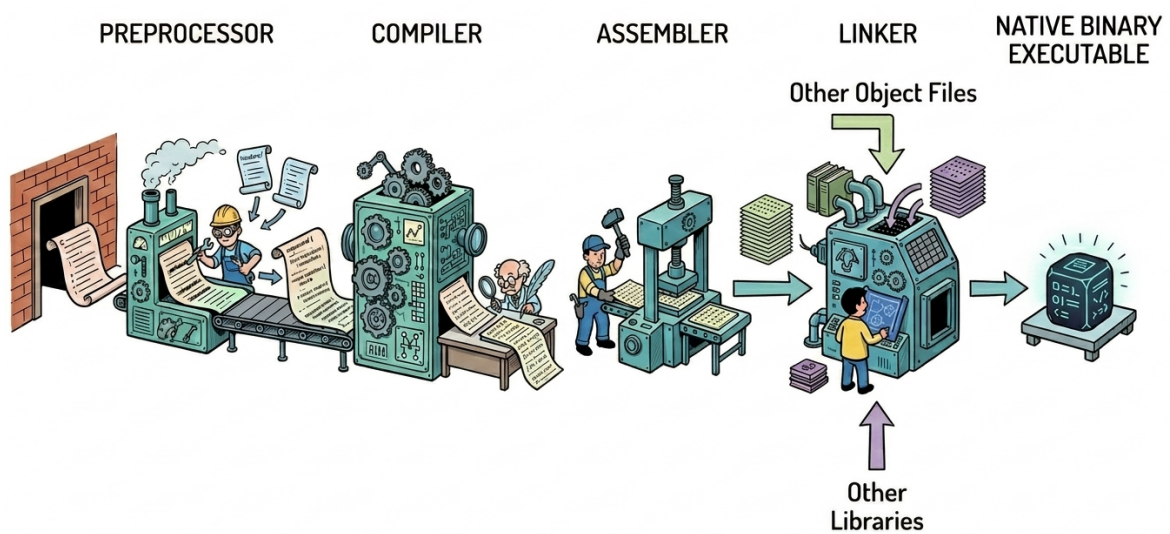


Figura 1.14: Toolchain di Compilazione C

```

1  int main(void) {
2      int x = 3;
3      int y = 5;
4      return x + y;
5  }

```

Vediamo i dettagli di ciascuna delle 4 fasi della toolchain, con i comandi per fermare il processo e ispezionare l'output generato:

1. **Preprocessore (cpp):** Analizza il testo, espande le direttive (es. `\{\}\#include`, `\{\}\#define`) ed elimina i commenti.

- **Comando:**

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

- **Risultato in main.i:**

```

1  # 1 "main.c"
2  # 1 "<built-in>" 1
3  # 1 "<built-in>" 3
4  # 482 "<built-in>" 3
5  # 1 "<command line>" 1
6  # 1 "<built-in>" 2
7  # 1 "main.c" 2
8  int main(void) {
9      int x = 3;
10     int y = 5;
11     return x + y;
12 }

```

- **Spiegazione:** Le righe che iniziano con il simbolo `\{\}\#` (dette *linemarkers*) sono generate dal preprocessore per comunicare al compilatore le informazioni di tracciamento sull'origine delle righe e sui file inclusi. Poiché non abbiamo importato alcun file header, il codice sorgente rimane intatto.

2. **Compilatore (gcc/clang):** Traduce il codice C pre-elaborato in codice Assembly specifico per l'architettura hardware (in questo esempio, ARM64/Apple Silicon).

- **Comando:**

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

- **Risultato in main.s:**

```

1      .build_version macos, 26, 0 sdk_version 26, 4
2      .section      __TEXT,__text,regular,pure_instructions
3      .globl _main          ; -- Begin function
4      ↪ main
5      .p2align      2
6      _main:                ; @main
7      .cfi_startproc

```

```

7 ; %bb.0:
8     sub sp, sp, #16
9     .cfi_def_cfa_offset 16
10    str wzr, [sp, #12]
11    mov w8, #3                                ; =0x3
12    str w8, [sp, #8]
13    mov w8, #5                                ; =0x5
14    str w8, [sp, #4]
15    ldr w8, [sp, #8]
16    ldr w9, [sp, #4]
17    add w0, w8, w9
18    add sp, sp, #16
19    ret
20    .cfi_endproc
21                                ; -- End function
22 .subsections_via_symbols

```

- **Spiegazione:** Questo codice è Assembly ARM64. Le righe che iniziano con il punto (es. `.section`, `.globl`) sono direttive per l'assembler. La funzione principale `{_main` alloca spazio sullo stack (`sub sp, sp, {#16}`), memorizza i valori 3 e 5 in memoria, li ricarica nei registri (`w8` e `w9`), esegue la somma (`add w0, w8, w9`) posizionando il risultato nel registro di ritorno `w0` (l'equivalente a 32 bit del registro `x0` per ARM), ripristina lo stack e ritorna con `ret`. L'elevato numero di metadati e direttive è tipico dei compilatori moderni per macchine Mac ARM.

