Indice

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 peró 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 **unitá 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'unitá 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 puó 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 giá esistono) sul loro nome;
- `#if`, `#else`, `#elif` e `#endif`. Hanno la stessa funzionalitá 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 utilitá 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 funzionalitá, 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`).

```
#include "File2"
#define PIPPO 1234
#define FUNZ(a) 2 * a + 3

double d = PIPPO + 10
#ifdef PLUTO
int j = 900;
#else
int j = 1000;
#endif

double k = FUNZ(j);
```

```
int v1;
double v2;
char v3;
```

```
int v1;
double v2;
char v3;

double d = 1234 + 10;

int j = 1000;

double k = 2 * j + 3
```

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 passato 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 puó 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 giá 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 funzionalitá. 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 piú indipendenti. Se si vuole avere del codice condiviso fra piú file, é preferibile che si trovi in un header file.
Puó capitare che un header file venga incluso piú volte nello stesso file `.cpp`, specialmente quando il progetto é molto grande. Di base questo non é un problema, dato che ció che accade é che il preprocessore deve eseguire piú 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 piú 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 é 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 giá detto, il compilatore non dá errori fintanto che esiste una firma di una funzione o il nome e tipo di una variabile. In altre parole, non dá errori fintanto che una variabile/funzione é *dichiarata*. É il linker a dare un errore nel caso in cui una variabile/funzione non é stata *definita*.
Si noti come una dichiarazione implichi anche una definizione, mentre non é necessariamente vero il contrario. Inoltre, dichiarare piú volte una stessa variable/funzione non dá errore, perché si sta semplicemente ripetendo piú volte la stessa operazione, mentre definire piú volte una stessa variabile/funzione da spesso errore perché il linker non é 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++ é 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 ció che viene restituito é il valore di successo o di errore dell'esecuzione del programma. Il suo numero di argomenti é 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, é il seguente:

```cpp
#include <iostream>

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

    return 0;
}
```

`std::cout` é 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 funzionalitá. I namespace vengono in genere utilizzati per suddividere le entitá di una libreria da tutte le altre, e specificare che tali entitá appartengono allo stesso gruppo. Nello specifico, `std` indica che l'entitá di cui `std` é prefisso proviene dalla libreria standard. In questo modo, é 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 é possibile sfruttare `std::cin`

```cpp
#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, é possibile scrivere sullo standard error mediante `std::cerr`. Questo é in genere piú rapido che scrivere sullo standard output perché lo standard error non fa buffering, e quindi l'overhead é 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 é 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" é 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 puó diventare tedioso. Per questo motivo é 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, é preferibile evitare di riportarla nei file header.
Il C++ é retrocompatibile con C, pertanto é possibile importare normalmente librerie C; tali funzionalitá 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 funzionalitá 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 importante troncando l'estensione. Nel caso specifico della libreria standard del C, molte delle funzionalitá 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 piú elaborate delle operazioni standard, come ad esempio il calcolo della radice quadrata (`sqrt`) o l'arrotondamento per eccesso o per difetto (`floor` e `ceiling`). Per importarla in un codice C++ é 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 funzionalitá) pertanto é anche possibile accedere alle funzioni di `math.h` mediante la direttiva `#include <cmath>` , e tali funzioni avranno `std` come namespace.

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

