# Indice

# 1. Introduzione

## 1.1. Hello, World!

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:

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

**Exercise 1.1.1:**

```
#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).

## 1.2. Tipi di dato elementari

Ogni nome (identificatore) in un programma C++ deve aver associato un **tipo**. Il tipo di un dato determina sia la quantitá di memoria assegnata al dato, sia quali operazioni possono essere eseguite sul dato, e come tali operazioni devono venire interpretate.
Nel C++ si distinguono i seguenti tipi di dato primitivi:

- **numeri interi**;
- **numeri decimali**;
- **booleani**;

- **caratteri**;
- **enumerativi**;
- `void`;
- **puntatori**;
- **reference**.

### 1.2.1. Numeri interi

I numeri interi sono rappresentati attraverso tre tipi di variabili: `int`, `unsigned int` e `signed int`. Esistono poi tre sottotipi di intero, rispetto alla dimensione: `int` (dimensione standard), `long int` (dimensione doppia rispetto a `int`) e `short int` (dimensione dimezzata rispetto a `int`). `long` puó venire usata come abbreviazione per `long int`, cosí come `short` lo é per `short int`. Quanto effettivamente occupino dei dati memorizzati con tali tipi dipende dall'implementazione in uso, ma in genere la dimensione di un dato `int` non puó essere inferiore a 16 bit.
I modificatori per il segno e per la dimensione possono essere combinati. Se non viene specificato direttamente, un tipo `int` viene inteso come un intero con segno. Pertanto, non esiste alcuna differenza fra `signed int` e `int`, semplicemente il secondo denota esplicitamente la presenza del segno. Il compilatore restituisce un warning se si cerca di assegnare un valore ad un tipo decimale che eccede la sua capacitá. La dimensione di un intero `signed`, `unsigned` o non specificato é sempre la stessa, ció che cambia é l'intervallo di valori rappresentabili.
Se un numero intero viene riportato cosí com'é, il compilatore assume che sia un numero in rappresentazione decimale. Se al numero viene anteposto `0`, questo viene inteso in rappresentazione ottale. Se al numero viene anteposto `0x`, questo viene inteso come in rappresentazione esadecimale. Se al sumero viene postposto `U`, viene inteso come numero senza segno, mentre se al numero viene postposto `L`, viene inteso come avente modificatore `long`.

### 1.2.2. Numeri decimali

Un valore decimale é un valore numerico floating point. Esistono in tre dimensioni: `float` (precisione singola), `double` (precisione doppia) e `long double` (precisione estesa). Quanto effettivamente occupino dei dati memorizzati con tali tipi dipende dall'implementazione in uso, ma in genere la dimensione di un dato `double` non puó essere inferiore a 32 bit.
Se non viene specificato diversamente, un dato floating-point é di tipo `double`. Il compilatore restituisce un warning se si cerca di assegnare un valore ad un tipo decimale che eccede la sua capacitá.

### 1.2.3. Booleani

Un valore booleano puó assumere solamente due valori, `true` e `false`. Un booleano puó essere usato per esprimere il risultato di una operazione logica. Internamente, un valore booleano viene considerato come un numero intero che puó avere esclusivamente valore 1 (`true`) oppure 0 (`false`). Infatti, stampando un booleano tramite `std::cout` viene restituito 1 oppure 0.
Se utilizzato in una espressione aritmetica, un valore booleano viene trattato come un intero (1 se `true`, 0 se `false`). D'altra parte, un numero intero che viene convertito in un booleano diventa `true` se ha un qualsiasi valore che non sia 0 e `false` altrimenti. Anche un puntatore puó essere convertito in un booleano. Un puntatore a `nullptr` viene convertito come `false` e come `true` altrimenti.

**Exercise 1.2.3.1:**

```
int x = 10;
int y = 5;

bool b = (x > y);     // True, since 10 is greater than 5
bool c = !(x != y)    // False, since 10 is not 5

b = -3;               // True, since -3 is not 0
c = &x;               // True, since it's not a null pointer
```

### 1.2.4. Caratteri

Una variabile di tipo `char` contiene un singolo carattere del set di caratteri supportato dall'implementazione in uso. Sebbene il set di caratteri supportato puó variare molto da implementazione a implementazione, é possibile

assumere che questo sia almeno 7-bit ASCII, e che contenga quindi almeno le dieci cifre decimali, le 26 lettere dell'alfabeto inglese ed i principali segni di punteggiatura.
Internamente, un carattere viene considerato come un numero intero costante. Riferirsi ai caratteri in quanto tali anziché riferirvisi direttamente mediante i numeri interi che li rappresentano permette di introdurre un livello di astrazione maggiore ed evitare di considerare gli specifici dettagli implementativi. Essendo numeri interi, possono essere manipolati aritmeticamente come di consueto, ma se stampati con `std::cout` vengono resi come caratteri.
La possibilitá di convertire un `char` in un `int` lascia aperta una questione: `char` é `signed` oppure `unsigned`? Un `char` di 8 bit potrebbe contenere un valore fra $-128$ e 127 oppure fra 0 e 255 a seconda se abbia o non abbia il segno. Sfortunatamente, la scelta non é standardizzata, ma dipende dall'implementazione. C++ permete di dichiarare una variabile di tipo `char` specificatamente come `signed` per rifarsi alla prima rappresentazione e `unsigned` per la seconda. Fortunatamente, la maggior parte dei caratteri di uso comune si trovano fra 0 e 127, pertanto la differenza é spesso irrilevante.

**Exercise 1.2.4.1:**

```
char a = 'b';    // a is 'b', which is 98
a /= 2;          // a is '1', which is 49
a++;             // a is '2', which is 50
a *= 10;         // a is out of range
```

I caratteri Unicode non possono essere rappresentati mediante `char`, perché richiedono troppa memoria. Per usecase di questo tipo é possibile rifarsi al tipo di dato `wchar_t`.
Il tipo `char` é la piú piccola entitá che puó essere indirizzata, pertanto non puó avere dimensione inferiore a 8 bit. `char` é il tipo di dato che piú di tutti ha la stessa dimensione nelle diverse implementazioni; per questo motivo, la dimensione di un `char` é quella che viene utilizzata dal C++ come "unitá di misura" per esprimere la dimensione degli altri tipi di dato. In particolare, la funzione della libreria standard `sizeof()` restituisce la dimensione in multipli di `char` del tipo di dato passato come argomento (in alternativa, é possibile recuperare queste informazioni dal file della libreria standard `#limits.h`).

**Exercise 1.2.4.2:**

```
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

#include <limits.h>
int main()
{
  std::cout << "largest float ==> " << numeric_limits<float>::max()
  << ", char is signed ==> " << numeric_limits<char>::is_signed()
  << std::endl;
}
```

Queste sono le relazioni fra le dimensioni fra i tipi che sono consistenti in tutte le implementazioni:

---

1 ≡ sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long)
1 ≤ sizeof(bool) ≤ sizeof(long)
sizeof(char) ≤ sizeof(wchar_t) ≤ sizeof(long)
sizeof(float) ≤ sizeof(double) ≤ sizeof(long double)
sizeof(char) ≡ sizeof(signed char) ≡ sizeof(unsigned char)
sizeof(int) ≡ sizeof(signed int) ≡ sizeof(unsigned int)
sizeof(short int) ≡ sizeof(signed short int) ≡ sizeof(unsigned short int)
sizeof(long int) ≡ sizeof(signed long int) ≡ sizeof(unsigned long int)

---

### 1.2.5. Enumerativi

Un `enum` é un tipo di dato che consente di associare in maniera automatica dei valori interi costanti a delle sequenze alfanumeriche, di modo da usarle come "segnaposto" per dei valori legati da una qualche semantica.

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

Ogni elemento che figura fra le quadre viene chiamato *enumeratore*, ed il valore che vi viene associato puó venire specificato oppure lasciato dedurre al compilatore. Nel secondo caso, a tutti gli enumeratori che vengono dopo l'ultimo a cui é stato associato un valore esplicitamente viene assegnato il valore a questo successivo. Se nessun valore viene specificato, agli enumeratori assegnati i numeri $1, 2, 3, \ldots$,
Ciascun `enum` va a costituire un tipo di dato a sé stante. Ovvero, il tipo di dato di un enumeratore é il nome dello specifico `enum` di cui fa parte, non `enum` generico. In una operazione aritmetica, un enumeratore viene considerato come il valore che gli é stato assegnato, e si comporta come un normale intero. Tentando di stamparlo con `std::cout` viene restituito il valore numerico, non il nome dell'enumeratore.
Il *range* di una enumerazione contiene i valori di tutti i suoi enumeratori arrotondati alla piú grande vicina potenza di due meno uno. L'estremo sinistro del range viene scalato a 0 se a nessun enumeratore é associato un valore negativo e alla piú piccola potenza negativa di due in caso contrario. Il `sizeof` di una enumerazione é la stessa di un tipo di dato sufficientemente grande da contenerne il range e non maggiore di `sizeof(int)`, a meno che uno o piú dei suoi enumeratori ha assegnato un valore troppo grande per essere rappresentato in un `int` o `unsigned int`.

**Exercise 1.2.5.1:**

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

### 1.2.6. Void

Il tipo `void` rappresenta una mancanza di informazione e non puó essere assegnato direttamente ad una variabile. Puó essere usato esclusivamente in due contesti: per contrassegnare che una funzione non ritorna alcun valore oppure per specificare che un puntatore si riferisce ad una variabile il cui tipo non é noto a priori[1].

**Exercise 1.2.6.1:**

```
void x;       // NOT allowed
void f();     // f() is a function that does not return anything
void* y;      // y is a pointer to an object of unknown type
```

### 1.2.7. Puntatori

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, in quanto gli indirizzi di memoria hanno tutti la stessa dimensione.

[1] Per indicare che una funzione non ritorna alcun valore non é possibile semplicemente non riportare il tipo della funzione. Questo sia per mantenere la retrocompatibilitá con il linguaggio C, sia perché altrimenti la grammatica del C++ diverrebbe piú complessa.

Un puntatore viene dichiarato come una normale variabile di tale tipo, ma postponendo `*` al tipo[2]. 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
variable_type* pointer_name = &variable_to_point
variable_type variable_name = *pointer_name
```

Il valore di default per un puntatore é `NULL` oppure (standard C++11) `nullptr`. É anche ammesso che un puntatore punti a 0, dato che una istruzione di quel tipo viene interpretata dal compilatore in maniera speciale (non venendo mai allocato nulla all'indirizzo 0).