3. **Assembler (as):** Traduce l'Assembly testuale in un file oggetto binario rilocabile (`main.o`) non ancora direttamente eseguibile.

- **Comando:**

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

- **Disassemblare per verificare:** Trattandosi di un file binario, non possiamo leggerlo direttamente come testo. Possiamo però usare uno strumento di disassemblaggio come `objdump` per convertire i byte binari in istruzioni assembly leggibili:

```
1 objdump -d main.o
```

- **Output di objdump:**

```

1 main.o: file format mach-o arm64
2
3 Disassembly of section __TEXT,__text:
4
5 0000000000000000 <tmp0>:
6      0: d10043ff      sub     sp, sp, #0x10
7      4: b9000fff      str     wzr, [sp, #0xc]
8      8: 52800068      mov     w8, #0x3                ; =3
9      c: b9000be8      str     w8, [sp, #0x8]
10     10: 528000a8      mov     w8, #0x5                ; =5
11     14: b90007e8      str     w8, [sp, #0x4]
12     18: b9400be8      ldr     w8, [sp, #0x8]
13     1c: b94007e9      ldr     w9, [sp, #0x4]

```

14	20: 0b090100	add	w0, w8, w9
15	24: 910043ff	add	sp, sp, #0x10
16	28: d65f03c0	ret	

- **Spiegazione:** A sinistra vedete gli indirizzi e i codici operativi esadecimali (es. 52800068 per `mov w8, \{\}#3`) che la CPU decodifica ed esegue, e a destra la corrispondente istruzione assembly mnemonica.

4. **Linker (ld):** Unisce i file oggetto e le librerie di sistema per creare l'eseguibile binario finale (`main`), completo di intestazioni di caricamento per il Sistema Operativo.

- **Comando:**

```
1 gcc main.o -o main
```

- **Ispezionare il binario eseguibile:** Possiamo visualizzare l'intero file in formato esadecimale usando `hexdump`:

```
1 hexdump -C main
```

