



Programmazione 1 (Canale O-Z)

Lezione 1: Introduzione alla Programmazione in C

Docente: Prof. Antonino Furnari

Email: `antonino.furnari@unict.it`

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



Dalla Fisica all'Astrazione

Il Ruolo del Programmatore Oggi

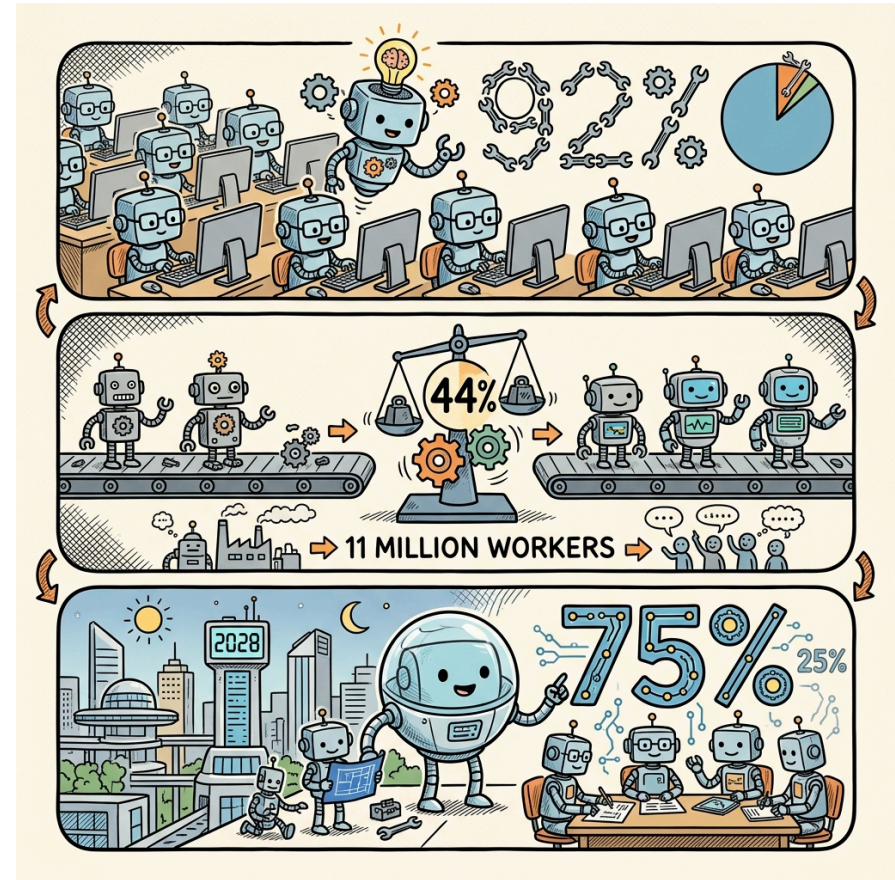
- **Pensiero Algoritmico:** Modellare problemi complessi reali in istruzioni deterministiche.
- **Motore Invisibile:** Climatologia, bioinformatica, medicina di precisione, finanza.
- **Non imparare sintassi a memoria:** Padroneggiare la logica sottostante.





L'Epoca dell'IA ed i Dati dell'Industria

- **GitHub Octoverse (2023):** 92% degli sviluppatori usa/sperimenta strumenti AI.
- **WEF Future of Jobs (2023):** 44% delle competenze trasformate in 5 anni (su 11.3M lavoratori).
- **Gartner (2024):** Entro il 2028, il 75% dei programmatori userà assistenti AI.



Perché Imparare a Programmare?

L'Informatica è il Linguaggio della Rivoluzione

- **IA ed Algoritmi:** L'IA non è magia; si basa su strutture dati, memoria e algebra.
- **Guidare il Cambiamento:** Imparare la grammatica dell'informatica per non subire la tecnologia.

? Riprendiamo la domanda iniziale: chi controlla il codice generato, se nessuno sa leggerlo?

L'Informatica è la Nuova Fisica

I Premi Nobel 2024

- **Nobel per la Fisica:** Hopfield & Hinton (apprendimento automatico e reti neurali).
- **Nobel per la Chimica:** Hassabis, Jumper & Baker (progettazione proteica e **AlphaFold**).