**Exercise 1.2.7.1:**

```
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. 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).

**Exercise 1.2.7.2:**

```
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
```

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 consi-

[2]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.

gliabile (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.
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 possibile avere dei puntatori a dei puntatori. Inoltre, nulla vieta di avere piú di un puntatore legato alla stessa area di memoria.

**Exercise 1.2.7.3:**

```
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.

**Exercise 1.2.7.4:**

```
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;

### 1.2.8. Reference

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
```

**Exercise 1.2.8.1:**

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

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

## 1.3. Dichiarazione e definizione

Nel C++ si fa distinzione fra **dichiarazione** e definizione di una entitá (che sia un tipo, un oggetto o una funzione). Una dichiarazione introduce un nome (identificatore) e descrive il suo tipo. Una dichiarazione é ció che il compilatore deve sapere si riferisce a tale identificatore. Una **definizione** é l'istanziazione/implementazione di tale identificatore. Una definizione é ció che il linker deve sapere si riferisce a tale identificatore.
Dichiarare una variabile significa notificare al compilatore che tale variabile esiste ed ha un certo nome. Definire una variabile significa richiedere esplicitamente di allocare la memoria necessaria a contenerne il valore.
La definizione é distinta dall'**inizializzazione**, ovvero assegnare un valore iniziale. Una variabile puó essere sia inizializzata mentre la si definisce, oppure puó essere fatto separatamente. Se una variabile non viene inizializzata, il suo valore potrebbe essere indeterminato.

**Exercise 1.3.1:**

```
// Declaration without definition

extern int error_number;
struct User;
class Person;

// Declaration with definition

int error_number;
double funny = 6.9;
char* name = "Hello, World!";
struct Date {int day, int month, int year;};
```

Dichiarare una funzione significa riportarne la **firma**, ovvero il tipo del valore di ritorno, il suo nome ed il numero e tipo dei suoi argomenti. Definire una funzione significa, oltre a dichiararla, anche specificarne il corpo.

**Exercise 1.3.2:**

```
// Declaration without definition

double square_root(double n);

// Declaration with definition

double square_root(double n)
{
  // body goes here...

  return ...;
}
```

Se un nome viene utilizzato all'interno di un programma ma non viene dichiarato, il programma non puó essere compilato. Se un nome viene utilizzato all'interno di un programma ma non viene definito, il programma

puó comunque essere compilato, ma non puó essere linkato. Una definizione implica anche una dichiarazione, il contrario non é necessariamente vero.
Dichiarare piú volte una stessa variable/funzione allo stesso modo non dá errore, perché si sta semplicemente specificando piú volte al compilatore che quella entitá "esiste". Viene restituito un messaggio di errore durante la compilazione nel caso in cui che vi siano piú dichiarazioni di una stessa variabile/funzione se queste sono discordanti fra loro. Definire piú volte una stessa variable/funzione restituisce sempre un messaggio di errore durante la compilazione. Due dichiarazioni/definizioni per variabili con lo stesso nome, anche se di tipo diverso, non sono ammesse.

**Exercise 1.3.3:**

```
int x;
int x;                  // NOT allowed

float y;
char y;                 // NOT allowed

extern double pi;
extern double pi;       // Allowed

extern int err_code;
extern long err_code;   // NOT allowed
```

Una dichiarazione é costituita da quattro componenti: una specifica opzionale, un tipo base[3], un dichiaratore, una inizializzazione opzionale. Ad eccezione delle funzioni e dei namespace, una dichiarazione termina con un punto e virgola. La specifica é costituita da una parola chiave come `virtual` o `extern`, che riporta una caratteristica dell'identificatore che esula dal suo tipo. Un dichiaratore é costituito da un nome e opzionalmente da un operatore. Gli operatori possono essere sia prefissi che postfissi; gli operatori prefissi, nella grammatica, hanno prioritá maggiore. Gli operatori piú comuni sono:

| | | |
|---|---|---|
| `*` | puntatore | prefisso (prima del dichiaratore) |
| `&` | referenza | prefisso (prima del dichiaratore) |
| `[]` | array | postfisso (dopo il dichiaratore) |
| `()` | funzione | postfisso (dopo il dichiaratore) |

É possibile dichiarare piú nomi in una sola dichiarazione. La dichiarazione é semplicemente costituita da una lista di dichiaratori separati da virgole. Gli operatori non vengono estesi a tutti i dichiaratori in una dichiarazione multipla, sono legati esclusivamente al dichiaratore in cui figurano.

**Exercise 1.3.4:**

```
int x, y, z;      // int x; int y; int z;
int *x, y, z;     // int *x; int y; int z;
int x, *y, z;     // int x; int *y; int z;
```

Un nome (identificatore) consiste di una sequenza di lettere e cifre. Il primo carattere di un nome deve per forza essere una lettera; il carattere `_` viene considerato come lettera. Lo standard C++ non impone un limite alla lunghezza di un nome, ma un compilatore o un linker potrebbe farlo. Alcune implementazioni del C++ permettono di utilizzare caratteri speciali (come `$`) nei nomi, ma é best practice comunque evitarlo per aumentare la compatibilitá. I nomi delle parole chiave della grammatica del C++ (`int`, `if`, `throw`, ecc...) non possono essere usati come nomi.

[3]Nei precedenti standard del C e del C++, il tipo base era opzionale, e se veniva omesso veniva implicitamente considerato `int`. Questa feature é stata rimossa.

**Exercise 1.3.5:**

```
// Valid identifiers
hello    ___    a123    INT    Hello_World    _x_y_z_    tHiSnAmE    __0__
// Non valid identifiers
Hello World    012345    float    $name    var.iable    else
```

I nomi che iniziano con il carattere `_` sono in genere riservati per l'implementazione e l'ambiente run-time, ed é pertanto best practice non usarli. Il carattere di spazio serve a separare i token della grammatica, pertanto due sequenze di caratteri inframezzati da spazi vengono sempre considerati due token distinti. C++ é case sensitive, pertanto le lettere maiuscole e minuscole sono considerate caratteri distinti.
É considerata bad practice scegliere due identificatori distinti che differiscono per pochi caratteri. I nomi di grandi scope é bene che abbiano un nome lungo di modo da metterli in evidenza, mentre le variabili poco importanti di scope piccoli possono essere chiamati anche con singole lettere. In genere, é piú utile dare nomi che riflettono la semantica che ha quell'entitá, piuttosto che il modo in cui viene implementata, perché rende il codice piú leggibile.

## 1.4. Tipi di dato composti

Combinando fra loro tipi primiti, é possibile costruire tipi di dato composti. Nel C++, i principali dati composti sono tre: **array**, **stringhe** e **strutture**.

### 1.4.1. Array

Un **array** é una sequenza di dati dello stesso tipo, memorizzati in aree di memoria contigue chiamate **celle**. Gli array sono utili 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. Dentro alle parentesi quadre é opzionalmente riportata la sua **dimensione**, ovvero il numero di celle di cui é costituito. 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, mentre l'opposto é ammissibile (le celle rimaste vuote vengono automaticamente riempite con 0).

```
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
```

Se la dimensione di un array non viene riportata, il compilatore la "deduce" sulla base del numero di elementi con il quale é stato inizializzato (se é stato inizializzato con $n$ elementi, allora gli viene assegnata in automatico la dimensione $n$). La dimensione di un array, se riportata esplicitamente, deve essere un valore costante, non il contenuto di una variabile, anche se il valore di tale variabile é noto, e non é ammesso che questa venga determinata a runtime[4]. Una volta fissata la dimensione di un array, non é piú possibile cambiarla (estenderla o restringerla).
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. Nonostante questo, é comunque considerato un comportamento semanticamente scorretto, perché (come nel caso dei puntatori) si sta cercando di accedere ad un'area di memoria non assegnata al programma, e gli effetti sono imprevedibili.

[4] Molti compilatori hanno introdotto una estesione che permette di avere array la cui dimensione viene calcolata a runtime (**variable-length arrays** o **VLA**). Per forzare il compilatore a non usare questa estesione, molti mettono a disposizione il flag `-pedantic` che garantisce aderenza totale allo standard ISO C++.

**Exercise 1.4.1.1:**

```
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...
```

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]
```

**Exercise 1.4.1.2:**

```
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.