```c
#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 quantitá di bit effettivamente utilizzata per rappresentare una variabile non é 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` é la piú piccola entitá che puó essere indirizzata, pertanto non puó avere dimensione inferiore a 8 bit;
- `int` non puó avere dimensione inferiore a 16 bit;
- `double` non puó 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 puó 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.

```c
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 é un tipo di dato che contiene un riferimento ad un indirizzo di memoria. Esistono tanti tipi di puntatori quanti sono i tipi primitivi; per quanto contengano un indirizzo di memoria (che é un numero), il loro tipo non é un intero, bensí é specificatamente di tipo "puntatore a...". Puntatori a tipi diversi sono incompatibili. Nonostante questo, la dimensione di un puntatore non dipende dal tipo di dato dato che gli indirizzi di memoria hanno tutti la stessa dimensione. Il valore di default per un puntatore é `NULL` oppure (standard C++11) `nullptr` .

Un puntatore viene dichiarato come una normale variabile di tale tipo, ma postponendo `*` al tipo[1]. Anteponendo `&` 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, puó essere manipolato come tale. In particolare, é possibile 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. Il numero di posizioni dipende dal tipo di puntatore: sommare N ad un puntatore equivale a spostare la cella di memoria a cui si riferisce di N volte la dimensione del tipo di dato a cui il puntatore si riferisce. La scrittura `p[n]` permette di risalire al valore che si trova `n` posizioni in avanti rispetto al puntatore `p` (é uno spostamento unito ad una dereferenziazione). La differenza fra due puntatori restituisce il numero di elementi che si trovano nell'intervallo fra le posizioni in memoria a cui i due si riferiscono. Il confronto (`=`) fra due puntatori viene fatto rispetto ai rispettivi valori, e non a ció a cui puntano.
Il fatto che sui puntatori sia possibile fare aritmetica puó presentare un problema, perché significa che é tecnicamente 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 puó accadere, perché il sistema operativo lo previene emettendo un messaggio di errore `Segmentation Fault` e fermando il programma prima che avvenga l'accesso. Per questo motivo non é consigliabile (a meno di casi eccezionali) inizializzare un puntatore fornendogli direttamente un indirizzo di memoria, perché questo comporta che si chieda al programma di accedere ad una area di memoria specifica senza poter sapere se il programma possa accedervi, dato che gli indirizzi in RAM vengono assegnati in maniera sostanzialmente arbitraria.
Cercando di stampare il valore di un puntatore mediante `std::cout` si ottiene effettivamente l'indirizzo di memoria a cui il puntatore é associato (espressa in esadecimale).

[1] Tecnicamente, 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.

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

int c = p[2];                   // A shortand for *(p + 2)

p = p + 3;                      // pointer's location is shifted by one.
                                // A shortand for p = p + 3 * sizeof(int)
std::cout << p << std::endl     // prints, say, 0xfffff7d7761c
```

Essendo un puntatore comunque una variabile, anch'esso si trova in una certa area di memoria, ed é pertanto possibile risalire all'area di memoria di un puntatore. Questo significa che é anche possibile avere dei puntatori a dei puntatori. Inoltre, nulla vieta di avere piú di un puntatore legato alla stessa area di memoria.

```
char s = 's';       // A char
char* ss = &s;      // A pointer to a char
char* sss = &ss;    // A pointer to a pointer to a char
char** f = &s;      // A pointer to a pointer to a char (in one go)
```

É possibile sfruttare dei puntatori di tipo `void` per aggirare le limitazioni imposte dal compilatore sui puntatori, in particolare i vincoli di tipo. Infatti, un puntatore di tipo `void` puó riferirsi a qualsiasi tipo di dato, ed é possibile riassegnare un puntatore di tipo `void` a dati diversi. Per operare la dereferenziazione é peró necessario compiere un casting esplicito al tipo di dato a cui il puntatore si riferisce in questo momento. Sebbene nel C vi fosse una certa utilitá nei puntatori `void`, nel C++ é da considerarsi una funzionalitá deprecata.

```
int i;
double d;

void* pi = &i;
void* pd = &d;
int* ppd = pd;              // NOT Allowed

int x = *((int*) (pi));     // Ok
int y = *((int*) (pd));     // Allowed, but VERY dangerous
```

Le utilitá dei puntatori sono riassunte di seguito:

- Permettono di riferirsi a piú dati dello stesso tipo;
- Permettono di condividere uno stesso dato in piú parti di codice senza doverlo ricopiare piú volte;
- Permettono di accedere ai dati indirettamente, non manipolando il valore della variabile in sé ma bensí accedendo alla memoria su cui tale dato si trova;
- Permettono il passaggio per parametri alle funzioni, non passando direttamente il valore ma il puntatore, risparmiando memoria;
- Permettono di costruire strutture dati dinamiche, come liste e alberi;

