Modularità

Anna Corazza

aa 2023/24

Compilazione

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

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

g++ Compilatore GNU `https://gcc.gnu.org/`: ne esistono anche altri

-o L'opzione `-o` ci permette di rinominare l'eseguibile prodotto in uscita

.cpp Il file contenente il codice C++

```
g++ -O3 -o eseguibile sorgente.ccp
```

- -O3 L'opzione `-ON` con N=0,1,2,3, ottimizza il codice (noi usiamo N=3)

Linking

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

```
g++ -c sorgente.ccp
```

-c Non produrre l'eseguibile

-o sorgente.o Lo mettiamo solo se vogliamo che l'uscita abbia un nome diverso da sorgente.o

.cpp Il file contenente il codice C++ (ma potrebbe non contenere nessun `main`)

  - viene poi fatto il linking dei file oggetto

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

  - se si chiama il comando g++ direttamente sui file sorgente, vengono eseguiti i due passi precedenti in sequenza
  - Esempio in linking

Script `bash`

Bourn Again SHell

- Quando i programmi crescono in dimensioni e complessità, diventa complicato ricordarsi e ripetere la procedura di compilazione e linking delle singole parti.
- Una soluzione (non l'unica) è di usare uno script di shell: `bash` è una shell di uso relativamente facile che ci permette di:
  1. Documentare i passi necessari.
  2. Eseguirli nella sequenza corretta.
- Script `build.sh`:

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

Compilazione e linking

- Domanda fondamentale: **come divido il codice** in file diversi?
- Dal punto di vista del funzionamento, potrei mettere tutto in un unico file:
  - ricompilare tutto per ogni piccola modifica
  - impossibile da leggere e capire: la struttura in file è importantissima per la comprensibilità del codice
  - impossibile spartire il lavoro tra componenti di una squadra
- Oltre al codice sviluppato, ci sono anche eventuali **librerie** che devono concorrere a formare un unico eseguibile.
- **Conclusione**: la divisione in file deve seguire una struttura logica per cui parti di codice
  - molto correlate: nello stesso file
  - poco correlate: files diversi

Modularità

Cosa vogliamo ottenere

- **Struttura logica** di un programma: i componenti sono organizzati in modo da avere insieme quelli che sono in relazione tra loro.
- **Struttura fisica** di un programma: il programma è diviso in uncerto numero di file, e ciascuno di essi contiene uno o più componenti logici.
- Vogliamo trovare una struttura fisica che rappresenti i componenti logici in modo:

  consistente
  comprensibile
  flessibile
- Tuttavia le due possono non coincidere (es: funzioni dello stesso namespace possono stare in file diversi)
- **Importante** separare le **interfacce** (es: dichiarazione di una funzione) dall'**implementazione** (es: la sua implementazione).

Unità di compilazione: file

Conseguenze

- Ogni file viene compilato separatamente.
- Ogni file deve essere ricompilato dopo ogni modifica.
- Se il file è esageratamente grande, rischio di doverlo ricompilare molte volte.
- Dividere i file in unità più piccole aiuta a contenere i tempi di compilazione.
- Inoltre, una buona organizzazione in file:
  - enfatizza la struttura logica del programma
  - aiuta l'umano a capire il programma
  - aiuta il compilatore a imporre la struttura logica

Header file

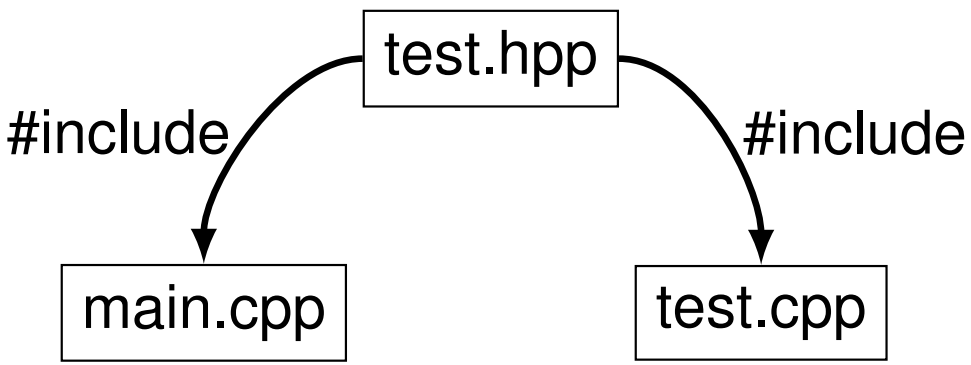
- La direttiva `#include file.hpp` viene elaborata dal preprocessore
- viene sostituita da contenuto del file
- Due diverse forme:

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

- Mettiamo nell'header solo le dichiarazioni, non le definizioni (un po' più complicato di così, ma lo vedremo in seguito)
- Semplice esempio SplitInFiles.

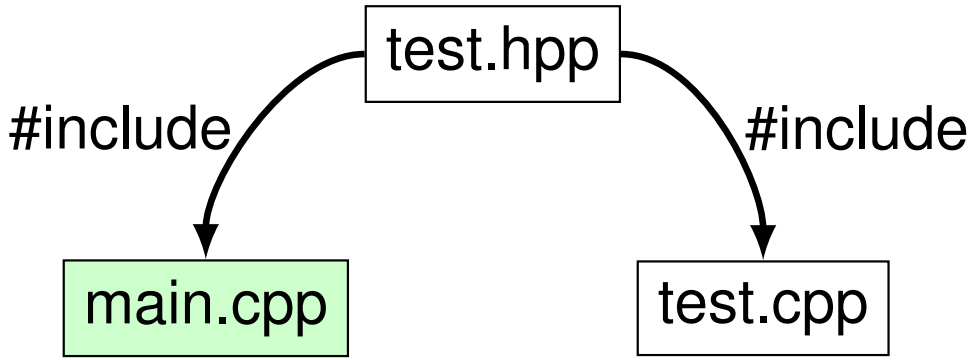
Esempio

- Alcune funzionalità sono state implementate in un file sorgente `test.cpp`.
- Devono venire usate in un file sorgente `main.cpp`: cosa possiamo fare?
  - includere tutto il codide nel file che lo usa non è una buona idea, perché con inclusioni multiple è facile creare conflitti
  - Useremo invece degli header file per importare nel codice che ne ha bisogno **esclusivamente** le interfacce, lasciando l'implementazione in un apposito file `.cpp`



