

Indice

| | |
|-------------------------------------|---|
| 1. Introduzione | 2 |
| 1.1. Processo di compilazione | 2 |
| 1.2. Tipi di dato | 6 |

1. Introduzione

1.1. Processo di compilazione

Un programma scritto nel linguaggio C++ é in genere costituito da uno o piú **file sorgente**, dei file di testo ciascuno contiene una parte del codice. Quando la compilazione viene invocata, prima che avvenga la compilazione vera e propria ciascun file viene modificato da una componente specifica del compilatore chiamata **preprocessore**. Questo converte il file di testo originale in un altro file di testo, nel quale sono state però fatte delle specifiche sostituzioni sulla base di **direttive**, contenute nel file stesso. Il risultato dell'operato del preprocessore é un file testuale di codice "puro", dove le direttive sono sostituite dalle rispettive valutazioni. Ciascuno di questi file viene detto **unità di compilazione**. Tale file esiste solo in memoria e viene passato al compilatore. Per ciascuno di questi il compilatore lo converte in un **file oggetto**, file che contiene la rappresentazione in codice binario dell'unità di compilazione in input. Tali file non sono portabili, perché il loro contenuto dipende sia dall'architettura su cui il compilatore é stato eseguito, sia dal sistema operativo su cui il compilatore é stato eseguito sia dal compilatore stesso. Tali file di per loro non sono eseguibili; un'ultima componente del compilatore é il linker, che unisce tutti i file oggetto in un solo eseguibile.

Il preprocessore interpreta delle istruzioni speciali (direttive), riconoscibili perché vi viene anteposto il simbolo **#**. Le direttive piú importanti e piú utilizzate sono:

- **#define** e **#undef**. Permettono la definizione di **macro** o di **tag**. Una macro é una stringa che, in ogni posizione del codice in cui viene individuata, deve venire sostituita una seconda stringa. Tale sostituzione non viene interpretata semanticamente dal compilatore, pertanto può essere sia una sostituzione tra due stringhe vere e proprie oppure la sostituzione di una stringa con una espressione. Una tag é una etichetta che viene registrata nella memoria del compilatore, da usarsi nelle direttive condizionali di seguito riportate. Macro e tag sono spesso riportate in maiuscolo per distinguerle dalle variabili vere e proprie, ma non vi sono restrizioni vere e proprie (al di lá di quelle che già esistono) sul loro nome;
- **#if**, **#else**, **#elif** e **#endif**. Hanno la stessa funzionalità del costrutto **if-else**, ma operano rispetto al codice e non rispetto alla sua logica. Possono essere usate, in combinazione con i tag, per rendere parti di codice bypassate durante la compilazione. Una loro possibile utilità consiste nel rendere il codice cross-platform, istruendo il compilatore ad operare in modi diversi a seconda della piattaforma su cui il codice viene eseguito;
- **#ifdef** e **#ifndef**. Come i precedenti, ma anziché effettuare una valutazione di espressioni logiche valutano se una certa macro o tag é stata definita oppure no;
- **#include**. Permette di riportare il nome di un file sorgente esterno da includere nel file sorgente attuale, di modo da avere accesso alle variabili e ai metodi in questo definito. Esiste in due forme: **#include "filename"** e **#include <filename>**. Entrambe hanno la stessa funzionalità, l'unica differenza sta nella posizione del filesystem in cui tali file vengono cercati. La prima predilige la ricerca di file usando percorsi assoluti (partendo quindi dalla cartella in esame) mentre la seconda predilige la ricerca di file usando il percorso standard in cui i file delle librerie si trovano (questo dipende da sistema operativo a sistema operativo, su Linux **/usr/include**).