1.3. Array

Un **array** é una sequenza di dati dello stesso tipo, memorizzati in aree di memoria contigue chiamate **celle**, utile per memorizzare dati che fra loro hanno un qualche legame logico. Un array viene dichiarato come una normale variabile di un certo tipo, ma accodando `[]` al nome. La dimensione di un array é fissata, direttamente

nel codice riportandola fra le parentesi quadre oppure venendo "dedotta" dal compilatore in base a come l'array viene inizializzato.
Un array viene inizializzato riportando fra parentesi graffe i valori, separati da virgole, che verranno assegnati ordinatamente a ciascuna posizione dell'array. Se vi sono piú valori che posizioni nell'array, il compilatore restituisce un errore. Non é possibile cambiare i valori assegnati alle celle di un array usando la medesima sintassi dell'inizializzazione, ma occorre farlo cella per cella. Riportando il nome dell'array con N fra le parentesi graffe si ottiene il valore nella (N - 1)-esima cella dell'array; gli array partono da 0. Il compilatore non restituisce un messaggio di errore se si cerca di accedere ad una cella di memoria che supera le dimensioni dell'array.

```
array_type array_name[n]                    // Size set to n
array_type array_name[n] = {v1, ..., vm}    // m can't be greater than n
array_type array_name[]                     // Size to be determined
array_type array_name[] = {v1, ..., vn}     // Size is set to n by the compiler
```

```
char array1[3]                    // Not initialised
char array2[3] = {'a', 'b', 'c'}  // Fully initialised (sizes match)
char array3[3] = {'a'}           // Partially initialised (less elements than size)
int array4[] = {1, 2, 3, 4, 5}    // Size is set to 5 by the compiler

array2 = {'p', 'q', 'r'}          // NOT allowed
array3[0] = 'f';                  // Allowed
array3[1] = 'b';                  // Allowed
array4[10] = 10;                  // NOT allowed

char x = array3[0]                // Allowed
char x = array2[5]                // Allowed, but dangerous
```

Un array puó essere costituito a sua volta da array; si parla in questo caso di **array multidimensionale**. Un array multidimensionale viene dichiarato come un normale array ma riportando tante parentesi quadre quante sono le dimensioni; il valore fra le parentesi quadre indica la lunghezza degli array di tale dimensione. Solamente la prima dimensione di un array multidimensionale (quella piú a sinistra) puó venire lasciata alla deduzione del compilatore: le restanti devono per forza essere specificate. I "sotto-array" di un array multidimensionale sono comunque tutti dello stesso tipo.

```
// Multidimensional array having n dimensions. First dimensions is long
// a, second is long b, etcetera
array_type array_name[a][b]...[n]
```

```
int array0[5][3];
int array1[2][3] = {{1, 2, 3}, {4, 5, 6}};
int v = array1[0][2];                          // First array, third element

int array2[][3] = {{1, 2, 3}, {4, 5, 6}};      // Allowed
int array3[3][] = {{1, 2, 3}, {4, 5, 6}};      // NOT allowed
int array4[][] = {{1, 2, 3}, {4, 5, 6}};       // NOT allowed
```

La sintassi degli array é molto simile alla sintassi dei puntatori, perché i due sono intimamente legati. Infatti, é possibile inizializzare un puntatore con il nome di un array (fintanto che i tipi sono coerenti); sebbene questo non sia formalmente corretto, dato che un array *non* é un puntatore, questo abuso di notazione é accettato per motivi storici. Un puntatore che assume come valore un array diventa un puntatore al primo elemento di tale array. Una volta inizializzato un puntatore con un array, é possibile utilizzarlo per scorrere lungo l'array in maniera naturale. Un puntatore puó anche essere inizializzato con un elemento dell'array specifico, e diventa un puntatore a tale cella di memoria.

```
int array[3] = {-1, -2, -3};

int *p;
p = array;                          // Allowed, but should be p = &(array[0])

int *q = array++;                   // Allowed, but should be q = &(array[1])
array++;                            // NOT allowed
p++;                                // Allowed, points to array[1]

int w = p[0]                        // Equals *array[1]
int x = p[1]                        // Equals *array[2]
```

