Lezione 01: Modularità

Compilazione

Un file sorgente .cpp (se corretto) può venir compilato in un eseguibile con:

```
g++ -o eseguibile sorgente.cpp
```

In questo caso il compilatore utilizzato è g++ da linea di comando, il comando `-o` ci permette di rinominare l'eseguibile prodotto in uscita. Il `.cpp` è il nostro file sorgente.

Linking

Posso spezzare il codice in diversi file sorgente: alcuni non conterranno alcun main.
Ogni file sorgente viene compilato separatamente in un file oggetto
`-o`

```
g++ -c sorgente.cpp
```

`-c` non produce l'eseguibile.

`-o` fileOggetto.o/eseguibile lo mettiamo solo se vogliamo dare all'uscita un nome diverso dal default.

`.cpp` è il file contenente il codice c++(non è detto contenga un main).

Viene poi fatto il linking dei file oggetto:

```
g++ -o eseguibile file1.o file2.o file3.0
```

Se si chiama il comando g++ direttamente sul file sorgente, vengono eseguiti i due passi precedenti in sequenza.

Script Bash

Quando i programmi crescono sia in dimensioni che in complessità è difficile ricordare le procedure sopra citate per ogni file. Una soluzione potrebbe essere uno script di shell. Bash è un esempio di script shell abbastanza semplice che ci permette di documentare i passi necessari ed eseguirli nella sequenza corretta, ad esempio:

```
#\!/bin/bash
g++ -O3 -o eseguibile file1.cpp file2.cpp
```

Modularità

Il concetto di **modularità** è fondamentale in programmazione perché consente di suddividere un progetto in parti più gestibili, riutilizzabili e comprensibili.

Anche se potrebbe sembrare comodo scrivere tutto in un unico file, nei progetti di una certa complessità questo approccio presenta diversi svantaggi:

- Ogni piccola modifica richiede la ricompilazione dell'intero file.
- Il codice diventa difficile da leggere e mantenere.
- È impossibile suddividere il lavoro in team in modo efficace.

Per questi motivi, è buona pratica **organizzare il codice in più file**, raggruppando insieme le parti strettamente correlate e separando quelle che hanno funzionalità distinte. Questo migliora la chiarezza, la manutenibilità e la collaborazione nel lavoro di squadra.

Struttura logica e struttura fisica

La **struttura logica** di un programma rappresenta l'organizzazione dei suoi componenti in base alla loro correlazione funzionale, raggruppando insieme quelli che svolgono compiti simili o connessi.

La **struttura fisica**, invece, riguarda la suddivisione del programma in file distinti, ognuno dei quali contiene uno o più componenti logici.

Questa distinzione consente di ottenere un progetto ben organizzato, chiaro e facilmente manutenibile.

Header file

La direttiva #include file.hpp viene elaborata dal preprocessore e viene sostituita dal contenuto del file, essa può avere due diverse forme:

```
#include <iostream> // from standard include
// dir
#include "myheader.hpp" // from current
directory

// Mettiamo nell'headre solo le dichiarazioni, non le definizioni
```

Gestione e conflitti

Supponiamo di avere un file sorgente `test.cpp` contenente alcune funzioni che vogliamo utilizzare anche in un altro file sorgente, ad esempio `main.cpp`. Includere tutto il codice all'interno di un unico file sarebbe una scelta poco saggia, poiché potrebbe generare conflitti e rendere il progetto difficile da gestire e manutenere.

Per questo motivo, è buona pratica separare l'interfaccia dall'implementazione utilizzando un file header, ad esempio `test.hpp`. In questo file andremo a dichiarare soltanto le funzioni (le loro *interfacce*), mentre la loro implementazione rimarrà nel corrispondente file `test.cpp`.
In questo modo, qualsiasi altro file che necessita di utilizzare quelle funzioni può semplicemente includere
`test.hpp`.

Tuttavia, può capitare che `test.hpp` venga incluso più volte, direttamente o indirettamente, attraverso altri file inclusi nella gerarchia. Questo può causare conflitti durante la fase di preprocessing. Per evitarli, si utilizzano delle direttive di preprocessore, comunemente note come *include guard*. Un esempio tipico è il seguente:

```
#ifndef TEST_HPP
#define TEST_HPP

// Dichiarazioni delle funzioni

#endif
```

Queste macro fanno in modo che il contenuto dell'header venga incluso una sola volta, anche se il file viene referenziato più volte nel progetto. In alternativa, è anche possibile usare la direttiva `#pragma once`, supportata dalla maggior parte dei compilatori moderni.