- **Output parziale di hexdump:**

```
1 00000000  cf fa ed fe 0c 00 00 01  00 00 00 00 02 00 00 00
   ↳ |.....|
2 00000010  10 00 00 00 e8 02 00 00  85 00 20 00 00 00 00 00
   ↳ |.....|
3 00000020  19 00 00 00 48 00 00 00  5f 5f 50 41 47 45 5a 45
   ↳ |...H...__PAGEZE|
4 00000030  52 4f 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |RO
   ↳ |.....|
5 00000040  00 00 00 00 01 00 00 00  00 00 00 00 00 00 00 00
   ↳ |.....|
6 00000050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
   ↳ |.....|
7 00000060  00 00 00 00 00 00 00 00  19 00 00 00 e8 00 00 00
   ↳ |.....|
8 00000070  5f 5f 54 45 58 54 00 00  00 00 00 00 00 00 00 00 |
   ↳ |__TEXT.....|
9 00000080  00 00 00 00 01 00 00 00  00 40 00 00 00 00 00 00
   ↳ |.....@.....|
10 00000090  00 00 00 00 00 00 00 00  00 40 00 00 00 00 00 00
   ↳ |.....@.....|
11 000000a0  05 00 00 00 05 00 00 00  02 00 00 00 00 00 00 00
   ↳ |.....|
12 000000b0  5f 5f 74 65 78 74 00 00  00 00 00 00 00 00 00 00 |
   ↳ |__text.....|
13 000000c0  5f 5f 54 45 58 54 00 00  00 00 00 00 00 00 00 00 |
   ↳ |__TEXT.....|
14 000000d0  28 03 00 00 01 00 00 00  2c 00 00 00 00 00 00 00
   ↳ |(. . . . , . . . .|
15 000000e0  28 03 00 00 02 00 00 00  00 00 00 00 00 00 00 00
   ↳ |(. . . . .|
16 000000f0  00 04 00 80 00 00 00 00  00 00 00 00 00 00 00 00
   ↳ |. . . . .|
```

```

17 00000100 5f 5f 75 6e 77 69 6e 64 5f 69 6e 66 6f 00 00 00 |
    ↳ __unwind_info...|
18 00000110 5f 5f 54 45 58 54 00 00 00 00 00 00 00 00 00 00 |
    ↳ __TEXT.....|
19 00000120 54 03 00 00 01 00 00 00 58 00 00 00 00 00 00 00 |T
    ↳ .....X.....|
20 00000130 54 03 00 00 02 00 00 00 00 00 00 00 00 00 00 00 |T
    ↳ .....|
21 00000140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    ↳ |.....|
22 00000150 19 00 00 00 48 00 00 00 5f 5f 4c 49 4e 4b 45 44
    ↳ |...H...__LINKED|
23 00000160 49 54 00 00 00 00 00 00 00 40 00 00 01 00 00 00 |IT
    ↳ .....@.....|
24 00000170 00 40 00 00 00 00 00 00 00 40 00 00 00 00 00 00 |.e
    ↳ .....@.....|
25 00000180 c8 01 00 00 00 00 00 00 01 00 00 00 01 00 00 00
    ↳ |.....|
26 00000190 00 00 00 00 00 00 00 00 34 00 00 80 10 00 00 00
    ↳ |.....4.....|
27 000001a0 00 40 00 00 38 00 00 00 33 00 00 80 10 00 00 00 |.e
    ↳ ..8...3.....|
28 000001b0 38 40 00 00 30 00 00 00 02 00 00 00 18 00 00 00 |8e
    ↳ ..0.....|
29 000001c0 70 40 00 00 02 00 00 00 90 40 00 00 20 00 00 00 |pe
    ↳ .....@.. ...|
30 000001d0 0b 00 00 00 50 00 00 00 00 00 00 00 00 00 00 00
    ↳ |...P.....|
31 000001e0 00 00 00 00 02 00 00 00 02 00 00 00 00 00 00 00
    ↳ |.....|
32 000001f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    ↳ |.....|
33
34 *
35 00000220 0e 00 00 00 20 00 00 00 0c 00 00 00 2f 75 73 72
    ↳ |.... ...../usr|
36 00000230 2f 6c 69 62 2f 64 79 6c 64 00 00 00 00 00 00 00 |/
    ↳ lib/dyld.....|
37 00000240 1b 00 00 00 18 00 00 00 c6 3c 4a 2f 9b dd 3f 27
    ↳ |.....<J/..?'|
38 00000250 af 05 1a cb 5a 82 0f e5 32 00 00 00 20 00 00 00
    ↳ |...Z...2... ...|
39 00000260 01 00 00 00 00 00 1a 00 00 04 1a 00 01 00 00 00
    ↳ |.....|
40 00000270 03 00 00 00 00 08 f2 04 2a 00 00 00 10 00 00 00
    ↳ |.....*.....|
41 00000280 00 00 00 00 00 00 00 00 28 00 00 80 18 00 00 00
    ↳ |.....(.....|
42 00000290 28 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    ↳ |(.....|
43 000002a0 0c 00 00 00 38 00 00 00 18 00 00 00 02 00 00 00
    ↳ |...8.....|
44 000002b0 00 00 4c 05 00 00 01 00 2f 75 73 72 2f 6c 69 62 |..L
    ↳ ...../usr/lib|
45 000002c0 2f 6c 69 62 53 79 73 74 65 6d 2e 42 2e 64 79 6c |/
    ↳ libSystem.B.dyl|