I puntatori possono essere associati anche ad array multidimensionali. Assegnare un puntatore ad un array multidimensionale corrisponde ad assegnarlo ad uno dei suoi sottoarray; scorrere con tale puntatore corrisponde a scorrere di sottoarray in sottoarray.

```
int array[2][3] = {{1, 2, 3}, {4, 5, 6}};

int* q = array;                              // NOT allowed
int (*p)[3] = array;                        // Pointer to first array of 3 integers

p++;                                         // Shifts to next 3-dimesional array
(*p)[1] = 10;                                // array[1][1] = 10
```

Una **reference** é un tipo di dato simile al puntatore. Una reference é di fatto un *alias* per un'altra variabile; ogni volta che viene fatta una manipolazione sulla reference, tale manipolazione viene propagata sulla variabile originale. Una reference deve necessariamente essere inizializzata quando viene dichiarata, pena messaggio di errore da parte del compilatore, perché una reference non inizializzata non ha alcun significato. L'inizializzazione deve essere rispetto ad una variabile, non rispetto ad un valore. Una volta dichiarato ed inizializzato, un reference non puó venire "sganciato" e riassegnato ad una variabile diversa, nemmeno se ha lo stesso tipo della precedente. Cosí come i puntatori, le reference sono di tipo "reference a...".

```
reference_type& reference_name = variable_to_be_referenced
```

```
int x = 10;
int& y = x;         // y references x
y++;                // de facto x++

int& z;             // NOT allowed
int& w = 10;        // NOT allowed
```

1.4. Struct, enum, typedef

Similmente agli array, che sono tipi primitivi, le **struct** sono considerate tipi compositi. Di fatto, una struct é un "raggruppamento" di dati anche di tipo diverso, chiamati **campi**. Sono di fatto una forma piú "rudimentale" del concetto di classe. Tutti i dati di una struct sono di default pubblici, quindi liberamente modificabili.
Una struct puó essere inizializzata allo stesso modo di come viene inizializzato un array, dove ogni elemento i-esimo all'interno delle parentesi graffe viene assegnato alla i-esima variabile contenuta nella `struct`. Una `struct` puó anche essere inizializzata parzialmente, ovvero assegnando un valore solamente ai primi n campi.

```
struct name_type {
  type_1 name_1;
  type_2 name_2;
  ...
  type_n name_n;
};
```

```
struct_type struct_name = {field_1, field_2, ..., field_n};
```

```
struct Point {
  int x;
  int y;
};

Point A = {5, 2};
```

L'operatore `.` permette di accedere ai dati di una `struct`, specificando il nome del campo a cui ci si riferisce. L'operatore `->` permette di accedere ad un campo di una struct quando ci riferisce ad essa tramite un puntatore e non direttamente (é una abbreviazione di una deferenziazione seguita da un accesso).

```
struct_name.field
```

```
pointer_to_a_struct->field
```

```
Point P = {5, 2};
Point* Q = &P;

P.x = 10;
Q->y = 8;       // same as (*Q).p = 8

std::cout << Q->x << " " << P.y << std::endl;    // prints 10 8
```

La memoria occupata da una `struct` dipende dalla politica di allocazione della memoria del compilatore. In genere, viene prediletta una allocazione di memoria che ottimizza l'accesso piuttosto che la dimensione. Per tale motivo, per ottenere la massima efficienza in termini di spazio occupato é preferibile disporre i dati all'interno in ordine decrescente di grandezza, di modo che piú dati possano venire "accorpati" in un'unica `word`.
Un `enum` é un tipo di dato che consente di associare in maniera automatica dei valori interi costanti a dei nomi di stringhe. Permette di usare delle stringhe come dei "segnaposto" per dei valori che dovrebbero essere legati da una qualche semantica.

```
enum name {name_1 = value_1, name_2 = value_2, ..., name_n = value_n};
```

Il valore a cui ciascun campo di un `enum` viene assegnato puó venire specificato oppure lasciato dedurre al compilatore. Nel secondo caso, a tutti i campi dell' `enum` che vengono dopo l'ultimo campo con un valore specificato viene assegnato il valore a quest'ultimo successivo. Se nessun valore viene specificato, ai campi di `enum` vengono ordinatamente assegnati i numeri $1, 2, 3, \ldots$,

```
enum day {Mon = 10, Tue = 20, Wed = 30, Thu = 40,
            Fri = 50, Sat = 60, Sun = 70};

day d;
d = Wed;          // Allowed
d = 10;           // NOT Allowed
int f = Fri;      // Allowed

enum days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};              // 1, 2, 3, 4, 5, 6, 7
enum days {Mon = 1, Tue, Wed = 5, Thu, Fri = 2, Sat, Sun};  // 1, 2, 5, 6, 2, 3, 4
```