**Exercise 1.4.1.3:**

```
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.

**Exercise 1.4.1.4:**

```
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
```

Usare i puntatori e gli array puó portare a scritture molto convolute. É possibile usare la seguente regola pratica per interpretare una scrittura di difficile comprensione:

- Si inizia dal nome della variabile;
- Ci si sposta a destra se possibile: se esiste qualcosa a destra della variabile (come `[]` o `()`), lo si processa;
- Ci si sposta a sinistra se necessario: se non c'é nulla a destra, ci si sposta a sinistra. In genere questo accade se c'é da processare `*`, che indica un puntatore;
- Si ripete finché la dichiarazione é terminata.

**Exercise 1.4.1.5:**

```
int i;                  // i as an int
int *i;                 // i as a pointer to an int
int **i;                // i is a pointer to a pointer to an int
int *i[5];              // i is an array of 5 pointers to int
int (*i)[5];            // i is a pointer to an array of 5 ints
```

### 1.4.2. 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`).
Una stringa puó essere inizializzata in due modi. Il primo é quello di usare la sintassi tipica degli array, e quindi riportare ordinatamente ogni carattere fra parentesi graffe; in questo caso, occorre che l'ultimo sia `\0`. Il secondo é quello di riportare semplicemente i caratteri fra doppi apici; il compilatore converte implicitamente

tale rappresentazione in lista di caratteri e aggiunge `\0` alla fine. Una stringa particolarmente lunga puó essere inizializzata "spezzandola" in piú sequenze racchiuse fra doppi apici: il compilatore si incarica di concatenarle automaticamente.

**Exercise 1.4.2.1:**

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

La 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 si ottiene un errore a runtime. Se si dichiara invece una stringa come array di caratteri, é possibile modificarla come fosse un normale array senza effetti collaterali.

**Exercise 1.4.2.2:**

```
char* m = "More";
m[2] = 'l';            // Allowed, but...

char d[] = "Down";
d[1] = 'a';            // Allowed
```

Essendo degli array di caratteri (per quanto gestiti in maniera speciale), una stringa si comporta come tale. Ovvero, se la sua dimensione viene fissata ma viene inizializzata con una stringa con meno caratteri di tale dimensione, i caratteri restanti vengono riempiti con il carattere nullo. Se la dimensione non viene specificata, viene dedotta dal compilatore sulla base di come viene inizializzata. Non é possibile inizializzare una stringa se non mentre la si dichiara.
Per inserire caratteri speciali all'interno di una stringa, é possibile usare l'escape character '\'. In particolare, alcuni caratteri speciali, come quello di tabulazione o di a capo, hanno delle escape sequence dedicate ( `\t` e `\n` rispettivamente, in questo caso). É possibile inserire volutamente il carattere nullo all'interno di una stringa, ma la maggior parte delle funzioni che manipolano stringhe non saranno in grado di notarlo.
Una stringa con prefisso `L` é una stringa di wide chars. Di conseguenza, il suo tipo é `const wchar_t`.
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.

**Exercise 1.4.2.3:**

```
#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[5] della funzione `main` completa é la seguente:

```
int main(int argc, char* argv[])
{
  ...
```

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

```
  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;
```

### 1.4.3. Struct

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};
```

**Exercise 1.4.3.1:**

```
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
```

**Exercise 1.4.3.2:**

```
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`.
Due `struct` diverse, anche se hanno gli stessi tipi e sono inizializzate allo stesso modo, sono comunque considerati due tipi distinti. Questo perché una variabile dichiarata come una certa `struct` ha per tipo tale `struct`.
Inoltre, anche una `struct` formata da un solo campo, anche se tale campo é un tipo di dato primitivo, é comunque considerata un tipo distinto da quest'ultimo.

**Exercise 1.4.3.3:**

```
struct S1 {int a;};
struct S2 {int a;};

S1 x;
S2 y = x;    // NOT Allowed, different types

int z = x;   // NOT Allowed, even if x is just a int
```

Una `struct` non puó essere definita ricorsivamente, ovvero non puó contenere a sua volta una `struct` dello stesso tipo come campo. Questo perché il nome di un tipo diventa disponibile non solamente dopo essere stato dichiarato, ma dopo che lo si definisce per la prima volta, perché il compilatore non puó sapere quanta memoria allocare. É peró possibile avere una `struct` che contiene un puntatore ad una `struct` dello stesso tipo, perché la memoria allocata per un puntatore é sempre la stessa, ed il problema non si pone.

**Exercise 1.4.3.4:**

```
// NOT allowed, incomplete type
struct Node {
  int value;
  Node Left;
  Node Right;
};

// Allowed
struct Node {
  int value;
  Node* Left;
  Node* Right;
};
```

In generale, il nome di una `struct` che é stata dichiarata ma non definita puó essere usato solamente nei casi in cui non é necessario conoscerne la dimensione o uno dei suoi campi.

**Exercise 1.4.3.5:**

```
struct List;

// NOT allowed, incomplete type
struct Table {
    int hash;
    List L;
};

// Allowed, but unusable until List is defined
struct Table {
    int hash;
    List* L;
};

void  f(List L);         // Allowed
List  g(int x, char* S); // Allowed
List  h(double y);       // Allowed
List  q(List L);         // Allowed
List* p(List* L);        // Allowed

int main(int argc, char* argv[])
{
    List* L;         // Allowed, only pointers involved
    List LL;         // NOT allowed, incomplete type
    f(*L);           // NOT allowed, size is needed to be returned
    g(5, "Hello");   // NOT allowed, size is needed to be passed
    h(3.14).n = 0;   // NOT allowed, fields are unknown
    List* P = p(L);  // Allowed, only pointers involved

    return 0;
}
```

## 1.5. Scope e namespace

Prende il nome di **blocco** qualsiasi sezione di codice racchiusa all'interno di due parentesi graffe. Ogni blocco induce uno **scope**, ovvero uno "spazio" entro al quale le variabili possono essere dichiarate. Una variabile dichiarata all'interno di un blocco si dice che é **locale** al blocco.
All'interno di uno stesso scope, le variabili non possono essere definite piú di una volta, mentre due variabili possono essere definite uguali in due scope diversi. Piú blocchi, e di conseguenza piú scope, possono essere contenuti l'uno nell'altro. Le definizioni non si "trasmettono" attraverso gli scope piú interni: due variabili con lo stesso nome e lo stesso tipo, se si trovano in due scope diversi (che siano disgiunti o l'uno contenuto nell'altro) sono considerate due entitá diverse. Gli argomenti di una funzione sono considerati facenti parte del blocco indotto dalla funzione.

**Exercise 1.5.1:**

```
{
  int x = 10;

  {
    int x = 5;                    // Allowed: different scope
    std::cout << x << std::endl;  // Prints 5
  }

  std::cout << x << std::endl;      // Prints 10
}

void f(int y)
{
  int x;                          // Allowed: different scope
  int y;                          // NOT allowed
}
```

Un **namespace** permette di raggruppare semanticamente funzioni, variabili, dichiarazioni e tipi, che possono trovarsi anche in file diversi, assegnandovi una etichetta univoca.
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). I confini del namespace sono determinati dal blocco di codice indotto dal namespace: per specificare che con una certa entitá del namespace al di fuori dello stesso é necessario utilizzare l'operatore `::`. É possibile riferirsi a tutto ció che appartiene ad un namespace da dentro il namespace stesso senza dover specificare `::`, perché é sottointeso.

```
namespace name_s                          name_s::entity
{
...
}
```

Un namespace particolarmente importante é `std`, quello della libreria standard del C++. Non a caso, diverse funzioni molto usate come `cin` e `cout` appartengono a tale namespace.

**Exercise 1.5.2:**

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

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

First::s = 10;
Second::s = 5;

int x = First::s;     // x = 10
```

I namespace permettono la modularizzazione del codice, separandolo in diverse componenti atomiche che dipendono fra di loro. Permettono inoltre di evitare conflitti fra nomi, dato che due entitá con lo stesso nome di due namespace diversi potrebbero avere due significati diversi. Infine, permettono di oscurare i dettagli implementativi di una funzione o di una classe, presentando solamente una interfaccia e la sua semantica.

Una entitá appartenente ad un namespace puó essere dichiarata e definita all'interno del blocco indotto dal namespace stesso, oppure puó essere dichiarata all'interno del blocco e (riferendovisi usando `::`) definita al di fuori. Non é peró possibile dichiarare una entitá membro di un namespace al di fuori del blocco da questi indotto. Le regole degli scope sono valide anche per i namespace, pertanto in uno stesso namespace non possono esserci due definizioni uguali.

**Exercise 1.5.3:**

```
namespace AAA
{
  int f();
}

int AAA::f()    // Allowed
{
  ...
}

int AAA::g()     // NOT allowed
{
  ...
}
```

La keyword `using` permette di introdurre un sinonimo locale per un elemento appartenente ad un namespace diverso da quello corrente. Dall'uso di `using` in poi e per tutta la lunghezza dello scope/namespace, é possibile riferirsi a quella entitá omettendo l'operatore `::`. Puó anche essere usato per indicare che, dall'uso di `using` in poi e per tutta la lunghezza dello scope, ogni entitá facente parte di un certo namespace diventa disponibile nello scope/namespace corrente omettendo `::`.
`using` puó semplificare il codice, risparmiando diversi `::` ridondanti, ma puó anche rendere il codice piú confuso, perché gli elementi del namespace "importato" e gli elementi locali diventano di fatto indistinguibili. `using` puó anche essere usato nello scope globale, di fatto stabilendo che lungo tutto il codice gli elementi di tale namespace sono accessibili globalmente omettendo `::`; questa scelta é considerata bad practice, ed é preferibile evitarla. Anche utilizzare `using` all'interno di un header file é considerata bad practice, perché tale scelta si propaga lungo tutto il codice che lo importa. `using` puó essere utile nella definizione dei namespace, per importare con piú facilitá entitá da altri namespace.

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

```
using name_s::entity
```

**Exercise 1.5.4:**

```
namespace First
{
  int s;
}

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

I namespace devono per forza essere dichiarati globalmente, pertanto ogni file che importa il file che contiene la sua dichiarazione lo ha disponibile. Inoltre, i namespace possono essere dichiarati piú volte, anche in file diversi;

quando il codice viene compilato, le diverse occorrenze di uno stesso namespace e le loro componenti vengono unite in una sola.