La Metafora del Direttore d'Orchestra

Agentic Engineering

- **Il Programmatore come Direttore:** Coordinare e supervisionare agenti IA autonomi.
- **Visione d'Insieme:** Conoscere la musica e lo spartito per evitare il caos.



⚙️ L'Evoluzione del Calcolo: L'Informatica Meccanica

La Pascalina (1642)

- **Algoritmo nel Metallo:** Operazioni aritmetiche rappresentate da ingranaggi e ruote numerate.
- **Riprogrammazione fisica:** Per cambiare il calcolo occorreva riprogettare e smontare i ruotismi meccanici.



⚙️ L'Evoluzione del Calcolo: L'Elettronica delle Valvole

ENIAC e la Fisicità del Ricablaggio (Anni '40)

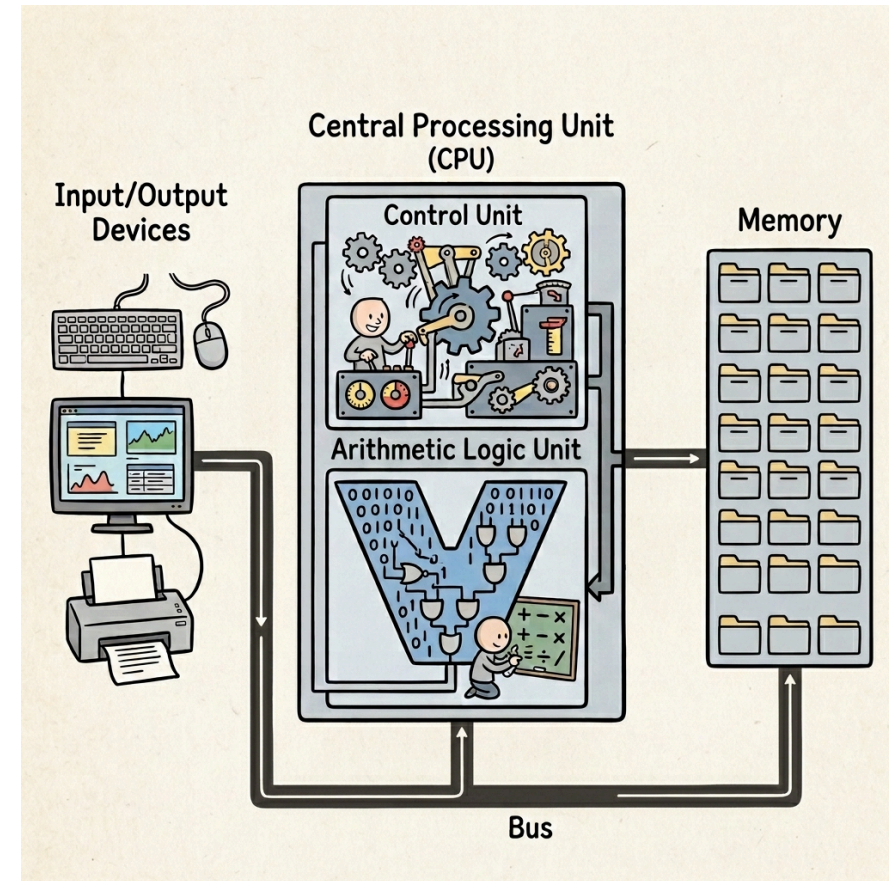
- **Valvole Termoioniche:** Calcoli elettronici infinitamente più veloci.
- **Riprogrammare = Ricablare:** Per cambiare programma occorreva scollegare e ricollegare fisicamente centinaia di cavi in pannelli telefonici.



L'Architettura di Von Neumann (1945)

Stored Program Concept & Macchina Universale

- **Programma Memorizzato:** Istruzioni e dati nello stesso spazio RAM.
- **CPU vs RAM:** Hardware immutabile, software flessibile.





