Programmazione C++

UniShare

Davide Cozzi
@dlcgold

Gabriele De Rosa
@derogab

Federica Di Lauro
@f_dila

Indice

Capitolo 1

Introduzione

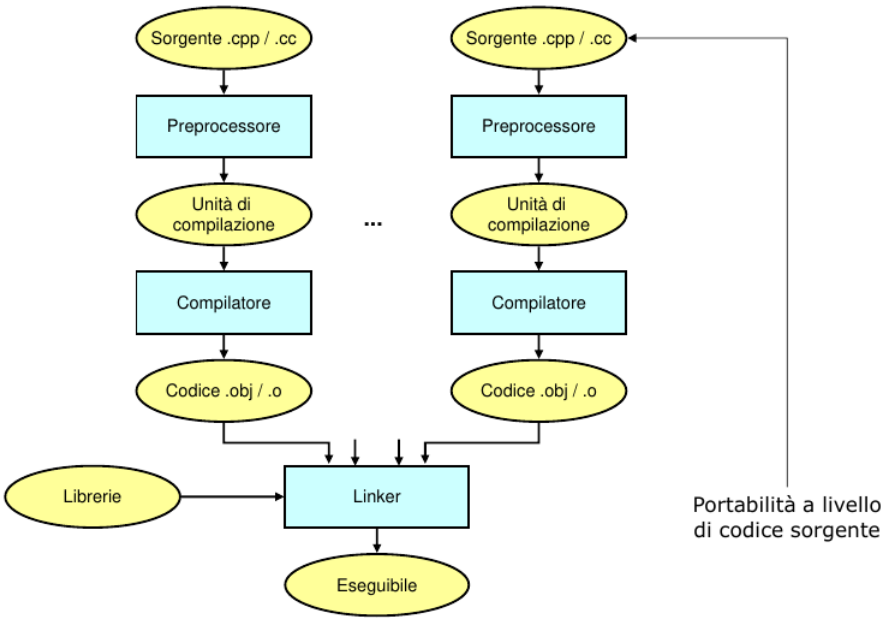
Questi appunti sono presi a lezione. Per quanto sia stata fatta una revisione è altamente probabile (praticamente certo) che possano contenere errori, sia di stampa che di vero e proprio contenuto. Per eventuali proposte di correzione effettuare una pull request. Link: `https://github.com/dlcgold/Appunti`.
Grazie mille e buono studio!

Capitolo 2

Introduzione a C++

Il **C++** nasce nel 1983 grazie a Stroustrup durante il suo dottorato che si occupava di studiare sistemi concorrenti. Conosceva il primo linguaggio OOP, chiamato **Simula**, che era comodo per i suoi scopi ma non riusciva a compilare il codice in maniera efficiente. Allora decise di passare a **BCPL** che però non è tipizzato e non è OOP. Decide quindi di inventarsi un nuovo linguaggio adeguato ai suoi scopi partendo dal **C** e unendo i concetti del **Simula**, ottenendo il **C++** nel 1998. Il C++ è superset del C e quindi è totalmente compatibile con esso. Il C++ è estremamente versatile, con paradigma procedurale, OOP (*class etc...*), modulare (*namespace etc...*) e generica (*template etc...*). In C++ si ha un ridotto overhead con una libreria standard semplice e minima, cosa che però costringe a cercare librerie esterne, spesso non completamente cross-platform. Nelle ultime versioni del C++ si ha anche l'introduzione di alcuni costrutti di programmazione funzionale (*lambda etc...*). Tra le ultime novità si annotano: *multitasking utilities, parallel execution, atomic operations, nullptr constant, delegation, initializer_list, automatic type deduction, hash table, smart pointer, tuple, espressioni regolari, random number facilities, new mathematical functions, type deduction per le classi, easy nested namespace, inizializzazioni negli if e switch, concepts, spaceship operator <=> etc* .... Il codice è al più possibile portabile anche se ovviamente ci sono limitazioni. Nella scrittura di codice C++ si preferisce evitare codice ibrido col C anche se è permesso.
Il C++ è un linguaggio compilato e i compilatori generano specifici codici macchina a seconda delle architetture in uso, si ha quindi portabilità a livello di sorgente (e non a livello di bytecode come per java, ovvero una volta "compilato" in bytecode si può eseguire l'eseguibile ovunque ci sia una JVM). Essendo portabile a livello di sorgente (con estensione ".cpp" o ".cc") ogni volta che si usa una nuova macchina bisogna ricompilare. In C++ ogni sorgente viene compilato a parte, producendo un file oggetto (con estensione ".o") e

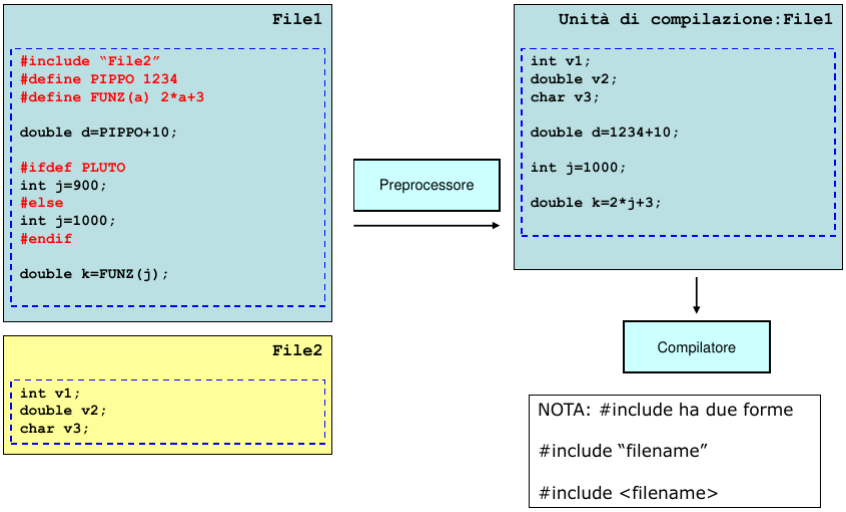
infine, mediante il *linker*, si ha la produzione dell'eseguibile includendo anche i sorgenti delle librerie esterne indicate. Prima di produrre il file oggetto si passa per il *preprocessore* che secondo alcune direttive produce delle *unità di elaborazione* che vengono poi compilate per produrre i file oggetto. Si ha quindi il seguente schema:



Il preprocessore trasforma un sorgente in un altro interperando delle direttive nel sorgente indicate col carattere #. Si hanno 3 gruppi:

- le *#define* che definiscono macro, ovvero sostituzioni di stringhe e il compilatore procede con una sorta di find-replace ma senza *type-checking* e quindi sarebbero da evitare. Si hanno anche le più gravi macro a funzioni. Noi useremo le macro per definire TAG. L'opposto di *#define* è *#undef*
- le direttive *if*, come *#if #ifdef/#ifndef #else/#elif #endif* per fare qualcosa in determinate situazioni. Comodo nel crossplatforming 8ho linux faccio qualcosa, ho windows faccio altro)
- le *#include* per includere altri sorgenti nel sorgente in suo (comprese le parti della libreria standard). Si ha sia la forma con i doppi apici ("") che con le parentesi angolari (<>) il cui comportamento dipende dal compilatore. Normalmente coi doppi apici si cerca prima nella cartella di lavoro che nel sistema e viceversa con le parentesi angolari

Nella pratica si ha quanto spiegato nella seguente immagine:



Il compilatore controlla la sintassi (analisi sintattica), il type checking, le chiamate a funzione (per passarle al linker) e poi genera il codice il codice di debug. Infine ottimizza il codice e produce il codice oggetto. Si ha poi il linker che unisce i file oggetto e elimina codice duplicato o inutilizzato, linkando le varie chiamate a classi e funzioni. Il linker ha quindi un solo errore: "*Unresolved External Symbol*". **La compilazione separata permette di non dover ricompilare ogni volta tutti i sorgenti**. Infine il linker aggiunge un codice di startup per l'avvio dell'eseguibile.
In C++ si hanno due tipi di sorgenti:

1. i file ".cpp" che contengono le definizioni di funzioni e classi, variabili. Sono compilati separatamente e possono essere visti come file di implementazione di una libreria di funzionalità

```
// Dichiarazione funzioni
// Se non fatto si ha errore
double f1(double v);
int f2(int v1, int v2);
void f3(int v);

// Funzioni Globali
double f1(double v)
{
  ...
}
int f2(int v1, int v2)
```

```
{
  ...
  // usa f1()
  ...
  // usa f3()
  ...
}
void f3(int v)
{
  ...
}
...
```

2. i file ".h" *(header file)* che contengono le dichiarazioni di funzioni, tipi di dati (classi e simili) e variabili. Possono essere visti come file di interfaccia di una libreria e sono inclusi dai file ".cpp" (e altri file ".h") per conoscere l'interfaccia di ciò che viene usato. Si permette quindi di evitare di ripetere ogni volta le dichiarazioni di qualcosa usato in più sorgenti. Si ha quindi il file ".cpp" con:

```
#include "file.h"
// Funzioni Globale
double f1(double v)
{
  ...
}
int f2(int v1, int v2)
{
  ...
  // usa f1()
  ...
  // usa f3()
  ...
}
void f3(int v)
{
  ...
}
...
```

e il file ".h" con:

```
// signature (dichiarazioni)
// di funzioni che voglio far
// conoscere ad un file .cpp
double f1(double v);
int f2(int v1, int v2);
void f3(int v);
...
```

Con la programmazione generica questa divisione viene persa con i file ".hpp".
Per ogni sorgente, escluso il main, bisogna avere un file header associato.
Si possono avere inclusioni a vicenda nei vari file header ma questo può generare problemi avendo più volte l'inclusione di un certo header. Per ovviare a questo si usano le *guardie* (, *#define, #endif*) che sono direttive del preprocessore che definiscono un TAG solo se non già definito. Queste sono così usate:

```
#ifndef ops_H
#define ops_H
#include "common.h"
// Contenuto di ops.h
#endif
```

Si devono usare tag univoci e usare il nome del file con l'underscore è comodo (magari in maiuscolo).
Vediamo ora la differenza tra definizone e dichiarazione. Definire implica scrivere il codice:

```
// definizione funzione
tipo_ritorno nome(tipo1 var1, tipo2 var2)
{
  // codice
}

// definizione variabile
tipo1 nome1;
tipo2 nome2=valore; // inizializzazione
```

Per dichiarare invece si ha:

```
// definizione funzione
tipo_ritorno nome(tipo1 var1, tipo2 var2)
{
  // codice
}

// definizione variabile
tipo1 nome1;
tipo2 nome2=valore; // inizializzazione
```

Una dichiarazione è anche una variabile. Dichiarare più volte la stessa cosa è rindondante mentre definire più volte è errore.
Mai includere un ".cpp" in un altro.