`typedef` permette di associare un alias ad un tipo di dato giá esistente. É utile per riferirsi ad un tipo avente un nome molto lungo con un alias piú corto. Puó essere utile anche per "mascherare" valori veri con nomi di comodo, di modo che da fuori da una classe i dati appaiano con nomi piú semplici da comprendere.

```
typedef old_name new_name;
```

```
typedef unsigned long int uli;

uli x = 10;                       // Same as unsigned long int x = 10
```

`const` é un modificatore che, posto davanti alla definizione di una variabile, la rende immutabile, ovvero non é piú possibile modificarne il valore in un secondo momento. Permette di creare delle costanti, ovvero valori che devono imprescindibilmente assumere uno ed un solo valore[2]. Se si tenta di aggiungere `const` ad una variabile che non viene inizializzata quando viene dichiarata viene restituito un messaggio di errore.

```
const var_type var_name = value;
```

[2] `const` occupa lo stesso spazio che nel C aveva `#define`; infatti, sebbene sia possibile anche in C++ definire costanti in questo modo, é da considerarsi una worst practice, dato che il linguaggio offre uno strumento preposto.

Il valore di una reference a cui viene aggiunto il modificatore `const` puó cambiare se il valore originale viene cambiato, ma non puó comunque venire modificato direttamente.

```
const float pi;              // NOT Allowed
const float pi = 3.14;       // Allowed

pi = 3.1415;                 // NOT Allowed

g = 1;
const int gamma = g;         // Allowed

int f = 2;
const int& e = f;
f++;                         // Allowed, now e = 3 even if constant
e++;                         // NOT Allowed
```

Rispetto ai puntatori, l'uso del modificatore `const` puó portare a conseguenze impreviste. Possono presentarsi tre situazioni, in base a dove viene posta la keyword `const` nella dichiarazione del puntatore:

- Il modificatore `const` si trova prima del tipo di dato del puntatore. In questo senso, il puntatore "protegge" la variabile, impedendo che sia possibile modificarla se si passa dal puntatore. Sia il puntatore, sia l'oggetto in sé se vi si accede direttamente, sono liberamente modificabili. Infatti, la keyword `const` si riferisce comunque sempre e solo al puntatore, anche se il dato a cui si riferisce non é una costante;
- Il modificatore `const` si trova dopo il tipo di dato del puntatore. In questo senso, é il puntatore stesso ad essere una costante, e non é piú possibile modificarlo (scollegarlo e collegarlo ad altro, per esempio), ma é possibile modificare il valore dell'oggetto in sé se vi si accede tramite il puntatore;
- Il modificatore `const` si trova sia prima che dopo il tipo di dato del puntatore. Sia il puntatore, sia l'oggetto se vi si accede tramite il puntatore, non sono modificabili.

Assegnare ad un puntatore (non necessariamente con `const`) un tipo di dato che ha il modificatore `const` restituisce un errore in fase di compilazione, perché si sta di fatto negando il "senso" dell'aver dichiarato tale variabile come constante in principio.

```
int i = 200;

const int* p1 = &i;              // 1st type
*p1 = 100;                       // NOT allowed
p1 = nullptr;                    // Allowed

int* const p2 = &i;              // 2nd type
*p2 = 100;                       // Allowed
p2 = nullptr;                    // Not allowed

const int* const p3 = &i;        // 3rd type
*p3 = 100;                       // NOT allowed
p3 = nullptr;                    // NOT allowed
```