```

```

46 000002d0 69 62 00 00 00 00 00 00 26 00 00 00 10 00 00 00 |ib
    ↳ .....&.....|
47 000002e0 68 40 00 00 08 00 00 00 29 00 00 00 10 00 00 00 |he
    ↳ .....).....|
48 000002f0 70 40 00 00 00 00 00 00 1d 00 00 00 10 00 00 00 |pe
    ↳ .....|
49 00000300 b0 40 00 00 18 01 00 00 00 00 00 00 00 00 00 00 |.e
    ↳ .....|
50 00000310 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    ↳ |.....|
51 00000320 00 00 00 00 00 00 00 00 ff 43 00 d1 ff 0f 00 b9
    ↳ |.....C.....|
52 00000330 68 00 80 52 e8 0b 00 b9 a8 00 80 52 e8 07 00 b9 |h..
    ↳ R.....R....|
53 00000340 e8 0b 40 b9 e9 07 40 b9 00 01 09 0b ff 43 00 91 |..@
    ↳ ...@.....C..|
54 00000350 c0 03 5f d6 01 00 00 00 1c 00 00 00 00 00 00 00 |.._
    ↳ .....|
55 00000360 1c 00 00 00 00 00 00 00 1c 00 00 00 02 00 00 00
    ↳ |.....|
56 00000370 28 03 00 00 40 00 00 00 40 00 00 00 54 03 00 00
    ↳ |(...@...@...T...|
57 00000380 00 00 00 00 40 00 00 00 00 00 00 00 00 00 00 00
    ↳ |...@.....|
58 00000390 00 00 00 00 03 00 00 00 0c 00 01 00 10 00 01 00
    ↳ |.....|
59 000003a0 00 00 00 00 00 10 00 02 00 00 00 00 00 00 00 00
    ↳ |.....|
60 000003b0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    ↳ |.....|
61
62 *
63 00004000 00 00 00 00 20 00 00 00 30 00 00 00 30 00 00 00
    ↳ |.... ...0...0...|
64 00004010 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00
    ↳ |.....|
65 00004020 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    ↳ |.....|
66 00004030 00 00 00 00 00 00 00 00 00 01 5f 00 12 00 00 00
    ↳ |....._.....|
67 00004040 00 02 00 00 00 03 00 a8 06 00 00 02 5f 6d 68 5f
    ↳ |....._mh_|
68 00004050 65 78 65 63 75 74 65 5f 68 65 61 64 65 72 00 09 |
    ↳ execute_header..|
69 00004060 6d 61 69 6e 00 0d 00 00 a8 06 00 00 00 00 00 00 |
    ↳ main.....|
70 00004070 02 00 00 00 0f 01 10 00 00 00 00 00 01 00 00 00
    ↳ |.....|
71 00004080 16 00 00 00 0f 01 00 00 28 03 00 00 01 00 00 00
    ↳ |.....(.....|
72 00004090 20 00 5f 5f 6d 68 5f 65 78 65 63 75 74 65 5f 68 |.
    ↳ __mh_execute_h|
73 000040a0 65 61 64 65 72 00 5f 6d 61 69 6e 00 00 00 00 00 |
    ↳ eader._main.....|
74 000040b0 fa de 0c c0 00 00 01 11 00 00 00 01 00 00 00 00
    ↳ |.....|

```

```

75 000040c0 00 00 00 14 fa de 0c 02 00 00 00 fd 00 02 04 00
    ↪ |.....|
76 000040d0 00 02 00 02 00 00 00 5d 00 00 00 58 00 00 00 00
    ↪ |.....]...X...|
77 000040e0 00 00 00 05 00 00 40 b0 20 02 00 0c 00 00 00 00
    ↪ |.....@. ....|
78 000040f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    ↪ |.....|
79
80 *
81 00004110 00 00 00 2c 00 00 00 00 00 00 00 01 6d 61 69 6e
    ↪ |...,.....main|
82 00004120 00 1b 34 0e f5 0e 6a a2 a6 9d c2 8a df f8 2b 80
    ↪ |..4...j.....+.|
83 00004130 3e 5c 27 2b 4d 63 c6 36 90 4c d6 80 e4 12 08 fc
    ↪ |>\'+Mc.6.L.....|
84 00004140 af ad 7f ac b2 58 6f c6 e9 66 c0 04 d7 d1 d1 6b
    ↪ |.....Xo..f.....k|
85 00004150 02 4f 58 05 ff 7c b4 7c 7a 85 da bd 8b 48 89 2c |.0X
    ↪ ..|. |z....H.,|
86 00004160 a7 ad 7f ac b2 58 6f c6 e9 66 c0 04 d7 d1 d1 6b
    ↪ |.....Xo..f.....k|
87 00004170 02 4f 58 05 ff 7c b4 7c 7a 85 da bd 8b 48 89 2c |.0X
    ↪ ..|. |z....H.,|
88 00004180 a7 ad 7f ac b2 58 6f c6 e9 66 c0 04 d7 d1 d1 6b
    ↪ |.....Xo..f.....k|
89 00004190 02 4f 58 05 ff 7c b4 7c 7a 85 da bd 8b 48 89 2c |.0X
    ↪ ..|. |z....H.,|
90 000041a0 a7 66 32 27 13 70 79 29 4b 7b 1c 83 53 97 b1 da |.f2
    ↪ '.py)K{..S...|
91 000041b0 09 de 2e 98 15 a8 bb 2c 87 ec eb 22 55 ca e6 0c
    ↪ |.....,"U...|
92 000041c0 40 00 00 00 00 00 00 00
    ↪ .....|
93 000041c8

```

- **Spiegazione:** Questo dump esadecimale (generato in formato Mach-O a 64-bit per Apple Silicon Mac) rappresenta l'eseguibile completo. All'offset 00000320 potete notare i reali byte macchina binari che abbiamo disassemblato nella fase precedente (es. ff 43 00 d1, ff 0f 00 b9, ecc.), ora inseriti in una struttura file strutturata insieme a metadati di caricamento, percorsi per il caricatore dinamico delle librerie (/usr/lib/dyld e /usr/lib/libSystem.B.dylib) e tabelle dei simboli generati dal linker.