I Livelli di Astrazione

Dal Silicio al C

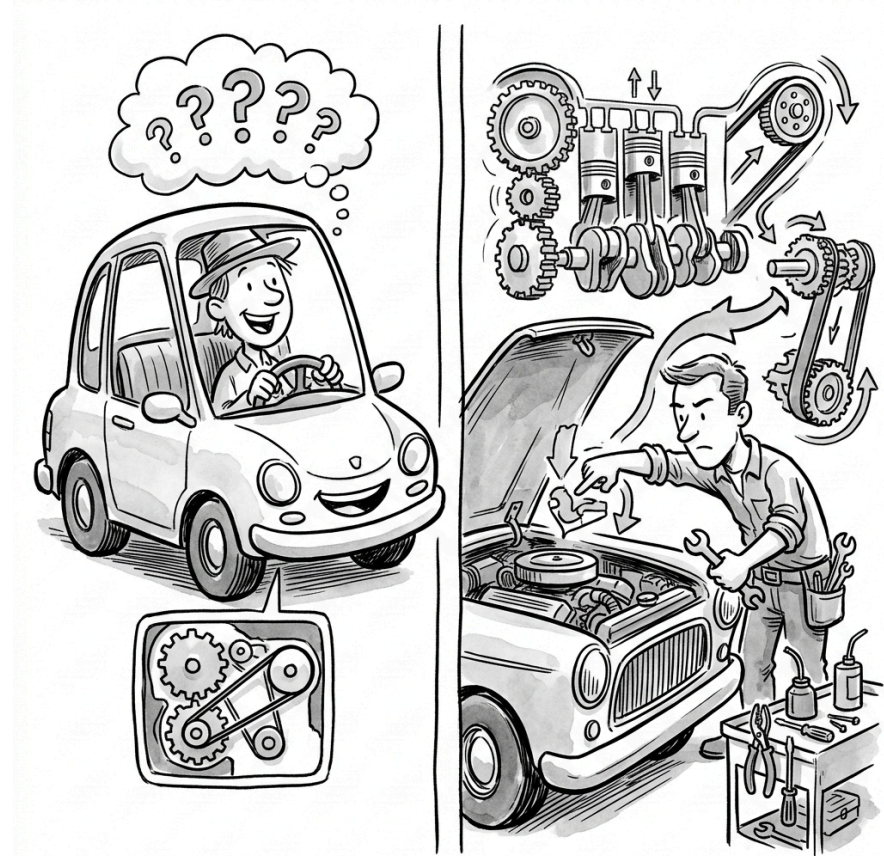
- **Codice Macchina (0 ed 1):** Tensioni elettriche.
- **Assembly:** Istruzioni mnemoniche hardware-specific.
- **Linguaggio C:** Astrazione algebrica vicina al pensiero umano.