**Exercise 1.5.5:**

```
namespace Ex
{
  int x;
}

namespace Ex     // Allowed
{
  int s;
}

int main()
{
  Ex::s = 30;
  Ex::x = 50;

  return 0;
}
```

I namespace possono anche essere usati per "occultare" delle dichiarazioni, di modo che tale namespace sia accessibile soltanto nello scope locale e non al di fuori e di modo che le dichiarazioni in tale namespace non interferiscano con l'esterno. Per fare questo é possibile utilizzare i **namespace anonimi**, che vengono introdotti dichiarando normalmente un namespace ma omettendone il nome. Tali namespace sono inaccessibili agli altri file, nemmeno se importano il file con tale definizione.

**Exercise 1.5.6:**

```
namespace
{
  int s;
}

int main()
{
  s = 10;    // Allowed
}
```

Ad un namespace puó venire assegnato un alias locale per semplificare il codice, riferendosi ad un namespace con un nome piú comodo. Tale nome é valido soltanto localmente, non é una ridefinizione, il nome originale rimane intatto.

```
namespace new_name = old_name;
```

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.

**Exercise 1.5.7**: 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);
```

## 1.6. Typedef, const, Casting

`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.

```c
typedef old_name new_name;
```

**Exercise 1.6.1**:

```c
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[6]. Se si tenta di aggiungere `const` ad una variabile che non viene inizializzata quando viene dichiarata viene restituito un messaggio di errore.

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

**Exercise 1.6.2**:

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

pi = 3.1415;                 // NOT Allowed

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

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. Non é peró vero il contrario: se una variabile viene dichiarata `const` non puó essere referenziata, perché questo violerebbe il senso stesso di averla dichiarata in quel modo.

[6] `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.

**Exercise 1.6.3:**

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

const char x = 'x';
char& y = x;                // 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.

**Exercise 1.6.4:**

```
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
```

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.

**Exercise 1.6.5:**

```
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

Inizializzare una variabile significa assegnarle un valore per la prima volta. Se una variabile non viene inizializzata, potrebbe o potrebbe non venirle essere assegnato un valore di default. Le variabili globali, le variabili membro di un certo namespace oppure le variabili locali dichiarate `static` vengono implicitamente inizializzate con un valore "neutro" che dipende dal tipo di variabile (0 per `int`, 0.0 per `double`, ecc...). Le variabili locali non dichiarate `static` e le variabili allocate dinamicamente non vengono inizializzate di default, ed occorre farlo esplicitamente. I membri degli array e delle strutture possono venire inizializzati oppure no a seconda che l'array o la struttura a cui appartengono é o non é `static`. Una istanziazione di una classe potrebbe (ed é bene che ce l'abbia) avere un costruttore che assegna valori di default agli attributi della classe.
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 quattro:

- 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 da qualsiasi altra unitá di compilazione che importa il (un) file in cui é dichiarata. Le variabili globali esistono all'interno della memoria fintanto che il programma é in esecuzione.
Le variabili dichiarate globalmente appartengono ad un proprio namespace, detto **namespace globale**. É possibile specificare che ci si sta riferendo a delle entitá che appartengono al namespace globale mediante `::` senza riportare alcun nome. In genere, questo non é necessario, perché tutto ció che non ha un namespace associato (se esiste) viene cercato o nel namespace globale o nello scope in cui ci si trova. L'unica situazione in cui `::` permette effetivamente di disambiguare si ha quando ci si vuole riferire ad una entitá del namespace globale che una un conflitto di nomi con una entitá del namespace o dello scope in uso. É comunque considerata bad practice riusare gli stessi nomi e tipi per variabili globali e locali, a meno di avere un motivo ragionevole per farlo.

**Exercise 1.7.1:**

```
int x;

namespace XXX
{
  int x;
}

int main()
{
  int x = 1;       // Allowed
  ::x = 5;         // Global x, not local x
  XXX::x = 10;     // x from XXX, not local x

  std::cout << x << ::x << XXX::x << std::endl;    // Prints 1510
  return 0;
}
```

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 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.

**Exercise 1.7.2:**

```
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**[7].

**Exercise 1.7.3:**

```
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.

**Exercise 1.7.4:**

```
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.

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

**Exercise 1.7.5:**

```
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;
```

## 1.8. Funzioni

Definire una **funzione** significa specificare quali operazioni tale funzione deve compiere. Una funzione non puó essere chiamata fintanto che non é stata dichiarata. Una dichiarazione di funzione le fornisce un nome, un tipo di valore di ritorno ed un numero di argomenti ciascuno con il loro tipo. Una funzione che non ritorna nulla ha come tipo di ritorno `void`. Una definizione di una funzione le fornisce il corpo.
La definizione di una funzione deve essere coerente con le precedenti (se esistono) dichiarazioni. Inoltre, le dichiarazioni devono essere fra loro concordi. Tutte le dichiarazioni e definizioni della stessa funzione devono avere lo stesso tipo di valore di ritorno e lo stesso tipo degli argomenti; i nomi degli argomenti possono anche essere diversi, ma per rendere il codice piú chiaro é preferibile che siano uguali.

**Exercise 1.8.1:**

```
void f(int a, double b);
void f(int p, double q);     // Allowed, but weird
void f(int x, double y) {}   // Allowed, but weird
void f(int a, char b) {}     // NOT allowed, type mismatch
```

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.

Qualsiasi dato può essere passato ad una funzione in tutti e tre i modi, pertanto scegliere una modalità piuttosto che l'altra dipende sostanzialmente da cosa si vuole ottenere. L'unica eccezione sono gli array statici che, per mantenere C++ retrocompatibile con C, devono per forza venire passati tramite puntatore. Se un array viene passato come argomento di una funzione, viene in automatico passato un puntatore a tale array come effettivo argomento. In altre parole, viene fatta una conversione da `T[]` a `T*`.

**Exercise 1.8.2:**

```
int strlen(const char*);

void f()
{
  char v[] = "Hello, World!";
  int i = strlen(v);
}
```

Il passaggio per valore è da preferirsi quando si ha interesse a non modificare il dato di origine o quando si richiede esplicitamente di ricreare una copia del dato originale, e tale dato è ragionevolmente piccolo. In genere, il passaggio per valore viene effettuato per i tipi primitivi.
Il passaggio per puntatore e per referenza sono da preferirsi quando si ha interesse a modificare il dato originale oppure quando non si vuole che tale dato venga modificato, ma passare l'intero dato per valore richiederebbe troppa memoria. In questo secondo caso in particolare, è bene che il puntatore/referenza passato sia dichiarato `const`, di modo che possa essere letto ma non modificato.
Il passaggio tramite puntatore e tramite referenza sono molto simili. Il passaggio tramite puntatore è in genere meno "sicuro" di quello per referenza, perchè il puntatore in questione potrebbe riferirsi ad un dato che non esiste e creare inconsistenze. Pertanto, è preferibile sempre controllare che il puntatore passato ad una funzione non sia `nullptr`.

**Exercise 1.8.3:**

```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;
}
```

Una funzione non dichiarata `void` deve per forza ritornare un valore del tipo corrispondente a quello riportato nella sua dichiarazione. Una funzione `void` non puó ritornare alcun valore, ad eccezione di una chiamata ad un'altra funzione `void`. L'unica funzione che non necessita di un `return` esplicito é `main`, che se non lo possiede il compilatore vi aggiunge implicitamente `return 0`. Una funzione può ritornare dati di qualsiasi tipo, ad eccezione degli array, dei quali è possibile ritornare esclusivamente un puntatore che vi si riferisce, mai l'array in se.
É considerata bad practise ritornare da una funzione puntatori o reference a dati locali alla funzione stessa. Questo perchè i dati locali ad una funzione vengono rimossi dallo stack una volta che la funzione è stata eseguita, quindi tali puntatori/reference si riferiscono a dati non validi. In genere il compilatore é in grado
Due funzioni che si trovano nello stesso namespace ma che hanno dei parametri diversi (come numero e/o come tipo) sono comunque considerate funzioni diverse. In altre parole, C++ supporta l'**overloading**. Se viene chiamata una funzione che condivide il nome con un'altra funzione, il compilatore è in grado di dedurre (più o meno correttamente) quale sia la "versione" corretta della funzione da chiamare. Se esiste piú di un candidato per una chiamata di funzione ed il compilatore non é in grado di stabilire quale sia quella intesa, viene restituito un messaggio di errore. Naturalmente, due funzioni con la stessa identica firma non sono ammesse a prescindere.