| | | |
|--|-------------------------|------------------------------------|
| <code>#include "File2"</code> | <code>int v1;</code> | <code>int v1;</code> |
| <code>#define PIPPO 1234</code> | <code>double v2;</code> | <code>double v2;</code> |
| <code>#define FUNZ(a) 2 * a + 3</code> | <code>char v3;</code> | <code>char v3;</code> |
| | | |
| <code>double d = PIPPO + 10</code> | | <code>double d = 1234 + 10;</code> |
| <code>#ifdef PLUTO</code> | | |
| <code>int j = 900;</code> | | <code>int j = 1000;</code> |
| <code>#else</code> | | |
| <code>int j = 1000;</code> | | <code>double k = 2 * j + 3</code> |
| <code>#endif</code> | | |
| | | |
| <code>double k = FUNZ(j);</code> | | |

Il compilatore analizza sintatticamente il codice sorgente per verificare che non siano presenti typo. Effettua inoltre una parziale analisi semantica, in particolare il **type checking** (ad esempio, valutare che un valore pas-

sato ad una funzione sia del tipo specificato nella firma della funzione, oppure che una variabile venga inizializzata con un dato coerente col suo tipo) e l'identificazione di variabili e funzioni esterne, che non possono essere incluse immediatamente ma che devono attendere la fase di linking, ed é quindi necessario riportare dei "placeholder". Il compilatore, oltre a convertire le istruzioni dal formato testuale a quello binario, aggiunge (se necessario) delle informazioni di debug aggiuntive utili per la fase di testing.

I riferimenti a componenti esterne al file oggetto vengono risolti dal linker, che cerca negli altri file oggetto le variabili e le funzioni che nel file in esame hanno un nome ma non una implementazione, sostituendo il "placeholder" con l'indirizzo di memoria della variabile/funzione presa dal file oggetto dove é contenuta. Un errore tipico che il linker può emettere é `Unresolved External Symbol`, che avviene quando in un file oggetto é riportato il nome di una funzione/variabile che non é presente in nessun file oggetto che il linker ha esaminato. Il linker unifica i vari file oggetto in un solo eseguibile, ed introduce del codice di **startup** per renderlo riconoscibile dal sistema operativo come tale. Oltre ai file oggetto del codice in esame, il linker si occupa anche di aggiungere (se necessario) le librerie esterne. Queste, tranne la libreria standard (che viene inclusa sempre in automatica) devono essere specificate manualmente.

Sebbene, come già detto, i file sorgente possono avere estensioni a piacere (fintanto che sono file di testo), per convenzione i file sorgente si dividono in due categorie:

- I file con estensione `.cxx`: contengono la definizione vera e propria di funzioni (il loro corpo) e variabili (il loro effettivo valore). Possono essere visti come l'implementazione di una libreria della quale é nota l'interfaccia. Tali file sono quelli che vengono effettivamente compilati.
- I file con estensione `.h`, anche chiamati **file header**: contengono la dichiarazione di funzioni (la loro firma), variabili (il loro tipo) e tipi di dato definiti dall'utente (classi e simili). Possono essere visti come l'interfaccia di una libreria, ovvero riportano solamente *cosa é necessario implementare* ma non l'implementazione in sé e per sé. Tali file non vengono in genere compilati, ma vengono inclusi nei file `.cxx` per rendere loro disponibili le interfacce da implementare.

Per tale motivo, i file sorgente dei codici C++ sono in genere a coppie: un file `.h` che contiene l'interfaccia ed il corrispettivo file `.cpp` che ne implementa le funzionalità. C++ non supporta le **forward declarations**: se nel codice é presente una funzione che non ha una firma (anche se non é nota l'implementazione), viene restituito un errore. Riportare le firme delle funzioni in file header permette di rendere nota al compilatore la firma di una funzione prima che questa venga implementata, di modo che non sia necessario rivedere l'ordine della dichiarazione delle funzioni ad ogni cambiamento.

Sebbene non sia impedito l'usare `#include` per includere un file `.cpp` in un file `.cpp`, questo comportamento viene in genere scoraggiato perché rende i due file non più indipendenti. Se si vuole avere del codice condiviso fra più file, é preferibile che si trovi in un header file.