La Metafora dell'Automobile e del Meccanico

Scegliere il Giusto Livello di Astrazione

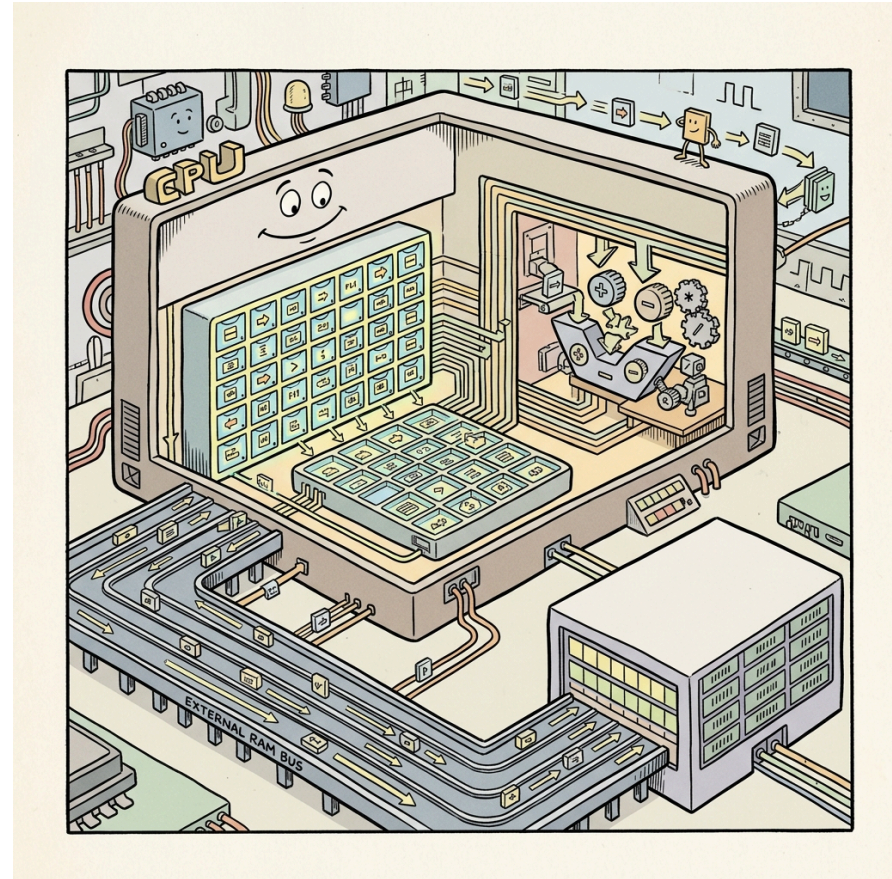
- **Guidatore (Alto Livello):** Volante e pedali per viaggiare senza conoscere il motore.
- **Meccanico (Basso Livello):** Aprire il cofano per riparare, ottimizzare o progettare.



⚡ Registri CPU & Bottleneck

Le "Tasche" Veloci del Processore

- **Collo di Bottiglia:** RAM distante collegata da bus fisici.
- **Registri CPU:** Celle di memoria interne ultra-veloci per i calcoli ALU.



Dal Codice Umano all'Assembly

L'Illusione delle Variabili

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

- Per la CPU le lettere `x` e `y` **non esistono**: sono un'illusione del C per farci ragionare in modo algebrico.
- Il compilatore le trasforma in **celle di RAM** e **registri**.

? Cosa diventano queste 4 righe per la CPU? ➡ slide successiva

Assembly in Azione (Sintassi Intel)

```
push    rbp                ; Salva frame pointer precedente
mov     rbp, rsp           ; Imposta nuovo frame pointer
mov     DWORD PTR [rbp-4], 3 ; x = 3
mov     DWORD PTR [rbp-8], 5 ; y = 5
mov     edx, DWORD PTR [rbp-4] ; Carica x in edx
mov     eax, DWORD PTR [rbp-8] ; Carica y in eax
add     eax, edx           ; eax = x + y
pop     rbp               ; Ripristina frame pointer
ret                                ; Ritorna eax al Sistema Operativo
```

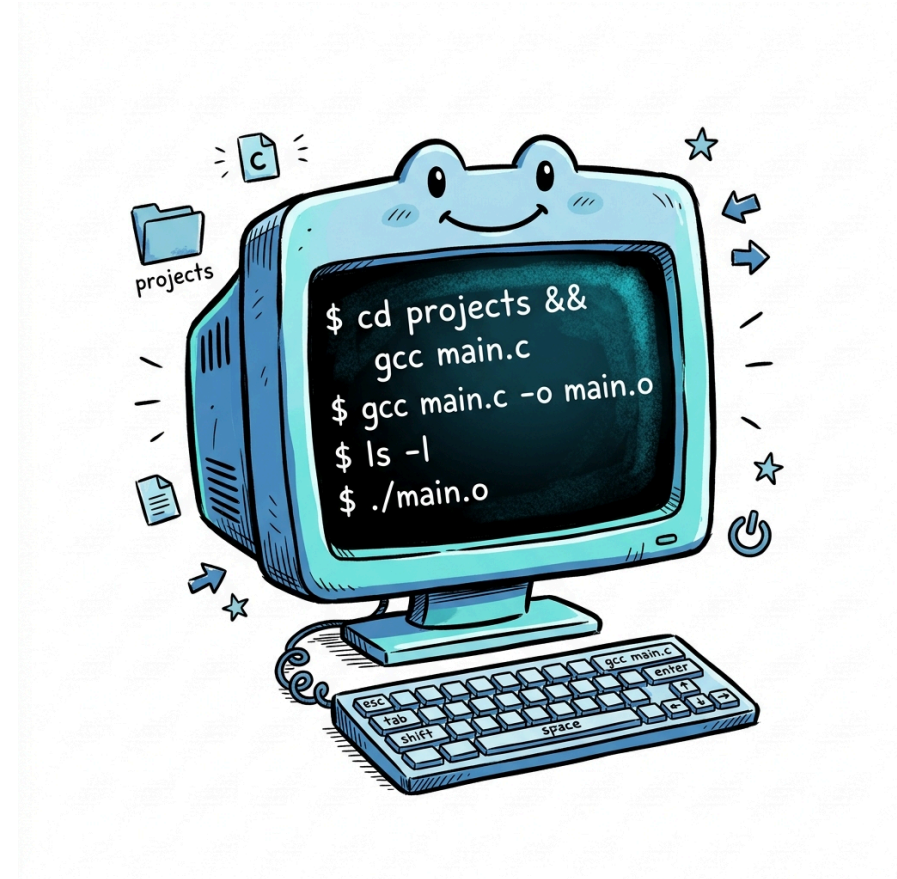
-  Sperimentate su godbolt.org!