**Exercise 1.8.4:**

```cpp
#include <iostream>

void f(int x, double y, char z) {}
void f(char x, int y, char z) {}
void f(float x, double y, int z) {}

int main()
{
  f(1, 0.0, 'a');       // First is called
  f('a', 0, 'b');       // Second is called
  f(1, 0, 'a');         // Ambiguous: all three valid
  f(1.0f, 0.0, 'a');    // Ambiguous: all three valid

  return 0;
}
```

Una funzione può anche avere dei valori di default opzionali assegnati ai parametri. Se una funzione con dei default sui parametri viene chiamata con un valore per tale parametro, allora tale parametro viene utilizzato normalmente, mentre se non viene specificato allora viene usato quello di default. I parametri di una funzione a cui viene assegnato un valore di default devono necessariamente essere contigui ed essere tutti sulla parte destra.

```cpp
ret_type func_name(type_1 par_1, ..., type_i par_i = def_i, ..., type_n par_n = def_n)
```

**Exercise 1.8.5:**

```cpp
#include <iostream>

void h(char c, int v = 20) {}        // Allowed
void f(char c = 10, int v = 90) {}   // Allowed
void g(char c = 10, int v) {}        // NOT allowed

int main()
{
  f(1, 29);                      // c = 1, v = 29
  f(1);                          // c = 1, v = 90
  f();                           // c = 10, v = 90

  return 0;
}
```

La keyword `inline` permette di dichiarare una funzione come funzione inline. Una funzione inline è una funzione che viene espansa in linea quando viene chiamata, ovvero l'intero codice della funzione inline viene inserito o sostituito nel punto della chiamata di funzione inline. Questa sostituzione viene eseguita dal compilatore C++ in fase di compilazione, non a runtime. Questo è particolarmente utile nel caso in cui il tempo necessario ad eseguire il cambio di contesto sia maggiore del tempo di esecuzione della funzione, e questo si verifica se la funzione è estremamente piccola, e permette così di migliorare le prestazioni.

## 1.9. Processo di compilazione

Raramente un programma scritto nel linguaggio C++ é costituito da un solo file sorgente che contiene l'intero codice. In genere, questo é costituito da uno o piú **file sorgente**, ciascuno contenente una parte del codice. Questo permette sia di suddividere logicamente il codice in piú componenti, enfatizzando quindi la sua struttura logica, sia di sfruttare la **compilazione separata**: quando una parte di codice viene modificata, é necessario ricompilare solamente il file che la contiene, non l'intero 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, a meno di richiederlo direttamente.
Le direttive sono riconoscibili perché sono precedute dal 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`).

**Exercise 1.9.1**:

```
#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
```

**Exercise 1.9.2:**

```
#include <iostream>

#define MAX(a, b) ((a) > (b) ? (a) : (b))

int main()
{
  std::cout << MAX(5, -2) << std::endl;
  return 0;
}
```

Il compilatore analizza sintatticamente ciascuna unitá di compilazione per verificare che non siano presenti errori di sintassi. 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 le fasi successive e devono pertanto venire contrassegnate da dei "placeholder". Il compilatore, aggiunge poi, se necessario delle informazioni di debug aggiuntive utili per la fase di testing.
Il compilatore converte ciascuna unitá di compilazione in un **file oggetto**. 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, perché potrebbero avere dei riferimenti a variabili o funzioni che sono state dichiarate ma non definite nell'unitá di compilazione stessa.
I file oggetto vengono unificati in un solo eseguibile dal **linker**. Questo cerca negli altri file oggetto le variabili e le funzioni che nel file in esame sono state dichiarate ma non definite, sostituendo il "placeholder" con l'indirizzo di memoria della variabile/funzione presa dal file oggetto dove é contenuta. Se in un file oggetto riportato il nome di una funzione/variabile che non ha peró una definizione in nessun altro file oggetto, viene restituito il messaggio di errore `Unresolved External Symbol`.

**Exercise 1.9.3:**

```
// file2.c
extern int x;
int f();
void g() { x = f(); }

// file1.c
int x = 1;
int f() { return 9; }
int main() { return 0; }
```

Il linker, oltre ad unificare i vari file oggetto in un solo eseguibile, introduce del codice di **startup** per renderlo riconoscibile dal sistema operativo come tale. Il linker si occupa inoltre di aggiungere (se necessario) le librerie esterne. Queste, tranne la libreria standard (che viene inclusa sempre in automatica) devono essere specificate manualmente.
La fase di linking termina con successo se ogni variabile/funzione é stata dichiarata in ogni file oggetto in cui compare, se (eventuali) dichiarazioni multiple sono fra loro consistenti, se ogni variabile/funzione usata é stata definita in uno dei file file oggetto che la utilizza e se tale definizione viene fatta esattamente una sola volta.

**Exercise 1.9.4:**

```c
// file2.c
int x = 0;
extern double b;
extern int c;

// file1.c
int x = 1;       // NOT allowed, x is already defined
int b = 1;       // NOT allowed, b was defined double
extern int c;    // Allowed, but pointless

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

Sebbene 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 di funzioni (il loro corpo) e variabili. 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 (mediante la keyword `extern`) e tipi di dato definiti dall'utente (classi e simili), namespace e costanti globali. 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 é stata dichiarata, 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 `.cxx` in un file `.cxx`, 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 uno stesso header file venga incluso in piú file `.cxx` facenti parte dello stesso programma, specialmente quando questo é molto grande. Nella maggior parte dei casi, questo comporta solamente che il preprocessore debba eseguire piú volte una stessa sostituzione della direttiva `#include`, che sebbene sia uno spreco di risorse non é di base un comportamento problematico. Esistono peró situazioni in cui includere piú volte un header puó effettivamente portare ad errori, ad esempio se un header contiene la definizione di una classe e viene incluso in piú file `.cxx` da linkare insieme, per il compilatore si sta cercando di definire la classe tante volte quanti file `.cxx` importano l'header.
Per prevenire situazioni di questo tipo si potrebbe organizzare il codice in modo da garantire che ogni inclusione dell'header file in ogni singolo file `.cxx` non comporti un conflitto. Non solo questo diventa molto difficile al crescere della dimensione del programma, ma in certi casi non é proprio possibile. Un metodo piú semplice prevede di dotare ogni header file di una **guardia**, che non é altro che una struttura del tipo:

```c
#ifndef SOMETHING_H
#define SOMETHING_H

// Content of the header goes here...

#endif
```

In questo modo, la prima volta che l'header file viene incluso in uno dei file `.cxx`, tale tag viene definito, mentre dalla seconda volta in poi tale tag giá esiste ed il contenuto dell'header file viene ignorato (perché é giá incluso).

In principio, ogni variabile globale viene inizializzata prima che venga chiamata la funzione `main()`. In una singola unitá di compilazione, queste vengono inizializzate nell'ordine in cui sono state dichiarate. Tuttavia, non c'é alcuna garanzia su quale sia l'ordine con cui le unitá di compilazione vengono scelte per inizializzarne le variabili globali. Per questo motivo, é preferibile evitare che i valori delle variabili globali di unitá di compilazione diverse dipendano fra loro.

# 2. Programmazione ad oggetti

## 2.1. Classi

Una **classe** viene dichiarata con la seguente sintassi:

```
class class_name {
  attribute1_type attribute1_name;
  attribute2_type attribute2_name;
  ...
  attributeN_type attributeN_name;

  method1_type method1_name;
  method2_type method2_name;
  ...
  methodN_type methodN_name;
};
```

Una classe é un tipo di dato definito dall'utente che é dotato anche di funzioni specifiche per poterlo manipolare. Di fatto, una classe é simile ad una `struct` con delle funzioni maggiormente accoppiate. Una istanziazione di una classe prende il nome di **oggetto**; come per una `struct`, l'operatore `.` permette di accedere alle funzioni e le variabili che contiene.
Le funzioni che appartengono ad una classe (**metodi**) e le variabili che appartengono ad una classe (**attributi**) possono essere dichiarate `public` oppure `private`. Un metodo/attributo `public` é accessibile dall'esterno se esiste un riferimento all'oggetto, mentre un metodo/attributo `private` é accessibile solamente da un metodo della classe.
Una sezione della classe puó venire dichiarata come `private` o come `public` riportando l'etichetta `private:` o `public:` rispettivamente. Tutto ció che si trova al di sotto di tale etichetta viene dichiarato come l'etichetta specifica. `public:` e `private:` possono essere ripetute anche più di una volta, ma per leggibilità è preferibile avere soltanto una ciascuna. Una sezione che non é esplicitamente dichiarata come `private` o `public` é assunta essere `private`.
Una sezione di una classe può anche essere dichiarata `protected`: tutto ciò che si trova in tale sezione è privato dal punto di vista dell'uso, ma può essere usato da una classe derivata da quella corrente.

**Exercise 2.1.1:**

```
class dbuffer {
  unsigned int size;
  int* buffer;
}
```

Una classe viene istanziata chiamando uno dei suoi **costruttori**. Un costruttore é un metodo di tale classe che accetta un certo numero di parametri (anche nessuno) e che istanzia la classe con tali valori. I costruttori sono metodi che hanno per nome il nome della classe stessa, ed in genere sono dichiarati `public`.

```
class_name::class_name(a1_type a1_value ..., aN_type aN_value) {
  a1_name = a1_value;
  ...;
  aN_name = aN_value;
}
```

**Exercise 2.1.2:**

```
dbuffer::dbuffer(unsigned int size, int value) {
  mBuffer = new int[size];
  mSize = size;

  for (unsigned int i = 0; i < size; i++) {
    mBuffer[i] = value;
  }
}
```

Essendo dei metodi, i costruttori supportano l'overloading, pertanto possono esserci piú costruttori per una stessa classe, fintato che la loro firma é distinta. Quale sia il costruttore corretto da richiamare viene dedotto dal compilatore sulla base del numero e del tipo di ciascun dato passato.