Puó capitare che un header file venga incluso più volte nello stesso file `.cpp`, specialmente quando il progetto é molto grande. Di base questo non é un problema, dato che ciò che accade é che il preprocessore deve eseguire più volte "a vuoto" una stessa sostituzione della direttiva `#include`; sebbene non sia un comportamento problematico, potrebbe comunque far sprecare tempo al preprocessore e rallentare il processo di compilazione. Per prevenirlo é possibile introdurre la cosiddetta **guardia**, che non é altro che una struttura del tipo:

```
#ifndef something_H
#define something_H
...
#endif
```

In questo modo, il preprocessore include `something.h` solamente se non é mai stato finora incluso.

La suddivisione del codice in più file oggetto permette la **compilazione separata**: un file sorgente deve venire ricompilato solamente se viene modificato direttamente.

Nel C++ si distingue tra **dichiarazione** e **definizione** di una funzione o di una variabile. Dichiarare una funzione significa riportare il tipo del valore di ritorno di tale funzione, il suo nome ed il numero e tipo dei suoi argomenti. Definire una funzione significa, oltre a dichiararla, anche riportarne il corpo.

```
// definition
return_value_type function_name(type_arg1 name_arg1, ..., type_argN name_argN)

// declaration
return_value_type function_name(type_arg1 name_arg1, ..., type_argN name_argN)
{
    // body goes here...
}
```

Definire una variabile significa notificare al compilatore che tale variabile esiste ed ha un certo nome, ma quale sia il suo valore non è da cercarsi nel file attuale (in genere questo viene fatto per la definizione di costanti globali in sostituzione a `#define`). Dichiarare una variabile significa sia esplicitarne il suo tipo, sia **inizializzarla**, ovvero assegnarle un valore iniziale; le due operazioni possono essere compiute separatamente o contemporaneamente. Una variabile non inizializzata assume in genere un valore casuale, che dipende dal contenuto della memoria che prima occupava tale variabile.

```
extern variable_type variable_name           // define a variable, without declaring

variable_type variable_name = initial_value // declare a variable and initialise

variable_type variable_name                // first declare a variable...
variable_name = initial_value              // then initialise it
```

Come già detto, il compilatore non dà errori fintanto che esiste una firma di una funzione o il nome e tipo di una variabile. In altre parole, non dà errori fintanto che una variabile/funzione è *dichiarata*. È il linker a dare un errore nel caso in cui una variabile/funzione non è stata *definita*.

Si noti come una dichiarazione implichi anche una definizione, mentre non è necessariamente vero il contrario. Inoltre, dichiarare più volte una stessa variabile/funzione non dà errore, perché si sta semplicemente ripetendo più volte la stessa operazione, mentre definire più volte una stessa variabile/funzione dà spesso errore perché il linker non è in grado di distinguere quale “versione” della variabile/funzione debba venire utilizzata.

```
float euclidean_distance(int x1, int y1, int x2, int y2)

double temperature;
temperature = 4.2;

extern float gamma_constant;
```

L'entry point di un programma C++ è una funzione avente nome `main`. Tale funzione deve essere globale e ne deve esistere una ed una sola copia. Il suo tipo di ritorno deve essere `int`, perché ciò che viene restituito è il valore di successo o di errore dell'esecuzione del programma. Il suo numero di argomenti è variabile (anche zero), e tali argomenti vengono forniti al programma direttamente dall'utente quando il programma viene avviato.

```
int main()
{
    ...
    return 0;
}
```

Il programma `Hello, World!` per il linguaggio C++, che stampa sullo standard output tale stringa, è il seguente:

```
#include <iostream>

int main()
{
    std::cout << "Hello, World!" << std::endl;

    return 0;
}
```

`std::cout` è un oggetto definito nella libreria standard del C++ (nello specifico, definito nell'header `iostream`, che viene importato) preposto alla stampa sullo standard output. A prescindere di quale sia il tipo di dato che `std::cout` debba stampare, questo lo restituisce come carattere.

La dicitura `std` rappresenta il **namespace**, ovvero uno spazio logico dentro al quale sono definite delle funzionalità. I namespace vengono in genere utilizzati per suddividere le entità di una libreria da tutte le altre, e specificare che tali entità appartengono allo stesso gruppo. Nello specifico, `std` indica che l'entità di cui `std` è prefisso proviene dalla libreria standard. In questo modo, è anche possibile avere funzioni/variabili che hanno lo stesso nome ma che hanno un significato diverso a seconda del namespace a cui appartengono.