Esempio

main.cpp

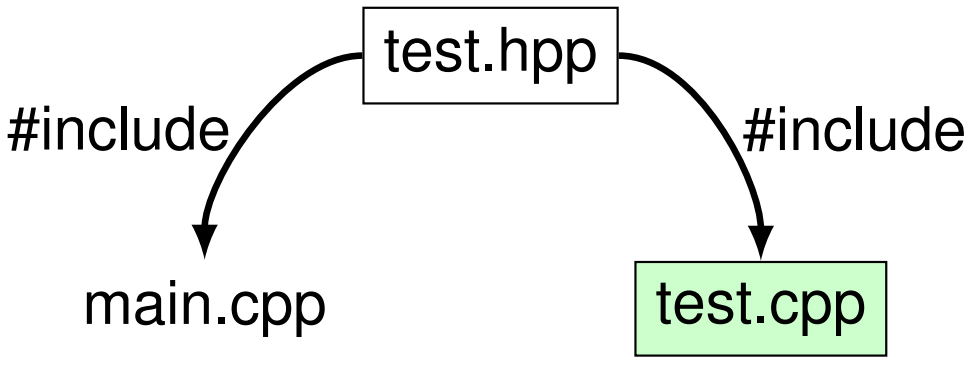


- main.cpp

```
1       #include <iostream>
2       #include "test.hpp"
3       using namespace std;
4       int main(){
5         cout << "Chiamata a funzione";
6         test();
7       }
```

Esempio

test.cpp

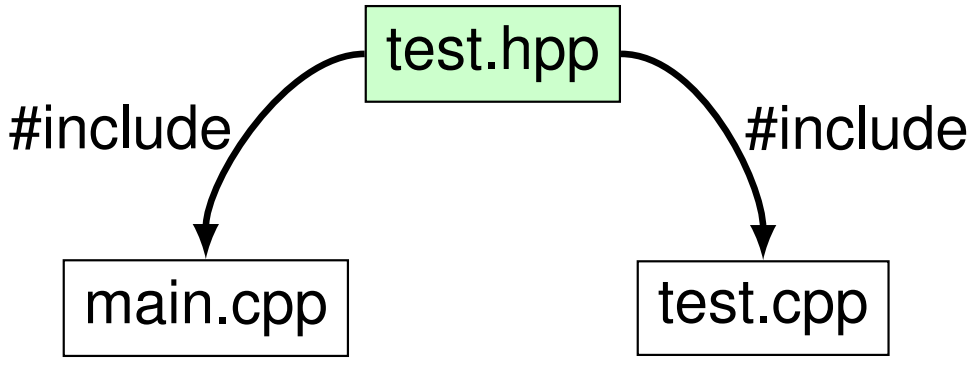


- test.cpp

```
1       #include <iostream>
2       #include "test.hpp"
3       void test(){
4         std::cout << "OK!" << endl;
5       }
```

Esempio: gestione dei conflitti

test.hpp



- `main.cpp`

```
1       #ifndef TEST_HPP
2       #define TEST_HPP
3       void test();
4       #endif
```

- Se `test.hpp` viene incluso in un file che a sua volta viene incluso in uno che già include `test.hpp` ...

Primo Passo: preprocessing

- Input: file sorgente
- Output: unità di traduzione (ma la distinzione viene fatta solo quando necessario)
- Operazioni: elaborazione macro, inclusione dei `#include`
- È l'output del preprocessing che viene passato al compilatore.
- L'organizzazione in file deve permettere al compilatore di elaborare ogni file separatamente dagli altri.
- In altre parole, in ogni file devono essere presenti o incluse tutte le dichiarazioni che servono alla compilazione.
- Tuttavia, le diverse dichiarazioni devono essere consistenti in tutto il programma: il linker si occupa di controllarlo.
- Il linking può essere completato prima o durante l'esecuzione (linking statico versus linking dinamico).

Strategie per partizionare il codice

Header singolo

- Mettere le definizioni in un numero opportuno di file `.ccp`.
- Mettere tutte le dichiarazioni necessarie in un unico file `.hpp` da includere (`#include`) in tutti i `.ccp`
- Funziona quasi sempre, quindi si parte da questa e la si modifica solo in caso di problemi.
- Passare a header multipli diventa inevitabile se il programma è grande.

Strategie per partizionare il codice

Header multipli

- Preparare un file header `.hpp` per ogni file sorgente `.ccp`.
- Il file header contiene le interfacce di tutto il contenuto del corrispondente file sorgente.
- Ogni file sorgente contiene un `#include` per l'header corrispondente e poi eventuali altri header per tutto ciò che utilizza e la cui implementazione è data in altri file.
- Una buona idea è quella di distinguere nel nome l'header rivolto all'utente e quello rivolto all'implementatore (ad esempio, `.hpp` verso `_impl.hpp`)
- Se il numero degli header file cresce, può diventare difficile da gestire.

Dove studiare

Str'13 : (2.2, 2.4, 15)).

Str'13 : B. Stroustrup, The C++ Programming Language, 2013, 4°ed.