```
class_name class_instance(a1_value, ..., aN_value);
```

Naturalmente, come ogni tipo di dato, anche una istanza di una classe puó essere creata usando la memoria dinamica.

**Exercise 2.1.3:**

```
dbuffer* d = new dbuffer();
...
delete d;
d = nullptr;
```

Porre `const` alla fine di un metodo segnala al compilatore che tale metodo non modifica lo stato dell'oggetto su cui agisce. Riportare questo modificatore é talvolta necessario perché la modifica dello stato interno di un oggetto potrebbe avvenire anche senza utilizzare uno dei parametri passati al metodo (chiamando un costruttore, ad esempio). Una funzione generica non può terminare con `const`, soltanto una funzione di classe.
É considerata best practice inizializzare i valori manipolati da un costruttore anche se questi li sovrascrive. Questo perché, semmai l'istanziazione di uno degli attributi dovesse fallire, l'oggetto rimarrebbe comunque in uno stato coerente[8].
Puó essere utile, al fine di avere la certezza che un costruttore sia stato invocato correttamente, inserire delle informazioni di debug. Ovvero, se si sospetta che il costruttore non stia venendo chiamato correttamente, é possibile inserire del codice che viene eseguito ogni volta che il costruttore viene invocato e fintanto che si sta compilando in modalitá debug:

```
#ifndef NDEBUG
...
#endif
```

`NDEBUG` é una etichetta specificata nella libreria standard, che assume valore vero se il codice é stato compilato in modalitá debug; se il codice viene compilato in modalitá release, tale etichetta assume valore falso, e ció che racchiude `#ifndef` viene ignorato.
In un costruttore potrebbe esserci ambiguitá fra il nome passato per inizializzare un attributo della classe ed il nome dell'attributo stesso. La keyword `this`, un puntatore speciale alla classe stessa, permette di specificare che ci si sta riferendo ad un attributo o ad una funzione che appartiene alla classe, disambiguando da eventuali omonimi.

[8] Standard del C++ piú recenti permettono di chiamare un costruttore dentro un altro costruttore. Sarebbe quindi possibile chiamare il costruttore di default dentro tutti gli altri, inizializzando gli attributi con valori coerenti ma risparmiando linee di odice. É comunque considerata una bad practice

**Exercise 2.1.4:**

```
dbuffer::dbuffer(unsigned int size) {
  ...
  this->size = size;
}
```

Come da programmazione ad oggetti, anziché esporre direttamente i dati di una classe all'esterno, si preferisce filtrarne l'accesso mediante dei metodi **getter** e **setter**. In C++ esiste peró un modo migliore per implementare tale funzionalitá, ed é utilizzare una reference per avere un getter e setter in una sola funzione. Infatti, dato che si sta passando il dato per reference, se lo si modifica si sta effettivamente modificando il dato originale.

**Exercise 2.1.5:**

```
#include <cassert>

int& dbuffer::value(unsigned int index) {
  assert(index < mSize);

  return mBuffer[index];
}

dbuffer a, b;
a.value(i) = b.value(j);   // a.setValue(i, b.getValue(j))
```

In genere, i metodi getter/setter vengono "spezzati" in due: uno che agisce in sola lettura quando il dato passato è modificato con `const` ed uno che agisce in lettura e/o scrittura quando il dato è passato normalmente. Il metodo di sola lettura ha il modificatore `const` all'inizio e alla fine, ma per il resto è identico all'originale. Dato che il C++ supporta l'overloading delle funzioni, i due metodi possono anche avere lo stesso nome, è il compilatore a determinare in automatico quale delle due versioni usare sulla base della presenza del `const` nel dato passato.

**Exercise 2.1.6:**

```
#include <cassert>

int& dbuffer::value(unsigned int index) {
  assert(index < mSize);

  return mBuffer[index];
}

const int& dbuffer::value(unsigned int index) const {
  assert(index < mSize);

  return mBuffer[index];
}
```

Per inizializzare i valori di una classe in un costruttore è possibile, in alternativa al semplice assegnamento, usare una **initialization list**. Questa consiste in una lista in cui riportare dei copy constructor per ciascun attributo da voler inizializzare. Anche un tipo di dato primitivo può essere inizializzato in questo modo, perchè anche questi hanno associati dei copy constructor.

```
class_name::class_name(a1_type a1_value ..., aN_type aN_value) : a1_type(a1_initValue) ...,
aN_type(aN_initValue){
  ...
}
```

L'initialization list è anche l'unico modo per poter inizializzare in una classe un attributo dichiarato `const`, dato che per definizione se lo si tentasse di fare in un costruttore si avrebbe un errore in compilazione.

**Exercise 2.1.7:**

```
#include <iostream>

dbuffer::dbuffer() : mSize(0), mBuffer(nullptr){

  #ifndef NDEBUG
  std::cout << "dbuffer()" << std::endl;
  #endif
}
```

## 2.2. Best practice per la creazione di classi

In un buon codice C++ ci si aspetta che esistano almeno quattro metodi per ciascuna classe:

- Il **costruttore di default**;
- Il **distruttore**;
- Il **copy constructor**;
- L'**operatore assegnamento**.

Il costruttore di default é un costruttore che non prende alcun parametro ed inizializza ogni attributo della classe con dei valori di default (in genere `NULL`, `nullptr`, 0, ecc...).

```
class_name::class_name() {
a1_name = [0 / NULL / nullptr];
...;
aN_name = [0 / NULL / nullptr];
}
```

Vi sono situazioni dove quali siano i valori di default da utilizzare in un costruttore non é scontato, oppure potrebbero cambiare nel tempo. In tal caso, una possibile soluzione consiste nel dichiarare una o piú variabili `static` all'interno della classe che riportano tali valori. Una variabile dichiarata `static` all'interno di una classe funziona come una costante; tutti le istanze della classe possono riferirvisi e ne esiste esattamente una sola copia, comune a tutte le istanze.
Il costruttore di default é fondamentale perché se non é presente il compilatore é costretto a crearne uno di default al meglio delle sue capacitá. Il problema é che tale costruttore autogenerato richiama il costruttore di default (se esiste) per ogni attributo della classe che non sia un dato primitivo. Pertanto, se la classe contiene degli attributi di dato primitivo e non ha un costruttore di default definito esplicitamente, questi dati non sono istanziati, e questo comporta uno stato della memoria incoerente.
Un secondo motivo dell'importanza del costruttore di default sta nel fatto che, se si volesse istanziare un array di oggetti, tale costruttore viene chiamato in automatico per ciascun oggetto, ed é l'unico che é possibile chiamare per un array di oggetti[9].

[9]Questo non é piú vero negli standard C++ piú recenti.

**Exercise 2.2.1:**

```
#include <iostream>

dbuffer::dbuffer() {
  mSize = 0;
  mBuffer = nullptr;

  #ifndef NDEBUG
  std::cout << "dbuffer()" << std::endl;
  #endif
}
```

Il distruttore ha la stessa sintassi del costruttore, ma é preceduto dal carattere `~` ed ha `void` come argomento. Nel suo corpo, sono riportate le istruzioni per azzerare o eliminare dalla memoria tutti gli attributi dell'istanza della classe:

```
class_name::~class_name(void) {
static1_name = 0;
...;
staticN_name = 0;

delete[] dynamic1_name;
dynamic1_name = nullptr;
...;
delete[] dynamicN_name;
dynamicN_name = nullptr;
}
```

Quando termina lo scope in cui un oggetto esiste, il distruttore viene richiamato automaticamente, liberando tutta la memoria a questo associata senza doverlo fare manualmente.

**Exercise 2.2.2:**

```
dbuffer::~dbuffer(void) {
  if (mBuffer != nullptr) {
    delete[] mBuffer;
  }

  mBuffer = nullptr;
  mSize = 0;

  #ifndef NDEBUG
  std::cout << "~dbuffer(void)" << std::endl;
  #endif
}
```

L'operato del costruttore e del distruttore cosí definiti sono parte di un pattern di programmazione ad oggetti chiamato **RAII** (**Resource Acquisition Is Initialization**). Questo pattern specifica che l'allocazione (acquisizione) delle risorse deve avvenire durante la creazione dell'oggetto (nello specifico, durante l'inizializzazione) da parte del costruttore, mentre la deallocazione delle risorse deve avvenire durante la distruzione dell'oggetto da parte del distruttore. In altre parole, affinché l'inizializzazione possa avere successo, l'acquisizione delle risorse deve avere successo. In questo modo, la risorsa é garantito che sia in possesso dell'oggetto solamente fintanto che l'oggetto esiste: se l'oggetto non ha dei leak, allora nemmeno la risorsa puó averne.
Il copy constructor permette di istanziare una classe "clonando" una istanza giá esistente della stessa classe. Il copy constructor funziona anche se i dati della classe sono privati, perché il costruttore ha comunque accesso ai dati di una classe per la quale é costruttore.