1.7.3 Il Problema del Mondo Esterno (stdio.h)

La CPU da sola conosce solo l'aritmetica ed i registri; non possiede alcuna nozione su come accedere al monitor, formattare caratteri grafici o leggere dalla tastiera. Per fare ciò, il programma deve interagire con il Sistema Operativo richiedendo l'ausilio delle librerie di sistema.

Il linguaggio C è improntato alla massima efficienza: non include di default alcuna funzione che non sia stata espressamente richiesta. Per abilitare le funzioni di Input/Output occorre includere l'intestazione standard tramite la direttiva `\{\}#include <stdio.h>`.

1.8 Anatomia dell'Hello World e i Commenti nel Codice

1.8.1 Anatomia del Primo Programma C Reale

Esaminiamo ora il classico primo programma C, arricchito dai commenti esplicativi:

```

1 // Questo ? un commento a riga singola (introdotto nello standard C99)
2 #include <stdio.h> /* Direttiva al preprocessore per l'I/O standard */
3
4 /*
5     Questo ? un commento multi-riga.
6     I commenti vengono totalmente ignorati dal compilatore
7     ed eliminati nella prima fase dal Preprocessore!
8 */
9 int main(void) {
10     printf("Benvenuti a Programmazione 1!\n"); // Stampa la stringa su
        ↪ schermo
11     return 0; // Restituisce il valore 0 al Sistema Operativo (Successo)
12 }
```

Suggerimento

Curiosità: Il Preprocessore in Hello World Per completezza, provate a lanciare il preprocessore su questo file che include <stdio.h> eseguendo:

```
1 cc hello.c -E -o hello.i
```

Se aprite il file `hello.i` risultante con un editor di testo, noterete che contiene migliaia di righe di codice (spesso più di 20.000 righe). Questo accade perché la direttiva `\{\}#include <stdio.h>` viene espansa dal preprocessore che copia letteralmente nel file tutte le definizioni e le dichiarazioni necessarie alla libreria di input/output standard prima dell'avvio del compilatore.

1.8.2 Sintassi dei Commenti in C

- **// Commento a riga singola:** tutto ciò che si trova dopo i due slash sulla medesima riga viene ignorato.
- **/* ... */ Commento multi-riga:** ignora tutto il testo compreso tra il simbolo di apertura `/*` ed il simbolo di chiusura `*/`, anche su più righe.

1.8.3 Il Punto e Virgola (;) come Terminatore

Nelle righe di codice `printf(...)` e `return 0;`, noterete la presenza costante del punto e virgola `;`.

- In C, il punto e virgola non è un separatore di istruzioni (come in altri linguaggi), ma un **terminatore di istruzione** (statement terminator).
- Indica al compilatore che un'istruzione è formalmente conclusa. Dimenticarlo è uno degli errori di sintassi più comuni e genera immediati errori in fase di compilazione.

1.8.4 Whitespace e Formattazione Libera (Free-form)

Il compilatore C ignora completamente tutti i caratteri invisibili come spazi, tabulazioni e ritorni a capo (detti **whitespace**), ad eccezione di quelli inseriti all'interno dei letterali stringa (il testo delimitato da virgolette "..."). Teoricamente, potreste scrivere l'intero programma `main` su un'unica riga:

```
1 int main(void) { printf("Benvenuti!"); return 0; }
```

Il compilatore produrrebbe lo stesso identico eseguibile binario nativo.

1.8.5 L'Importanza dell'Indentazione

Se il compilatore ignora gli spazi, perché ci preoccupiamo di organizzare il codice su più righe e con rientri precisi?

- **Per la leggibilità umana:** L'indentazione (l'uso di spazi o tabulazioni per rientrare ad ogni blocco racchiuso tra parentesi graffe `{ } { }`) è una convenzione fondamentale e una questione di stile condivisa.
- Consente a voi e ai vostri colleghi di leggere e interpretare all'istante la struttura logica del codice. Scrivere codice privo di indentazione rende la diagnosi degli errori estremamente complessa e pruriginosa.

1.8.6 Verifica dell'Hello World con `echo $?`

Per compilare il nostro codice utilizzeremo il comando `gcc`.