Infine, il linker si occuperà di unire tutte le parti del codice in modo coerente, garantendo che le dichiarazioni e le definizioni siano correttamente collegate tra loro senza ridondanze o ambiguità.

Strategie di partizionamento del codice

Header singolo

Prevede un unico file `.hpp` contenente tutte le dichiarazioni, incluso da tutti i file `.cpp` del progetto. Le implementazioni risiedono in file `.cpp` separati. È una soluzione semplice e funzionale per progetti piccoli o medi, ma può diventare rigida e poco scalabile.

Header multipli

Ogni file `.cpp` ha un proprio header `.hpp` che ne dichiara le interfacce.
Ogni
`.cpp` include il proprio header e quelli delle dipendenze.
Questa struttura è più modulare e adatta a progetti grandi.
Per maggiore chiarezza, si può distinguere tra header pubblici (
`.hpp`) e interni (`_impl.hpp`).

In sintesi:

- **Header singolo**: più semplice, meno scalabile.

- **Header multipli**: più organizzato e flessibile, ma più complesso da gestire in progetti molto grandi.

Il comando make

`make` è uno strumento utile per gestire e mantenere progetti composti da più file sorgente.
La sua funzione principale è
verificare automaticamente quali file devono essere ricompilati dopo una modifica, evitando di ricompilare tutto il progetto.

Per funzionare, `make` utilizza un file chiamato `makefile`, che descrive:

- **Le dipendenze** tra i file
- **Le regole di compilazione**
- **Le macro** (per semplificare i comandi)
- **Regole implicite**, basate sulle estensioni dei file

Sebbene sia usato principalmente per compilare programmi, può essere impiegato anche per gestire altri compiti complessi che richiedono il rispetto di dipendenze tra file o operazioni.

```
# ------------------------------------------------
# Variabili configurabili
# ------------------------------------------------

# Compilatore da usare
CXX = g++

# Opzioni di compilazione (es: warning e standard C++)
CXXFLAGS = -Wall -Wextra -std=c++17

# Nome del file eseguibile finale
TARGET = programma

# Lista dei file oggetto (.o)
```

```
OBJS = main.o math.o utils.o

# ------------------------------------------------
# Regola di default (quella eseguita con solo "make")
# ------------------------------------------------

all: $(TARGET)

# ------------------------------------------------
# Regola per creare l'eseguibile finale
# $@ = nome del target (programma)
# $^ = lista delle dipendenze (tutti gli .o)
# ------------------------------------------------

$(TARGET): $(OBJS)
    $(CXX) $(CXXFLAGS) -o $@ $^

# ------------------------------------------------
# Regole per compilare i singoli file sorgente
# $< = prima dipendenza (es: math.cpp)
# ------------------------------------------------

main.o: main.cpp math.hpp utils.hpp
    $(CXX) $(CXXFLAGS) -c $<

math.o: math.cpp math.hpp
    $(CXX) $(CXXFLAGS) -c $<

utils.o: utils.cpp utils.hpp
    $(CXX) $(CXXFLAGS) -c $<

# ------------------------------------------------
# Comando per pulire la directory (make clean)
# ------------------------------------------------

clean:
```

```
    rm -f *.o $(TARGET)

# ------------------------------------------------
# Comando per forzare la ricompilazione completa
# ------------------------------------------------

rebuild: clean all

# ------------------------------------------------
# Phony targets: non sono veri file, evitano conflitti
# ------------------------------------------------

.PHONY: all clean rebuild
```

Il comando `make` gestisce la compilazione confrontando **la data di modifica** del file obiettivo con quella dei file da cui dipende.

- Se **una qualsiasi dipendenza è più recente** dell'obiettivo, allora l'obiettivo viene ricreato.
- Questo processo è **ricorsivo**: ricompilare un file può comportare la ricompilazione di tutti gli obiettivi che da esso dipendono, direttamente o indirettamente.
- Di default, `make` costruisce **il primo obiettivo** definito nel `makefile`.
- L'ordine degli altri obiettivi non è importante: `make` segue automaticamente le **dipendenze**, ricostruendo ciò che serve nell'ordine corretto.