 **Live:** incollare il codice su godbolt e mostrare la corrispondenza riga C ↔ istruzioni.

La Riga di Comando (Il Terminale)

La Porta d'Accesso al Sistema Operativo

- **Interfaccia Testuale:** Comunicare direttamente con l'OS senza l'uso del mouse.
- **Rinascita con l'AI:** Strumenti CLI e agenti di sviluppo autonomi si appoggiano ad essa.
- **Standard Unix:** Tratteremo shell Unix (`zsh` / `bash`). Su Windows useremo WSL.





Comandi Base della Shell Unix

Comando	Descrizione
<code>pwd</code>	Mostra la cartella corrente (<i>Print Working Directory</i>).
<code>ls</code>	Elenca i file e le cartelle nella directory corrente.
<code>cd <dir></code>	Spostati nella cartella specificata (<i>Change Directory</i>).
<code>cd ..</code>	Torna indietro di una cartella.
<code>./<prog></code>	Esegue un programma/comando nella cartella corrente.
Tab	Autocompleta nomi di file e cartelle.
Su / Giù	Scorri la cronologia dei comandi eseguiti.
<code>clear</code> / <code>Ctrl + L</code>	Pulisce lo schermo del terminale.

 **Live:** aprire il terminale, `pwd`, `ls`, creare la cartella della lezione e spostarsi con `cd`.

Pratica al Terminale: il Mistero del Silenzio

1. Scriviamo `somma.c` (il codice di prima)

2. Compiliamo (sempre con `-Wall`):

```
gcc -Wall somma.c -o somma
```

3. Eseguiamo:

```
./somma
```

➡ Non appare nulla!

- Il flag `-Wall` (*Warnings: All*): il compilatore segnala anomalie e potenziali bug non bloccanti.
- **Perché il silenzio?** Nessun `printf` = nessun output a schermo.

? Dove è finito il risultato di `x + y`?



Interrogare il Sistema Operativo: `echo $?`

4. Chiediamo al Sistema Operativo l'ultimo valore di ritorno:

```
echo $?
```

```
8
```

- `return x + y;` (cioè `ret` in Assembly) ha consegnato `8` al **Sistema Operativo**.
- `echo $?` stampa l'**Exit Status** dell'ultimo programma terminato.
- **Convenzione POSIX:** `0` = successo, `≠ 0` = codice di errore.



Live: cambiare i valori di `x` e `y`, ricompilare e rieseguire `echo $?`.