```
class_name::class_name(const class_name &other) {
  a1_name = other.a1_name;
  ...
  aN_name = other.aN_name;
}
```

**Exercise 2.2.3:**

```
dbuffer::dbuffer(const dbuffer &other) {
  mSize = 0;
  mBuffer = nullptr;

  mBuffer = new int[other.mSize]
  for (unsigned int i = 0; i < other.mSize; i++) {
    mBuffer[i] = other.mBuffer[i];
  }

  mSize = other.mSize;

  #ifndef NDEBUG
  std::cout << "~dbuffer(const dbuffer&)" << std::endl;
  #endif
}
```

Il copy constructor ha anche una seconda utilitá. Se viene definita una funzione che fra i suoi argomenti ha una istanza di una classe passata per valore, é necessario che nello spazio di memoria della funzione venga creata una copia di tale istanza. Quando la funzione viene chiamata, questa chiama implicitamente il copy constructor per effettuare tale copia. Inoltre, se una classe viene istanziata come si inizializza una variabile, viene invocato in automatico il copy constructor (non puó pertanto considerare una inizializzazione).

**Exercise 2.2.4:**

```
void f(dbuffer dbx) { ... };

dbuffer db2 = db1;     // Same as dbuffer db2(db1)
f(db2);                // Calls dbuffer dbx(db2)
```

Come per il costruttore di default, se non viene definito il copy constructor il compilatore ne crea uno in automatico. Il problema é che il copy constructor autogenerato opera tale copia “membro a membro”, sia che i dati siano valoriali sia che siano puntatori, e non c’é garanzia che tale modo di copiare rifletta la logica della classe. In particolare, una copia di puntatori rende i due oggetti interdipendenti, dato che condividono uno stesso dato con i rispettivi puntatori, ed in genere questo non é un effetto voluto.
L’operatore assegnamento sostituisce il contenuto dell’istanza della classe su cui viene chiamato con quello dell’istanza della classe passata come argomento.

```
class_name& operator=(const class_name &other) {
  if (this != &other) {
  /* temporary copy */

  /* cleanup */

  /* substitution */
  }

  return *this;
  }
```

Il controllo `if (this != &other)` é una sezione standard dell’operatore di assegnamento. Questo controlla se l’indirizzo del puntatore corrisponde a quello dell’istanza passata come argomento. Tale controllo é necessario

perché se i due indirizzi sono effettivamente uguali, allora si sta cercando di assegnare un oggetto a sé stesso, che é una operazione "a vuoto", e quindi non c'é alcun bisogno di eseguirla (sarebbe uno spreco di risorse). Anche la chiusura `return *this` é standard, e di fatto ritorna l'oggetto stesso per reference. Sebbene possa sembrare astruso, tale `return` é utile per poter avere assegnamenti a catena.
Anziché copiare direttamente i campi dell'istanza `other` con quelli dell'istanza `this`, si preferisce costruire un oggetto temporaneo e poi, se tale costruzione ha avuto successo, sostituire i campi di questo oggetto temporaneo con l'istanza `this`. Questo perché prima di effettuare la copia da `other` a `this` é necessario svuotare `this` (essendo una istanza giá esistente, potrebbe non essere vuota), e se la copia dovesse fallire l'oggetto `this` si ritroverebbe svuotato. In questo modo, in caso di fallimento, l'oggetto `this` rimane nello stato precedente alla tentata copia. É infatti (quasi) garantito che una sostituzione non possa fallire, mentre non vi é garanzia sull'allocazione di memoria.

**Exercise 2.2.5:**

```
dbuffer& dbuffer::operator=(const dbuffer &other) {
  if (this != &other) {
    // temporary copy
    unsigned int tmpsize = other.mSize;
    int* tmpbuffer = new int[other.mSize];
    for (unsigned int i = 0; i < tmpsize; i++) {
      tmpbuffer[i] = other.mBuffer[i];
    }

    delete mBuffer;
    mBuffer = nullptr;
    mSize = 0;

    mBuffer = tmpbuffer;
    mSize = tmpsize;
  }

  return *this;
}
```

L'operatore di assegnamento puó essere scritto in maniera sostanzialmente automatica impiegando il copy constructor e la funzionalitá `std::swap` offerta dalla libreria standard, presente nell'header file `algorithm`. Questo é l'unico caso in cui in C++ é considerata best practice usare un costruttore all'interno di un altro costruttore.

```
#include <algorithm>

class_name& class_name::operator=(const class_name &other) {
  if (this != &other) {
    class_name tmp(other);
    std::swap(this->attribute1, tmp.attribute1);
    std::swap(this->attribute2, tmp.attribute2);
    ...
    std::swap(this->attributeN, tmp.attributeN);

    // tmp is removed automatically when out of scope
  }

  return *this;
}
```

**Exercise 2.2.6:**

```
#include <algorithm>

dbuffer& dbuffer::operator=(const dbuffer &other) {
  if (this != &other) {
    dbuffer tmp(other);
    std::swap(this->mSize, tmp.mSize);
    std::swap(this->mBuffer, tmp.mBuffer);
  }

  return *this;
}
```

Utile puó anche essere un metodo di serializzazione, che semplicemente stampi il valore dei campi della classe.

**Exercise 2.2.7:**

```
void dbuffer::print() {
  std::cout << mSize << std::endl;
  for (unsigned int i = 0; i < mSize; i++) {
    std::cout << mBuffer[i] << " ";
  }
  std::cout << std::endl;
}

dbuffer a;
a.print();
```

Questo può essere fatto in maniera più elegante ridefinendo l'operatore `<<`. Il vantaggio è che in questo modo l'oggetto non viene solamente restituito, se necessario, su standard output, ma può essere scritto su qualsiasi cosa che sia uno stream (un file, una socket, ecc...).

```
std::ostream &operator<<(std::ostream &os, const class_name &thing)
{
  ...
  return os;
}
```

Dato che tale operatore ha un significato globale, sia la definizione che la dichiarazione di tale ridefinizione deve venire fatta globalmente.

**Exercise 2.2.8:**

```
// Even better
std::ostream &operator<<(std::ostream &os, const dbuffer &db)
{
  os << db.getSize() << std::endl;
  for (unsigned int i = 0; i < db.getSize(); i++) {
    os << db[i] << " ";
  }
  os << std::endl;

  return os;
}

dbuffer a;
std::cout << a << std::endl;
```

La funzione `std::swap` opera in questo modo: se l'argomento passato è di un tipo primitivo, lo scambio viene effettuato normalmente scambiando gli indirizzi di memoria, mentre se è una classe utilizza (se esiste) il metodo `swap` interno alla classe. Può essere quindi utile, soprattutto se la classe è una classe container, aggiungervi un metodo `swap`.

**Exercise 2.2.9:**

```
#include <algorithm>

void dbuffer::swap (dbuffer &other) {
  std::swap(this->size, other.size);
  std::swap(this->buffer, other.buffer);
}
```

Ridefinire il metodo `swap` può essere utile anche per semplificare la ridefinizione dell'operatore di assegnamento.

**Exercise 2.2.10:**

```
#include <algorithm>

dbuffer& dbuffer::operator=(const dbuffer &other) {
  if (this != &other) {
    dbuffer tmp(other);
    tmp.swap(*this);
  }

  return *this;
}
```

I tipi di dato esposti dalla classe che non è certo se resteranno uguali nel tempo è bene che vengano mascherati da un `typedef`. Questo ha sia il vantaggio di nascondere i dettagli implementativi all'esterno, sia di poter mantenere uguale l'utilizzo della classe anche se il tipo del dato esposto dovesse cambiare (particolarmente utile per chi usa la classe). Naturalmente, dato che questo `typedef` è dentro alla classe, per accedere a quel tipo è necessario specificare il nome della classe mediante l'operatore `::`.

**Exercise 2.2.11:**

```
class dbuffer {
  public:
  typedef unsigned int size_type;

  private:
  size_type size;
  int* buffer;
}

const int& dbuffer::operator[](dbuffer::size_type index) const {
assert(index < mSize);

return mBuffer[index];
}
```

## 2.3. Ridefinizione degli operatori

C++ supporta la ridefinizione di quasi tutti gli operatori, cambiandone il significato per usa sola classe oppure globalmente. Ridefinire un operatore permette di modificarne la semantica per meglio adattarla ad una specifica situazione[10].
Per poter ridefinire un operatore é necessario riferirvisi tramite il suo nome. Il nome di un operatore é dato dalla parola chiave `operator` seguita dal simbolo dell'operatore. Questo perché un operatore é di fatto una funzione, e puó pertanto essere invocata come tale: semplicemente, l'uso del solo simbolo agisce come una abbreviazione. Un operatore può essere ridefinito sia globalmente, e che quindi avrá lo stesso significato a prescindere dall'argomento, che come membro di una classe, e che quindi avrá quel significato solamente se come argomento vi é un oggetto di tale classe. L'unico aspetto di un operatore che non é possibile modificare (oltre al suo simbolo) é il numero di argomenti; ad esempio, se un operatore é binario, non lo si puó ridefinire come unario. Inoltre, non è possibile "inventare" operatori da zero, ma solo ridefinire quelli forniti dalla grammatica del linguaggio.

**Exercise 2.3.1:**

