Programmazione 2

Lezione 1 | 23 Febbraio

Università degli Studi di Modena e Reggio Emilia — UnimoreE

1. Il linguaggio C: caratteristiche fondamentali

Il linguaggio C è caratterizzato da tre proprietà fondamentali che lo distinguono dai linguaggi moderni ad alto livello.

1.1. Procedurale

Il programma C è **procedurale**: è diviso in funzioni. Non esistono oggetti né classi — esistono **dati**, **memoria** e **procedure** (funzioni).

Funzione in C

Una funzione è un blocco di codice che esegue una specifica operazione. Può essere richiamata da altre parti del programma, accetta *argomenti* (input) e restituisce un *valore di ritorno* (output).
A basso livello, una funzione è fondamentalmente un **indirizzo di memoria** a cui il programma salta per eseguire la procedura.

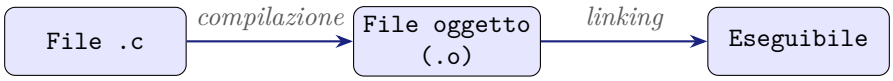
Nota

È possibile emulare pattern di programmazione a oggetti anche in C puro, senza usare le funzionalità del C++. Questo è esattamente quello che viene fatto nel **kernel di Linux**.

1.2. Compilato

Il codice C viene **compilato** e trasformato in linguaggio macchina. Questo consente ottimizzazioni che un sistema interpretato non può fare: il compilatore ha l'intero sorgente di fronte a sé e può, ad esempio, eliminare completamente rami `else` che non verranno mai eseguiti.

1.2.1. Le due fasi della compilazione



Fase	Cosa succede
Compilazione	Il file `.c` viene trasformato in un file oggetto mappato in codice macchina
Linking	Il file oggetto viene collegato alle librerie necessarie per creare l'eseguibile finale

1.3. Dichiarativo e tipizzato

Ogni variabile ha un **tipo fisso** che non può cambiare. Lo sviluppatore deve sempre dichiarare il tipo prima di usare una variabile.

Perché? Per l'efficienza della memoria: scegliendo il tipo giusto per il dato giusto, non si spreca spazio. In Python, anche un semplice intero è un oggetto completo in memoria — molto più pesante.

2. I compilatori

2.1. GCC — GNU Compiler Collection

Il compilatore che utilizziamo nel corso. È una **toolchain** (catena di strumenti):

Componente	Ruolo
GCC (GNU Compiler Collection)	Il compilatore effettivo
binutils	Crea i codici binari: trasforma il codice C in oggetto, poi in eseguibile
C Library	Tutte le funzioni standard di C, implementate in C
GDB (GNU Debugger)	Il debugger

GCC è **multi-piattaforma** e indipendente dall'architettura. Esiste anche **MSVC** (Microsoft Visual C++) con caratteristiche simili.

2.2. Compilatori proprietari

Esistono compilatori specializzati per specifiche architetture hardware: `arm-gcc`, `intel-gcc`, `cuda`, ecc.

Chi costruisce l'architettura conosce dettagli hardware non pubblici e li sfrutta per ottenere codice più performante. Anche un miglioramento del 5–10% può essere molto rilevante in certi contesti.

2.3. GDB — Il Debugger

GDB (GNU Debugger) permette di eseguire un programma **step by step**: si avanza istruzione per istruzione e si ispezionano i valori delle variabili a runtime.

Introspezione

La capacità di capire *perché* una variabile cambia valore e *come* il programma si comporta nel dettaglio durante l'esecuzione.

3. Versioni del linguaggio C

Anno	Versione	Note
1973	—	Proposta del linguaggio da parte di **Dennis Ritchie**
1989	C89 / C90	Prima standardizzazione
1999	**C99**	La più usata attualmente
2011	C11	Aggiunge funzionalità che simulano il C++
2018	C18	Revisione di C11

Attenzione

Non tutti i compilatori implementano tutti gli standard, e non è detto che l'hardware su cui si compila supporti gli standard più recenti. Per questo si tende a essere **conservativi** e usare C99. Usare C11 o C18 potrebbe essere problematico su alcuni sistemi. Lo standard si imposta direttamente come flag al compilatore.

4. Sviluppare in C: le insidie

In C non c'è un interprete che gestisce la memoria (come Python) né un ambiente virtuale (come Java). Il programma fa letteralmente **qualsiasi cosa** gli si dica — incluse cose sbagliate.

Il problema più subdolo non è quando il programma crasha in modo evidente: è quando dà errori **non deterministici**, corrompendo dati senza far esplodere nulla immediatamente.

4.1. Le insidie principali

- **Corruzione di memoria:** se il programma non si comporta come progettato, può sovrascrivere dati in memoria
- **Stack overflow:** errori nella gestione dello stack
- **Cast pericolosi:** trattare un tipo di memoria come se fosse un altro (es. allocare 1 byte e poi leggerlo come se fosse una word a 32 bit)

4.2. Buone pratiche

- Controllare sempre **input** e **valori di ritorno**
- Usare funzioni "sicure" (deterministiche sulla memoria)
- **Non sottovalutare i warning**

4.3. Errori vs Warning

Tipo	Cosa significa	Cosa fare
Errore fatale	Il compilatore non riesce a compilare il codice	Devi fixarlo per forza
Warning	Il codice viene compilato, ma ci sono situazioni potenzialmente pericolose	Non ignorarli

Attenzione

I cast tra tipi generano warning. I warning non sono decorativi: bisogna avere piena consapevolezza di quello che si sta facendo.