Dai Traduttori all'Hello World: Interpreti

Linguaggi Interpretati (Traduzione Simultanea)

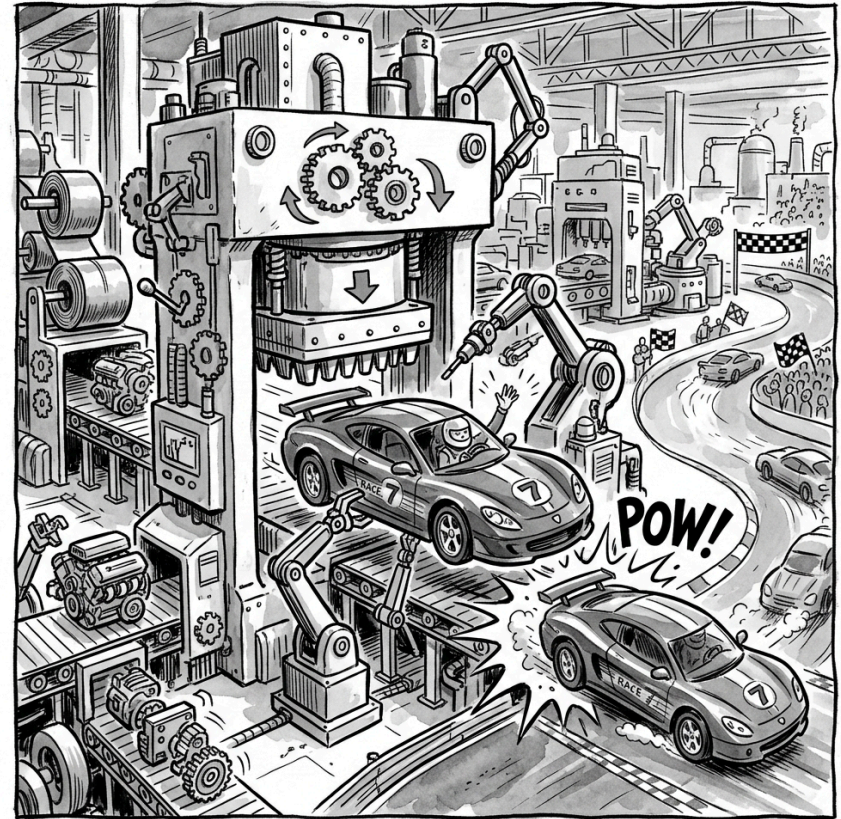
- **Interprete in Runtime:** Legge ed esegue il codice sorgente riga per riga (es. Python, JavaScript).
- **Flessibilità:** Traduzione simultanea durante l'esecuzione del programma.