1.5. Stringhe

Una **stringa** é un tipo di dato che permette di memorizzare informazioni alfanumeriche. In C, le stringhe sono degli array di `char` il cui ultimo carattere é il carattere speciale `\0`. Quando a `cout` viene fornito un array di `char` con queste caratteristiche, vengono ordinatamente stampati tutti i caratteri dell'array ad eccezione di `\0`. Un puntatore a `char` viene interpretato come una stringa, pertanto non é possibile, a meno di usare una sintassi

particolarmente convoluta, riferirsi ad una stringa tramite un puntatore. Le stringhe del C hanno dei metodi che si trovano nell'header file `string.h` (o `cstring`).

```
char strc[10] = "Hello";
char strl[] = {'W', 'o', 'r', 'l', 'd', '!', '\0'};
char* strp = "Hello, World!";                    // should be const char* strp
```

Una sintassi del tipo `char* str = "..."` é ammessa perché é un residuo del modo in cui C gestisce le stringhe, ma non é tecnicamente corretta. Infatti, una stringa scritta in questo modo ha implicitamente un `const` davanti, perché indica una stringa costante che viene raggiunta attraverso un puntatore non costante. Infatti, se si cerca di manipolare tale stringa tramite tale puntatore viene restituito un errore.
In C++, questa é la forma piú "basica" di stringa, e pertanto andrebbe evitata a meno di circostanze particolari. Le stringhe C++ sono degli oggetti veri e propri, definiti come `std::string`. L'header file `string` contiene diversi metodi per manipolarle.

```
#include <string>

std::string s1;
s1 = "Hello";
std::string s2 = "World!";
```

Le stringhe C sono ancora utilizzate come argomenti dalla riga di comando. Infatti, la sintassi standard[3] della funzione `main` completa é la seguente:

```
int main(int argc, char* argv[])
{
  ...
  return 0;
}
```

`argc` (*argument count*) é una variabile intera e cattura il numero di argomenti passati al programma quando é stato invocato. `argv` (*argument value*) é un array di puntatori, ciascuno facente riferimento ad una stringa, ed a sua volta ciascuna stringa é l'i-esimo argomento passato al programma. L'unica eccezione é `argv[0]`, che é invece il nome dell'eseguibile stesso (pertanto, gli argomenti vanno contati a partire da 1).
Gli argomenti in `argv` sono sempre e comunque stringhe. Per interpretarne il contenuto come tipi di dato primitivi diversi (come `int` o `float`) sono possibili due strade:

- Usare le funzioni di basso livello del C, come `atoi` o `atof`;
- Usare gli oggetti `stringstream` dell'header C++ `sstream`.

```
var_type = std::atoX(argv[i]);
```

```
#import <sstream>
std::stringstream s_name(argv[i]);
type name;
s_name >> name;
```

[3] Alcuni compilatori accettano anche versioni non strettamente conformi a tale standard, ma é comunque best practise aderirvi.

1.6. Casting

Cosí come in (quasi) tutti i linguaggi di programmazione tipizzati, in C++ é possibile fare **casting**, ovvero trasformare il tipo di dato di una variabile in un tipo di dato diverso, purché compatibile. Alcuni cast sono **impliciti**, ovvero dove il compilatore opera "dietro le quinte" un cambio di tipo se questo é in grado di intuirlo da solo. Questo é comodo, perché non é necessario specificare istruzioni aggiuntive, ma puó essere rischioso perché potrebbe diventare difficile ricostruire a ritroso che tale casting é avvenuto. Il cast C **esplicito** ha invece questa forma:

```
var_type1 = (Type1)var_type2
```

C++, per quanto possa utilizzare i due cast sopra citati, possiede i seguenti cast speciali:

```
var_type1 = static_cast<Type1>(var_type2)
var_type1 = const_cast<Type1>(var_type2)
var_type1 = reinterpret_cast<Type1>(var_type2)
var_type1 = dynamic_cast<Type1>(var_type2)
```

- `static_cast` é sostanzialmente analogo al casting esplicito del C;
- `const_cast` é un cast speciale utile per "de-proteggere" i dati, permettendo di accedere ad un dato costante attraverso un puntatore;
- `reinterpret_cast` é un cast speciale che "forza" un cast anche quando questo porta a conclusioni ambigue, di fatto "reinterpretando" il significato dei singoli byte;
- `dynamic_cast` é un cast speciale che permette di fare downcasting in una gerarchia di classi.

```
int i;
double d;
i = static_cast<int>(d);              // Similar to i = (int)d in C fashion

int* pi;
const int* cpi = &i;
pi = static_cast<int*>(cpi);          // NOT allowed, can't edit i through cpi
pi = const_cast<int*>(cpi);           // Allowed

char* c;
c = reinterpret_cast<char*>(&i);      // Allowed, integer now a char sequence
*(c + 2)                              // Editing i byte by byte
```