Suggerimento

Prendere buone abitudini: Il flag `-Wall` Quando compilate, abituatevi fin dal primo giorno ad aggiungere il flag `-Wall` (*Warnings: All*):

```
1 gcc -Wall hello.c -o hello
```

Questo comando dice al compilatore di non limitarsi a segnalare errori bloccanti, ma di avvisarvi con dei messaggi di avvertimento (*warnings*) su comportamenti strani o rischiosi nel vostro codice. Risparmierete ore di debugging!

Eseguiamo il programma appena generato:

```
1 ./hello
```

Output a schermo:

```
1 Benvenuti a Programmazione 1!
```

Se ora verifichiamo l'Exit Status restituito al Sistema Operativo:

```
1 echo $?
```

Il terminale risponderà 0.

Nota

Convenzione di Successo In tutti i sistemi operativi POSIX (Linux, macOS, Unix), l'Exit Status 0 rappresenta la convenzione universale per indicare che il programma è stato completato con **successo e senza errori**. Qualsiasi valore diverso da zero (es. 1, 2, 139) indica un codice di errore specifico.

1.9 Quiz di Autovalutazione

Mettetevi alla prova con questo test rapido per verificare la vostra comprensione dei concetti di questa lezione. Cliccate su ciascuna domanda per verificare la risposta corretta e la spiegazione.

Quiz 1

Domanda

1. Nell'architettura di von Neumann, quale componente si occupa di coordinare l'intero sistema recuperando le istruzioni dalla memoria ed eseguendole?

- A) La memoria RAM.
- B) L'Unità di Controllo (Control Unit - CU) all'interno della CPU.
- C) L'Unità Aritmetico Logica (ALU).
- D) Il bus di sistema per il trasferimento dei dati.

Soluzione
Risposta corretta: B
Spiegazione: La Control Unit (CU) recupera le istruzioni dalla RAM tramite il ciclo di Fetch-Decode-Execute e coordina le altre componenti hardware del sistema.

Quiz 2

Domanda

2. Dove risiedono le variabili ed il codice di un programma mentre è in corso di esecuzione?

- A) Esclusivamente all'interno dei registri della CPU per tutta la durata del programma.
- B) Nella memoria RAM.
- C) Sull'hard disk o SSD.

- D) All'interno del file header `stdio.h`.

Soluzione
Risposta corretta: B
Spiegazione: La RAM ospita sia il codice del programma che i dati dinamici (variabili) durante il tempo di esecuzione (*runtime*). I registri contengono solo i dati correnti calcolati in quel preciso istante.

Quiz 3

Domanda

3. Qual è la differenza principale tra un linguaggio compilato (come il C) ed uno interpretato (come Python)?

- A) I linguaggi compilati sono tradotti in codice macchina nativo prima dell'esecuzione; quelli interpretati sono letti riga per riga a runtime da un programma interprete.
- B) I linguaggi compilati non possono gestire input dall'utente.
- C) I linguaggi interpretati sono più veloci nell'esecuzione su microcontrollori.
- D) I linguaggi compilati non necessitano di una funzione `main`.

Soluzione
Risposta corretta: A
Spiegazione: Il compilatore C traduce in blocco il codice in linguaggio macchina proprietario della CPU. Un interprete invece legge e converte il codice riga per riga durante l'esecuzione, introducendo un sovraccarico di prestazioni.

Quiz 4

Domanda

4. Che cos'è il linker all'interno della toolchain di compilazione?

- A) Il software che rileva gli errori di sintassi nel codice sorgente.
- B) Il programma che unisce il codice oggetto generato dal compilatore con le librerie esterne precompilate per creare un eseguibile finale.
- C) L'editor di testo utilizzato per scrivere il codice.
- D) La componente hardware che sposta i dati tra i registri ed il bus.

Soluzione
Risposta corretta: B
Spiegazione: Il linker unisce i file oggetto (.o) prodotti dal compilatore ed eventuali librerie esterne (es. stdio del C) risolvendo tutti i riferimenti agli indirizzi delle funzioni per generare il file eseguibile finale.

Quiz 5

Domanda

5. A cosa serve la direttiva `\{\}\#include <stdio.h>` posta in cima a un programma C?

- A) A ordinare al compilatore di allocare 8 byte di memoria nello Stack.
- B) A importare il prototipo delle funzioni della libreria standard di Input/Output (come `printf`).
- C) A dichiarare la funzione principale `main`.
- D) A azzerare il valore di ritorno del programma.

Soluzione
Risposta corretta: B
Spiegazione: `\{\}\#include <stdio.h>` è una direttiva del preprocessore che inserisce l'header file `stdio.h`, contenente le dichiarazioni delle funzioni base per interagire con lo schermo e la tastiera.