5. Hello World: analisi riga per riga

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

Listing 1: Hello World in C

5.1. `#include <stdio.h>`

- `#` → direttiva al **preprocessore** (eseguita *prima* della compilazione vera e propria)
- `include` → importa un file
- `<stdio.h>` → file cercato nei percorsi standard del sistema
  - `stdio` = **Standard Input/Output**
  - `.h` = **header file**: contiene le dichiarazioni delle funzioni di una libreria
  - Il preprocessore prende il contenuto del file e lo "incolla" nel sorgente

Nota

Per includere librerie personali si usa `"percorso/relativo"` invece delle parentesi angolari `< >`.

5.2. `int main()`

- `int` → tipo del valore di ritorno
- `main` → nome della funzione: è **la prima funzione eseguita dal programma**
- `()` → nessun argomento (in questa versione semplice)
- `{ ...}` → corpo della funzione

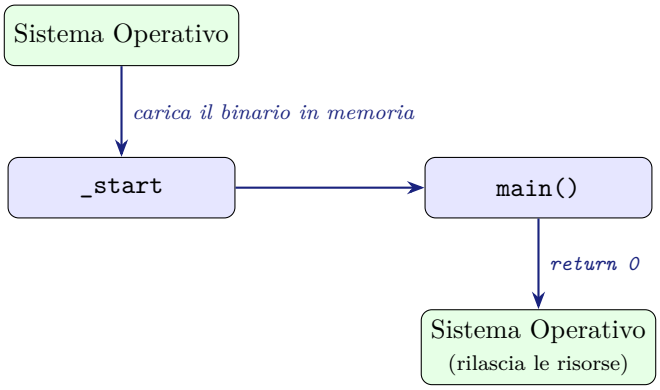
5.3. `printf("Hello, World!\n")`

- `printf` → funzione per stampare su standard output (da `stdio.h`)
- `"Hello, World!\n"` → la stringa da stampare
- `\n` → **escape sequence** per il carattere di fine riga
- Le istruzioni in C terminano sempre con `;`

5.4. `return 0`

- Ritorna il controllo alla funzione chiamante: in questo caso, il **sistema operativo**
- `0` = nessun errore
- Qualsiasi valore $\neq 0$ viene interpretato come errore

5.4.1. Flusso di esecuzione



6. Come si compila

Da riga di comando:

```
gcc -Wall -g -o helloworld helloworld.c
```

Listing 2: Compilazione con GCC

Flag	Significato
`-Wall`	Mostra **tutti** i warning
`-g`	Mantiene i **simboli di debug** (necessari per GDB)
`-o helloworld`	Imposta il nome del file di output
`helloworld.c`	Il file sorgente

Per eseguire il programma compilato:

```
./helloworld
```

Nota

Usare la **WSL** (Windows Subsystem for Linux) dentro VSCode per compilare. In Linux l'estensione `.c` è una convenzione, non un requisito tecnico.

7. Makefile

Un **Makefile** permette di automatizzare la compilazione senza riscrivere ogni volta il comando `gcc`. Diventa indispensabile quando il progetto ha **molti file**.

7.1. Struttura base

```
helloworld: helloworld.c
    gcc -Wall -g -o helloworld helloworld.c
```

Listing 3: Makefile minimale

Per compilare basta:

```
make
```

7.2. Anatomia di una regola

```
target: dipendenze
    comando
```

Elemento	Cosa è
`helloworld`	Il **target**: l'obiettivo della compilazione
`helloworld.c`	Le **dipendenze**: i file da cui dipende il target
`gcc ...`	Il **comando** (deve essere indentato con un TAB, non spazi!)

Attenzione

Se il target è più recente delle dipendenze, `make` non ricompila niente. Ricompila solo se rileva che le dipendenze sono cambiate (confronta le date di modifica). Per forzare una ricompilazione: `touch -m helloworld.c`.

7.3. Makefile con variabili

```
CC=gcc
CFLAGS=-Wall -g

helloworld: helloworld.c
    $(CC) $(CFLAGS) -o $@ $^

clean:
    rm -f helloworld
```

Listing 4: Makefile con variabili e meta-caratteri

Variabile / Simbolo	Significato
`CC`	Il comando del compilatore (default: `cc`)
`CFLAGS`	Le flag di compilazione
`$@`	Meta-carattere: sostituito con il **target**
`$^`	Meta-carattere: sostituito con tutte le **dipendenze**

Nota

Grazie ai meta-caratteri, se si vuole cambiare il nome del sorgente basta modificare solo la riga delle dipendenze — il comando si aggiorna automaticamente.

7.4. Il target `clean`

```
clean:
    rm -f helloworld
```

`make clean` cancella i file compilati. Utile per pulire il progetto prima di una ricompilazione da zero.

7.5. Makefile vs sistemi di build grafici

I sistemi di build con interfaccia grafica dipendono molto dalla configurazione locale. Il Makefile invece è **portabile**: funziona su qualsiasi sistema Unix-like senza configurazioni aggiuntive.