1.7. Ciclo di vita delle variabili

In C++ esistono diversi modi per allocare le variabili. Diversi modi implicano diversa visibilitá, ovvero sono accessibili in un qualche modo in un punto piuttosto che un altro del programma. I modi sono tre:

- Allocazione **globale**;
- Allocazione **automatica**;
- Allocazione **statica**;
- Allocazione **dinamica**;

Una variabile é globale se non appartiene a nessuna funzione o classe. Tale variabile é accessibile da qualsiasi punto dell'unitá di compilazione corrente, e se dichiarata con il modificatore `extern` é accessibile anche da qualsiasi altra unitá di compilazione che la importa. Esistono all'interno della memoria fintanto che il programma é in esecuzione.

Definire una variabile globale che deve essere accessibile da ogni singola unitá di compilazione del codice é una delle poche situazioni in cui puó avere senso avere un file `main.h`.
Una variabile é automatica se viene allocata e deallocata automaticamente nello/dallo stack. Sono le variabili comunemente intese, che si trovano all'interno di una funzione e che esistono solamente fintanto che tale funzione é in esecuzione. Sono (potenzialmente) accessibili solamente dal blocco in cui sono state definite. Naturalmente, le variabili automatiche dichiarate all'interno di `main()` esistono fino alla fine dell'esecuzione del programma.

```
void f(int param)      // Will exist as long as f exists
{
  int i;             // Will exist as long as f exists

  if (i) {
    int k;         // Will exist as long as the if block exists
    ...
  }

  ...
}

int main()
{
  int j;             // Will exist as long as main exists
                     // (Until the program stops)
}
```

Una variabile é statica se é dichiarata con il modificatore `static`. Sebbene la loro visibilitá sia ristretta a quella del blocco in cui sono state definite, esisteranno comunque in memoria fino alla fine del programma. Combinano il lifespan di una variabile globale con la visibilitá di una variabile dinamica.
Una variabile é dinamica se ne viene richiesto esplicitamente il quantitativo di memoria da allocare e la loro deallocazione. Tali dati non si trovano, come le variabili precedenti, sullo stack, ma bensí sullo heap, pertanto é necessario gestirne l'esistenza in maniera oculata (si rischia di saturare la memoria con dati inutili). I dati dinamici, sebbene possano essere di qualsiasi tipo, sono particolarmente utili per quando é necessario allocare molta memoria, dato che la dimensione dello stack é in genere molto contenuta.
Un dato dinamico viene creato mediante la keyword `new` e distrutto mediante la keyword `delete`.

```
Object* name = new Object;                    delete name;
```

`new` restituisce un puntatore all'oggetto dinamico cosí creato, e tale puntatore é l'unico modo per poter accedere a tale oggetto (il puntatore potrebbe comunque essere memorizzato sullo stack). Questo significa che se tale puntatore, per qualche motivo, perde il riferimento a tale oggetto, questo rimane nello heap senza piú possibilitá di accedervi, e finché il programma non termina l'area di memoria che questo occupa non puó essere sovrascritta, generando un **orfano**. Infatti, a differenza di altri linguaggi di programmazione, in C++ non esiste un **garbage collector**[4].

[4] É peró possibile estendere C++ aggiungendo un garbage collector esterno, come Boehm GC.

```
struct Obj {
  double d;
  int arr[1024];
};

Obj* o = new Obj;      // o is a pointer to a value in heap

o->d = 10.23;

o = new Obj;           // Allowed, but now old values are lost in heap

new Obj;               // Allowed, but memory is allocated for nothing
```