In maniera molto simile, per leggere input da tastiera è possibile sfruttare `std::cin`

```
#include <iostream>

int main()
{
    int something;
    std::cin >> something;
    std::cout << "I got " << something << std::endl;

    return 0;
}
```

Nel caso in cui si voglia mostrare un messaggio di errore, è possibile scrivere sullo standard error mediante `std::cerr`. Questo è in genere più rapido che scrivere sullo standard output perché lo standard error non fa buffering, e quindi l'overhead è minore. Aiuta inoltre a separare i messaggi di errore dal normale flusso di esecuzione del programma.

Gli operatori `<<` e `>>` sono operatori che rispettivamente inseriscono dati in uno stream ed estraggono dati da uno stream. C++ supporta la **ridefinizione** degli operatori, pertanto è possibile assegnare ad un operatore una funzione diversa a seconda del tipo di dato che si richiede che questo manipoli. In effetti, tali operatori sono essi stessi una ridefinizione, dato che il loro uso di "default" è lo shift logico (a sinistra e a destra rispettivamente). Sebbene i namespace abbiano il pregio di separare in maniera elegante diversi package, riportarne il nome ogni volta che viene riportato un suo oggetto o funzione può diventare tedioso. Per questo motivo è possibile includere l'istruzione `using namespace` per specificare al compilatore che, all'interno del file corrente, tutte le funzioni e gli oggetti potrebbero provenire da tale namespace. Dato che l'effetto di questa istruzione viene propagato, è preferibile evitare di riportarla nei file header.

Il C++ è retrocompatibile con C, pertanto è possibile importare normalmente librerie C; tali funzionalità non sono legate ad un namespace vero e proprio, ma si trovano nel namespace globale. Spesso le librerie pensate per il linguaggio C possono venire utilizzate nel C++ in maniera nativa incapsulando tali funzionalità in un namespace. La differenza fra le due, ovvero fra le librerie per C importate in C++ e librerie in C++ propriamente dette, sta nel nome dell'header importato: le seconde sono importate specificando il file per intero, estensione inclusa, mentre le seconde vengono importate troncando l'estensione. Nel caso specifico della libreria standard del C, molte delle funzionalità di tale libreria sono incapsulate dalla libreria standard del C++ in header che hanno il medesimo nome ed una 'c' come prefisso.

La libreria standard del C `math.h` contiene alcune funzioni matematiche più elaborate delle operazioni standard, come ad esempio il calcolo della radice quadrata (`sqrt`) o l'arrotondamento per eccesso o per difetto (`floor` e `ceil`). Per importarla in un codice C++ è sufficiente specificare la direttiva `#include <math.h>` e le funzioni da questa fornite sono disponibili senza dover specificare un namespace (non avendolo). In alternativa, la libreria standard del C++ incapsula `math.h` nel namespace `std` (senza modificarne le funzionalità) pertanto è anche possibile accedere alle funzioni di `math.h` mediante la direttiva `#include <cmath>`, e tali funzioni avranno `std` come namespace.

```
#include <math.h>
sqrt(16);
```

```
#include <cmath>
std::sqrt(16);
```

1.2. Tipi di dato

Nel C++ si distinguono i seguenti tipi di dato primitivi:

- **booleani**, `bool`;
- **caratteri**, `char`;
- **numeri interi**, `int`;
- **numeri decimali**, `float` (singola precisione) `double` (doppia precisione);
- **puntatori**;
- **reference**;

Possono essere poi aggiunti dei *modificatori* ai tipi primitivi:

- `unsigned`, che rimuove il segno dai tipi numerici;
- `long`, che raddoppia il numero di bit usato per rappresentare il valore di una variabile di tipo `int` o `double`. Scrivendo solamente `long` si sottintende `long int`;
- `short`, che dimezza il numero di bit usato per rappresentare il valore di una variabile di tipo `int`. Scrivendo solamente `short` si sottintende `short int`;

La quantità di bit effettivamente utilizzata per rappresentare una variabile non è fissata, ma dipende dalle caratteristiche dell'architettura su cui il codice viene compilato e dal sistema operativo. Lo standard fissa comunque dei vincoli di massima:

- `char` è la più piccola entità che può essere indirizzata, pertanto non può avere dimensione inferiore a 8 bit;
- `int` non può avere dimensione inferiore a 16 bit;
- `double` non può avere dimensione inferiore a 32 bit.