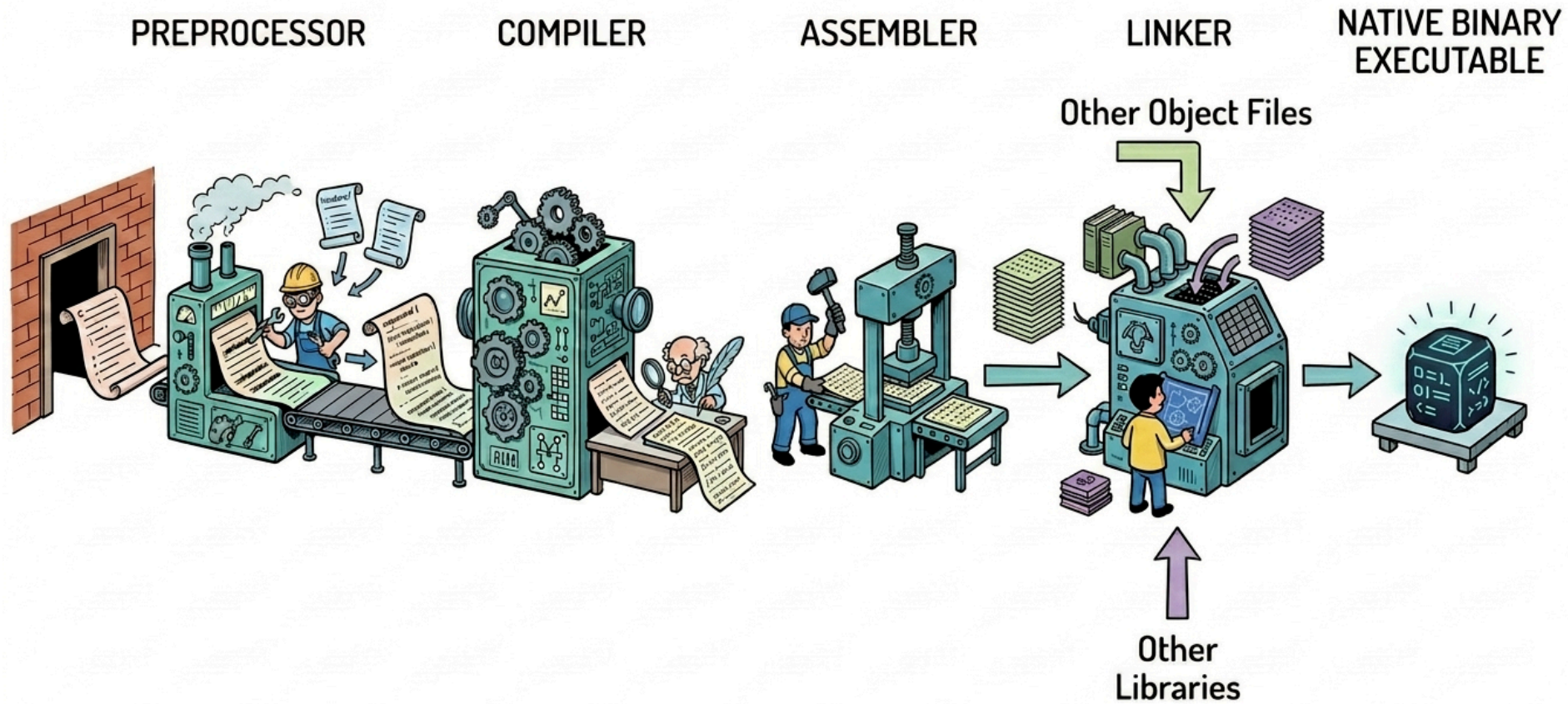
Dai Traduttori all'Hello World: Compilatori

Linguaggi Compilati (La Fabbrica del C)

- **Compilatore in Anticipo:** Analizza e traduce l'intero sorgente in un file binario eseguibile nativo (es. C, C++).
- **Efficienza Massima:** Esecuzione ultra-veloce e diretta sull'hardware.



Toolchain del Compilatore C (4 Fasi)



Toolchain: il Nostro Esempio (`main.c`)

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

- Lo stesso codice della somma, salvato come `main.c`.
- **Niente** `#include`: così l'output di ogni fase resta leggibile.
- Fermeremo `gcc` dopo **ciascuna** delle 4 fasi per ispezionare cosa produce.

Fase 1 — Preprocessore (`cpp`) → `main.i`

Espande le direttive (`#include` , `#define`) ed elimina i commenti.

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

```
# 1 "main.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 482 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "main.c" 2
int main(void) {
    int x = 3;
    int y = 5;
    return x + y;
}
```

- Le righe `#` sono *linemarkers*. Nessun header incluso ➡ il codice resta intatto.

Fase 2 — Compilatore → `main.s` (Assembly)

Traduce il C pre-elaborato in Assembly per la CPU (qui ARM64 / Apple Silicon).

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

```
_main:                                ; @main
    ...                               ; (direttive .section, .globl, .cfi_* omesse)
    sub    sp, sp, #16                ; alloca spazio sullo stack
    str    wzr, [sp, #12]
    mov    w8, #3                     ; x = 3
    str    w8, [sp, #8]
    mov    w8, #5                     ; y = 5
    str    w8, [sp, #4]
    ldr    w8, [sp, #8]                ; carica x nel registro w8
    ldr    w9, [sp, #4]                ; carica y nel registro w9
    add    w0, w8, w9                  ; w0 = x + y (registro di ritorno)
    add    sp, sp, #16                ; libera lo stack
    ret
```

 **Live:** aprire `main.s` completo e mostrare le direttive per l'assembler (`.section`, `.globl`, ...).

Fase 3 — Assembler (`as`) → `main.o` (Oggetto)