Quiz 6

Domanda

6. Cosa significa il parametro `void` tra le parentesi tonde in `int main(void)`?

- A) Che il programma restituisce un valore nullo.
- B) Che la funzione `main` non accetta alcun argomento o parametro in ingresso.
- C) Che il programma contiene un errore di sintassi.
- D) Che la funzione principale alloca solo puntatori a memoria Heap.

Soluzione
Risposta corretta: B
Spiegazione: In C, specificare `void` all'interno della lista dei parametri di una funzione indica in modo esplicito che la funzione non accetta parametri.

Quiz 7

Domanda

7. In un sistema Unix/macOS, cosa indica un valore di exit status uguale a 0 restituito da un programma?

- A) Che il programma è crashato prima dell'esecuzione.
- B) Che il programma è terminato con successo e senza errori.
- C) Che non è stato stampato alcun testo a schermo.
- D) Che la memoria RAM è stata riempita completamente.

Soluzione
Risposta corretta: B
Spiegazione: Per convenzione POSIX, l'exit status 0 rappresenta la conclusione ottimale del programma. Qualsiasi valore diverso da zero indica codici di errore definiti dallo sviluppatore.

Quiz 8

Domanda

8. Quale comando da terminale permette di ispezionare l'exit status restituito dall'ultimo programma eseguito?

- A) `cat result`
- B) `echo $?`
- C) `gcc -o output`
- D) `pwd`

Soluzione
Risposta corretta: B
Spiegazione: Nei terminali Bash o Zsh, la variabile speciale `$?` contiene il codice di uscita dell'ultimo processo terminato. Usando `echo $?` se ne visualizza il valore a schermo.

1.10 Esercizi Pratici

Esercizio 1: Il Primo Eseguitibile

Domanda

Create un file chiamato `saluto.c`. Scrivete un programma in C che stampi a schermo il testo *"Benvenuti al corso di Programmazione 1 - A.A. 2026/2027!"* seguito da una nuova riga. Compilate il codice da terminale generando un eseguibile chiamato `saluto` ed

eseguitelo.

```
1 gcc saluto.c -o saluto
2 ./saluto
```

Comandi da lanciare nel terminale:

```
1 #include <stdio.h>
2
3 int main(void) {
4     printf("Benvenuti al corso di Programmazione 1 - A.A.
5     ↪ 2026/2027!\n");
6     return 0;
7 }
```

Codice sorgente saluto.c:

Soluzione

Esercizio 2: Ispezione dell'Exit Status di Errore

Domanda

Modificate l'istruzione `return 0;` del vostro programma dell'Esercizio 1 inserendo un valore di uscita a vostra scelta diverso da zero (ad esempio `return 42;`). Compilate nuovamente ed eseguite. Verificate tramite terminale che il sistema operativo riceva correttamente il nuovo codice di uscita personalizzato.

Il terminale risponderà stampando il valore 42.

```
1 gcc saluto.c -o saluto
2 ./saluto
3 echo $?
```

Comandi da terminale per verificare:

```
1 #include <stdio.h>
2
3 int main(void) {
4     printf("Benvenuti al corso di Programmazione 1 - A.A.
5     ↪ 2026/2027!\n");
6     return 42; // Restituiamo un codice di uscita personalizzato
7 }
```

Codice sorgente modificato:

Soluzione

Esercizio 3: Stampa Multilinea

Domanda

Scrivete un programma in C che stampi a schermo un rettangolo di asterischi di dimensione 5 colonne per 3 righe, utilizzando una sola chiamata alla funzione `printf`. *Suggerimento: ricordatevi di inserire il carattere speciale di ritorno a capo `\n` nei punti strategici della stringa.*

```
1 gcc rettangolo.c -o rettangolo
2 ./rettangolo
```

Comandi da lanciare nel terminale:

```
1 #include <stdio.h>
2
3 int main(void) {
4     printf("*****\n*****\n*****\n");
5     return 0;
6 }
```

Codice sorgente rettangolo.c:

Soluzione

Esercizio 4: Trovare il Comando nel System Path

Domanda

Utilizzando il terminale (su macOS o Linux), determinate in quale cartella di sistema si trova l'eseguibile del compilatore `gcc`. *Suggerimento: utilizzate il comando `which` oppure `whereis`.*

Spiegazione: Il comando `which` analizza le directory elencate nella variabile d'ambiente `$PATH` del sistema operativo e restituisce il percorso assoluto del primo file eseguibile corrispondente al nome specificato (es. `/usr/bin/gcc`).

```
1 which gcc
```

Comando da lanciare nel terminale:

Soluzione

Esercizio 5: Scrittura di Commenti e Codice

Domanda

Auto-esplicativo

Scrivete un programma minimale in C che non esegua alcuna stampa ma che restituisca al sistema operativo il codice di uscita 12. All'interno del file, inserite sia un commento a riga singola (`//`) sia un commento multilinea (`/* ... */`) per documentare lo scopo del programma. Compilate e verificate l'exit status tramite terminale.