Le informazioni relative a quanti bit sono allocati per una variabile di ogni tipo si trova in un file della libreria standard chiamato `#limits.h`, che può eventualmente essere importato per ricavare tali informazioni. In alternativa, la funzione della libreria standard `sizeof()` restituisce la dimensione in multipli di `char` del tipo di dato passato come argomento.

```
std::cout << sizeof(char) << std::endl;    // prints, say, 1
std::cout << sizeof(int) << std::endl;     // prints, say, 4
std::cout << sizeof(double) << std::endl;  // prints, say, 8
```

Un puntatore è un tipo di dato che contiene un riferimento ad un indirizzo di memoria. È possibile utilizzarli per accedere ai dati indirettamente, non manipolando il valore della variabile in sé ma bensì accedendo alla memoria su cui tale dato si trova. Esistono tanti tipi di puntatori quanti sono i tipi primitivi; per quanto contengano un indirizzo di memoria (che è un numero), il loro tipo non è un intero, bensì è specificatamente di tipo “puntatore a...”. Puntatori a tipi diversi sono incompatibili. Il valore di default per un puntatore è `NULL` oppure (standard C++11) `nullptr`.

Un puntatore viene dichiarato come una normale variabile di tale tipo, ma postponendo `*` al tipo¹. Antepo-
nendo `&` al nome di una variabile se ne ottiene l'indirizzo di memoria. Per ricavare il valore della cella di memoria a
cui un puntatore é associato si antepone `*` al nome della variabile. L'atto di "risalire" al valore a cui un puntatore
é legato é una operazione che prende il nome di **dereferenziazione**.

```
variable_type* pointer_name           // declares a pointer
variable_type* pointer_name = &variable_to_point // declares and initialises a pointer
variable_type variable_name = *pointer_name // dereferences a pointer
```

```
int* p = nullptr;           // initialises a pointer p
int s = 10;                 // initialises an integer s
p = &s;                     // p points to the memory address of s
std::cout << *p << std::endl; // retrieves the value to which p is
                                // pointing, and prints it
(*p)++;                     // retrieves the value to which p is
                                // pointing, and increments it by one
```

Un puntatore, pur non venendo considerato un intero, può essere manipolato come tale. In particolare, é possi-
bile sommare un intero ad un puntatore, e l'operando `+` viene reinterpretato non come una somma nel senso
stretto del termine ma come lo spostamento di un offset di tante posizioni quante ne viene specificato. Inoltre,
cercando di stampare il valore di un puntatore mediante 'std::cout' si ottiene effettivamente l'indirizzo di me-
moria a cui il puntatore é associato (espressa in esadecimale)

Il fatto che sui puntatori sia possibile fare aritmetica può presentare un problema, perché significa che é tecnica-
mente possibile, dall'interno di un programma C++, raggiungere aree di memoria che non sono di competenza del
programma stesso, semplicemente incrementando o decrementando il valore di un puntatore. Fortunatamente
questo non può accadere, perché il sistema operativo lo previene emettendo un messaggio di errore e fermando
il programma prima che avvenga l'accesso.

```
int d = 1;
int* p = &d;

p++; // pointer's location is shifted by one.
      // Where is it now?
std::cout << p << std::endl // prints, say, 0xfffff7d7761c
```

¹Tecnicamente, la scrittura `type* v` é equivalente a `type * v` e a `type *v`. Questo perché il token `*` viene riconosciuto dal
parser singolarmente, quindi non c'é differenza nella sua posizione. Scegliere uno stile piuttosto che un'altro dipende da preferenze
personali.