Traduce l'Assembly testuale in un file oggetto **binario** (non ancora eseguibile).

```
gcc -c main.s -o main.o
objdump -d main.o      # disassembliamo per poterlo leggere
```

```
0000000000000000 <tmp0>:
   0: d10043ff      sub     sp, sp, #0x10
   4: b9000fff      str     wzr, [sp, #0xc]
   8: 52800068      mov     w8, #0x3          ; =3
  c: b9000be8      str     w8, [sp, #0x8]
 10: 528000a8      mov     w8, #0x5          ; =5
  ...
 20: 0b090100      add     w0, w8, w9
 24: 910043ff      add     sp, sp, #0x10
 28: d65f03c0      ret
```

- A sinistra: **indirizzi** e **codici operativi** esadecimali che la CPU esegue. A destra: l'istruzione mnemonica.

Fase 4 — Linker (`ld`) → `main` (Eseguibile)

Unisce file oggetto e librerie di sistema nell'eseguibile finale.

```
gcc main.o -o main
./main
echo $?           # 8
```

Ispezioniamo il binario in esadecimale:

```
hexdump -C main
```

 **Live:** l'output è lungo centinaia di righe, lo guardiamo insieme al terminale. Cerchiamo:

- i byte macchina della Fase 3 (`ff 43 00 d1 ...`);
- i percorsi delle librerie (`/usr/lib/dyld`, `libSystem.B.dylib`) aggiunti dal linker.

Toolchain: Riepilogo

Fase	Comando	Produce
1. Preprocessore	<code>gcc -E main.c -o main.i</code>	C espanso (<code>.i</code>)
2. Compilatore	<code>gcc -S main.i -o main.s</code>	Assembly (<code>.s</code>)
3. Assembler	<code>gcc -c main.s -o main.o</code>	Oggetto binario (<code>.o</code>)
4. Linker	<code>gcc main.o -o main</code>	Eseguibile

- Di solito basta **un solo comando**: `gcc -Wall main.c -o main` esegue tutte e 4 le fasi.

Il Problema del Mondo Esterno (`stdio.h`)

- La CPU conosce solo **aritmetica e registri**: non sa nulla di monitor o tastiera.
- Per stampare serve l'aiuto del **Sistema Operativo** tramite le **librerie di sistema**.
- Il C non include nulla che non sia richiesto: per l'I/O serve

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

? Ora possiamo finalmente far *parlare* il nostro programma: come?

Anatomia dell'Hello World & Commenti

```
// Commento a riga singola (C99+)
#include <stdio.h> /* Libreria standard di I/O */

/* Commento multi-riga:
   eliminato dal preprocessore! */
int main(void) {
    printf("Benvenuti a Programmazione 1!\n");
    return 0; // Exit status 0 (Successo)
}
```

 **Live:** compilare **sempre con** `-Wall` ed eseguire, poi verificare l'exit status:

```
gcc -Wall hello.c -o hello && ./hello ➡ Benvenuti a Programmazione 1!
```

```
echo $? ➡ 0
```



Regole di Scrittura del Codice C

- **Commenti:** `//` fino a fine riga, `/*` ... `*/` su più righe. Il preprocessore li elimina.
- **Punto e virgola** `;`: è un **terminatore** di istruzione. Dimenticarlo = errore di compilazione.
- **Whitespace:** spazi, tab e a capo sono ignorati (tranne dentro `" ... "`).
- **Indentazione:** non serve al compilatore, serve **a noi** per leggere la struttura.

```
int main(void) { printf("Benvenuti!"); return 0; } // compila, ma è illeggibile
```

 **Live:** togliere un `;` e leggere insieme l'errore di `gcc`. Poi `gcc -E hello.c -o hello.i`: quante righe produce `#include <stdio.h>`?

Cosa Abbiamo Imparato Oggi

- Il programmatore guida l'IA: servono **fondamenti** per verificare ciò che produce.
- Von Neumann: **programma e dati** nella stessa memoria.
- Il C è un'**astrazione**: le variabili diventano RAM e registri.
- Toolchain: **preprocessore** → **compilatore** → **assembler** → **linker**.
- `gcc -Wall file.c -o prog, ./prog, echo $?.`

 **Per casa:** esercizi della Lezione 1 sul sito del corso.

Grazie per l'attenzione!

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