Soluzione

Codice sorgente commenti.c:

```

1 #include <stdio.h>
2
3 /*
4  Corso di Programmazione 1
5  Questo programma dimostra l'uso dei commenti multilinea
6  e restituisce l'exit status 12.
7  */
8
9 int main(void) {
10     // Restituiamo 12 per simulare un exit status di errore
11     ↪ specifico
12     return 12;
13 }
```

Comandi da lanciare nel terminale:

```

1 gcc commenti.c -o commenti
2 ./commenti
3 echo $?
```

Il terminale risponderà stampando il valore 12.

Esercizio 6: Compilazione con Avvisi (Warnings)**Domanda**

Provate a compilare un file C in cui dichiarate una variabile senza usarla, forzando il compilatore a segnalare tutti gli avvisi possibili tramite l'opzione `-Wall`. Scrivete i comandi da lanciare nel terminale.

Spiegazione: Il flag `-Wall` attiva quasi tutti i messaggi di avvertimento (*warnings*) del compilatore. Nel compilatore GCC/Clang, compilando questo file apparirà un avviso simile a: `warning: unused variable 'x',`

```

1 gcc -Wall warning.c -o warning
```

Comando di compilazione:

```

1 #include <stdio.h>
2
3 int main(void) {
4     int x = 10; // Variabile dichiarata ma inutilizzata
5     return 0;
6 }
```

Codice sorgente `warning.c`:**Soluzione**

Esercizio 7: Compilazione di file sorgenti multipli

Domanda

Immaginate di avere due file sorgenti separati: `main.c` (contenente la funzione `main`) e `helper.c` (contenente funzioni di supporto). Scrivete la sequenza di comandi GCC per compilarli prima in file oggetto (.o) e poi collegarli (linking) in un unico file eseguibile chiamato `app`.

```
1 # Compila i singoli sorgenti in file oggetto senza fare il linking
  ↪ (-c)
2 gcc -c main.c -o main.o
3 gcc -c helper.c -o helper.o
4
5 # Collega i file oggetto in un unico eseguibile finalizzato
6 gcc main.o helper.o -o app
```

Comandi da lanciare nel terminale:

Soluzione

Esercizio 8: Errore nel preprocessore

Domanda

Scrivete un codice C volutamente errato in cui omettete il simbolo `\{\}\#` davanti alla direttiva `include <stdio.h>`. Compilatelo con `gcc` e analizzate l'errore del compilatore. Spiegate perché fallisce in fase di preprocessing.

Spiegazione: Il preprocessore analizza solo le righe che iniziano con `\{\}\#`. Senza questo carattere, la riga viene vista dal compilatore vero e proprio come normale codice C. Il compilatore cerca quindi di interpretare `include` come il tipo di una variabile o funzione, fallendo immediatamente poiché non riconosce la parola chiave.

Errore generato da GCC: error: unknown type name 'include' o error: expected '=', ',', ';', '...',

```
1 include <stdio.h> // Errore: manca il simbolo '\{\}\#'
2
3 int main(void) {
4     printf("Test errore\n");
5     return 0;
6 }
```

Codice sorgente errato `err\{\}\_preprocessore.c`:

Soluzione

Esercizio 9: La Sequenza di Escape per il Backslash

Domanda

Scrivete un programma in C che stampi a schermo il percorso assoluto tipico di Windows: C:\Program Files\MyCode\ . Spiegate come gestire i caratteri di backslash per evitare che vengano interpretati come sequenze di escape (es. \P, \M).

Spiegazione: In C, il carattere backslash \ è un carattere di escape che introduce comandi speciali (come \n per andare a capo o \t per tabulare). Per stampare a schermo un carattere \ letterale, è necessario effettuarne l'escape raddoppiandolo: \\.

```
1 gcc percorso.c -o percorso
2 ./percorso
```

Comandi da lanciare nel terminale:

```
1 #include <stdio.h>
2
3 int main(void) {
4     // Usiamo il doppio backslash '\\' per dire a printf di
5     printf("C:\\Program Files\\MyCode\\n");
6     return 0;
7 }
```

↪ stampare un singolo '\\', testuale.

Codice sorgente percorso.c:

Soluzione