`delete` contrassegna il contenuto dinamico associato al puntatore come scrivibile da parte del sistema operativo. Naturalmente, se si cerca di chiamare `delete` su un puntatore su cui é giá stato chiamato si puó avere un comportamento indefinito, perché i vecchi valori *potrebbero* essere giá stati sovrascritti da altri dati. Pertanto, una best practice é, dopo aver chiamato `delete` riassegnare il puntatore a `nullptr`, perché chiamando `delete` su un puntatore nullo non succede nulla. Cercando di chiamare `delete` su un puntatore che non si riferisce a dei dati dinamici viene restituito un errore a runtime.

```
struct Obj {
  double d;
  int arr[1024];
};

Obj* o = new Obj;

delete o;            // Values in heap related to o are flagged as removable

delete o;            // Allowed, but VERY dangerous
```

Gli array sono spesso allocati dinamicamicamente, perché in genere il loro contenuto richiede molto spazio. Occorre peró prestare attenzione al fatto che la deallocazione del contenuto dinamico va fatta con `delete[]`, che libera ricorsivamente tutto il contenuto che si trova all'interno, altrimenti solamente il contenuto che si trova in prima posizione viene liberato. É anche possibile allocare dinamicamente array multidimensionali usando puntatori a puntatori, anche se a tale livello di complessitá diventa molto piú ragionevole utilizzare una classe.

```
int size1 = 5, size2 = 3;

int** array = new int*[size1];

for (int i = 0, i < size1, i++)
    array[i] = new int[size2];

array[3][2] = 7;

for (int i = 0, i < size1, i++)
    delete[] array[i];

delete[] array;
```

Un modo alternativo per raggruppare semanticamente funzioni, variabili, dichiarazioni e tipi é quello del namespace, che assegna una etichetta univoca a ciascuna di queste entitá. Tale etichetta diviene parte del nome dell'entitá stessa, e due entitá che hanno lo stesso nome ma diverso namespace saranno comunque trattati come distinti (naturalmente, all'interno di uno stesso namespace non possono esistere due entitá con lo stesso nome).

```
namespace name_s
{
...
}
```

```
name_s::entity
```

```
namespace First
{
  int smth;
  int g() {...}
}

namespace Second
{
  int smth;        // Allowed, namespace is not the same
}

First::smth = 10;
Second::smth = 5;
int smth = 2;        // Allowed, signature is still different

int x = First::g();
```

La keyword `using` permette di specificare che, da quel momento in poi, tutte le entitá che vengono nominate, se non hanno un namespace associato, *potrebbero* avere sottinteso il namespace passato come argomento. Puó anche essere usato per specificare una singola entitá che deve essere "estratta" dal namespace, senza riferirvisi con il nome del namespace.

```
namespace First
{
  int s;
}

int main()
{
  using First::s;       // Now First::s is just s
  int s = 10;           // NOT allowed
  s = 10;               // Allowed
}
```

1.8. Funzioni

In C++, esistono tre modi per passare un parametro ad una funzione. Di base, quando un valore viene passato ad una funzione, il parametro di tale funzione viene inizializzato con il valore passato. Al di lá di questo, che é comune a tutti i modi, i modi sono i seguenti:

- **Passaggio per valore**, in cui viene semplicemente creata una copia locale alla funzione del parametro che viene passato, e tale copia non influenza in alcun modo la versione originale del dato nota alla funzione chiamante;
- **Passaggio per puntatore**, in cui viene passato un puntatore ad una risorsa nota alla funzione chiamante. Questo significa che se la funzione chiamata manipola il dato associato al puntatore, tale valore viene modificato anche rispetto alla funzione chiamante. Se nella funzione chiamata viene modificato il puntatore in sé, sia il puntatore originale che l'oggetto a cui si riferisce rimangono comunque intatti;
- **Passaggio per referenza**, in cui viene passata una reference ad un dato noto alla funzione chiamante; se la funzione chiamata lo modifica, tale modifica si ripercuote anche sul dato originale.

```cpp
#include <iostream>

void fun1(int j)
{
  j = j / 2;
}

void fun2(int* j)
{
  *j = *j + 3;
  int x = 30;
  j = &x;          // Nothing changes for i
}

void fun3(int& j)
{
  j = j * 2;
  int& x = j;
  x++;             // Something changed for i
}

int main()
{
  int i = 10;
  fun1(i);          // Nothing changes

  int* g = &i;
  fun2(g);         // i is now 13

  int& r = i;
  fun3(r);         // i is now 27

  return 0;
}
```