```
// Redefine + so that it sums two objects value-by-value

dbuffer operator+(const dbuffer& lhs, const dbuffer& rhs)
{
  assert(lhs.size() == rhs.size());

  dbuffer result(lhs.size());

  for (dbuffer::size_type i = 0; i < lhs.size(); i++)
    result[i] = lhs[i] + rhs[i];

  return result;
}

dbuffer a(10, 3);
dbuffer b(10, 18);

dbuffer c = a + b;   // Allowed
```

[10]Tecnicamente, quello che si ottiene ridefinendo un operatore potrebbe anche essere ottenuto introducendo una funzione o un metodo speciale con la medesima semantica, ma la ridefinizione è in genere più comodo e più intellegibile

Gli unici operatori che non possono essere in alcun modo ridefiniti sono `.` e `::` [11]. Inoltre, gli operatori `[]`, `=`, `->` e `()` possono essere ridefiniti ma solamente rispetto ad una classe, non globalmente.
Un operatore binario puó essere ridefinito come un metodo (di classe) non statico avente un argomento oppure come una funzione (globale) avente due argomenti. Per un qualsiasi operatore binario `@`, la scrittura `aa@bb` puó essere interpretata nel primo senso come `aa.operator(@)(bb)` oppure nel secondo senso come `operator@(aa, bb)`. Nel primo caso, la presenza di un argomento solo é dovuta al fatto che `operator@` viene chiamato come metodo dell'oggetto `aa`, ed infatti `aa` é sottinteso con `this`.

**Exercise 2.3.2:**

```
// Redefine + so that it sums two objects value-by-value
// 'lhs' is just 'this'

dbuffer operator+(const dbuffer& rhs)
{
  assert(this->size == rhs.size);

  dbuffer result(this->size);

  for (size_type i = 0; i < this->size; i++)
    result[i] = *this[i] + rhs[i];

  return result;
}
```

Un operatore unario puó essere ridefinito come un metodo (di classe) non statico avente un argomento oppure come una funzione (globale) avente un argomento. Gli operatori unari esistono sia in forma prefissa che postfissa, pertanto ridefinire l'operatore in uno dei due modi non sottintende la ridefinizione nell'altro modo.
Per un qualsiasi operatore unario prefisso `@`, la scrittura `@aa` puó essere interpretata nel primo senso come `aa.operator(@)` oppure nel secondo senso come `operator@(aa)`. Nel primo caso, l'assenza di argomenti solo é dovuta al fatto che `operator@` viene chiamato come metodo dell'oggetto `aa`, ed infatti `aa` é sottinteso con `this`.
Per un qualsiasi operatore unario postfisso `@`, la scrittura `aa@` puó essere interpretata nel primo senso come `aa.operator(@)(int)` oppure nel secondo senso come `operator@(aa, int)`. Tale variabile `int` non ha nessun significato e non verrá mai effettivamente usata, ma é necessaria perché altrimenti la firma di un operatore unario prefisso e postfisso sarebbe identica ed il compilatore non avrebbe modo di distinguere fra le due.

[11] Questo perché hanno un nome come argomento, anziché un valore, e permettere una loro ridefinizione introdurrebbe un notevole livello di complessitá nella grammatica del linguaggio

**Exercise 2.3.3:**

```
// ++i
dbuffer& operator++(dbuffer& rhs)
{
  for (dbuffer::size_type i = 0; i < rhs.size(); i++)
    rhs[i] = rhs[i] + 1;

  return rhs;
}

// i++
dbuffer operator++(dbuffer& lhs, int)   // to distinguish between the two
{
  dbuffer tmp(lhs)

  for (dbuffer::size_type i = 0; i < lhs.size(); i++)
    lhs[i] = lhs[i] + 1;

  return tmp;
}
```

Essendo gli operatori di fatto delle funzioni, La ridefinizione degli operatori supporta l'overloading, pertanto è possibile ridefinire uno stesso operatore per più volte (a patto che la signature sia diversa). Il compilatore sceglierà in automatico di volta in volta quale "versione" dell'operatore usare in base al contesto.

**Exercise 2.3.4:**

```
// First version: two objects
dbuffer operator+(const dbuffer& lhs, const dbuffer& rhs)
{
  ...
}

// Second version: an object and an integer
dbuffer operator+(const dbuffer& lhs, int rhs)
{
  dbuffer result(lhs.size());

  for (dbuffer::size_type i = 0; i < lhs.size(); i++)
    result[i] = lhs[i] + rhs;

  return result;
}

dbuffer a(10, 3);
dbuffer b(10, 18);

dbuffer c = a + b;    // Uses the first version
dbuffer d = a + 25;   // Uses the second version
```

Ridefinire un operatore potrebbe non preservare le proprietá che aveva in origine. Ad esempio, un operatore binario `@` commutativo, se viene ridefinito per la classe di `aa` ma non per la classe di `bb`, rende `aa@bb` potenzialmente corretto ma `bb@aa` errata. Inoltre, ridefinire l'operatore `@` non ridefinisce in automatico `@=`, occorre farlo manualmente (se si desidera).

**Exercise 2.3.5:**

```
class complex {
private:
  double real_part;
  double imaginary_part;

public:
  complex operator+(complex a, complex b) {    // Complex plus complex
    complex r = a;
    r.real_part += b.real_part;
    r.imaginary_part += b.imaginary_part;
    return r;
  }

  complex& operator+=(complex a) {             // Immediate version
    this->real_part += a.real_part;
    this->imaginary_part += a.imaginary_part;
    return *this;
  }

  complex operator+(complex a, double b) {     // Complex plus real
    complex r = a;
    r.real_part += b;
    return r;
  }

  complex& operator+=(double a) {              // Immediate version
    this->real_part += a;
    return *this;
  }

  complex operator+(double a, complex b) {     // Real plus complex
    complex r = b;
    r.real_part += a;
    return r;
  }
}
```

Se si necessita di ridefinire un operatore di un'altra classe rispetto alla classe in esame, è strettamente necessario ridefinirlo globalmente, dato che l'alternativa sarebbe modificare la classe originale.
Usare un costruttore per effettuare le conversioni di tipo presenta due principali problemi. Il primo é che non é possibile convertire un oggetto di classe in un tipo primitivo (non essendo classi) ed il secondo é che non é possibile convertire un oggetto di una nuova classe in un oggetto di una classe definita in precedenza.
Questi problemi possono essere aggirati ridefinendo l'operatore di casting. Data una classe `X`, l'operatore `X::operatorT()` specifica come convertire un oggetto di classe `X` nel tipo (o nella classe) `T`. Nonostante non abbiano un valore di ritorno nella firma, tali ridefinizioni devono restituire comunque un valore. A prescindere da quale semantica venga fornita alla nuova versione dell'operatore, la sua precedenza rimane quella originale.

**Exercise 2.3.6:**

```
dbuffer::operator int() const {
  return this->size;
}

dbuffer x(19);
int i = static_cast<int>(x);   // Allowed, i = 19
```

Una funzione globale potrebbe necessitare di accedere ai dati di una funzione, anche se questi sono privati. Ad esempio, l'operatore `<<` potrebbe necessitare di accedere ai dati privati di una classe per poter scrivere su file attributi che questa non espone, di modo da poterla poi ricostruire in lettura.
Una soluzione é dichiarare una funzione con il modificatore `friend`, che permette al metodo cosí dichiarato di poter accedere agli attributi, anche se dichiarati privati, della classe in cui si trova la dichiarazione. É anche possibile dichiarare una intera classe come `friend`. In questo caso, tutti i metodi della classe dichiarata `friend` ottengono automaticamente questa proprietá.

**Exercise 2.3.7:**

```
friend std::ostream &operator<<(std::ostream &os, const dbuffer &db);
```

Una classe dichiarata `friend` non introduce un nuovo nome nel namespace, perché dichiarare una classe `friend` significa semplicemente dotare una classe che si suppone giá esistere di una proprietá aggiuntiva. Pertanto, una classe dichiarata `friend` lo diventa effettivamente se é giá stata dichiarata oppure venire dichiarata dopo ma nel medesimo namespace.

**Exercise 2.3.8:**

```
class X {/* ... */};        // X is friend of Y

namespace N {
  class Y {
    friend class X;
    friend class Y;
    friend class AE;
  };

  class Z {/* ... */};   // Z is friend of Y
}

class AE {/* ... */};      // AE is not friend of Y
```

Lo stesso discorso vale per le funzioni dichiarate `friend`. Una funzione dichiarata `friend` lo diventa effettivamente se é giá stata dichiarata, se viene dichiarata dopo ma nel medesimo namespace oppure se ha un oggetto della classe in cui é presente la dichiarazione `friend` come argomento.

**Exercise 2.3.9:**

```
void g() {/* ... */};          // g is friend of X

class X {
  friend void f();
  friend void g();
  friend void h(const X&);   // Has X as argument so ok
};

void f() {/* ... */};          // f is not friend of X
```

Aggiungere una funzione `friend` semplifica notevolmente il programma, ma introduce anche un accoppiamento forte tra due classi. In casi in cui l'accoppiamento fra due classi esiste comunque, un metodo `friend` é una buona scelta, ma spesso é possibile ottenere quello che fa un metodo `friend` usando dei metodi e delle conversioni di tipo.
Essendo un operatore una funzione, gli argomenti su cui agisce vengono passati come verrebbero passati ad una funzione. Per questo motivo, specialmente se l'operatore ridefinito agisce su oggetti di classe, é buona pratica

passargli gli argomenti per reference, perché é sostanzialmente garantito che un operatore modifichi una variabile piuttosto che ritornare un valore.
Di default, un costruttore avente un solo parametro viene inteso dal compilatore come una conversione di tipo implicita. A volte questo é effettivamente il comportamento desiderato, ma non sempre é cosí.

**Exercise 2.3.10:**

```
complex(z) = 2    // Initialize with complex(2)
string s = 'a'    // Make a string of length int('a')
```

Questa conversione puó venire impedita dichiarando un costruttore come `explicit`, di modo che tale costruttore venga chiamato solamente se effettivamente lo si desidera. Nello specifico, quando il copy constructor viene invocato, un costruttore `explicit` non verrá mai implicitamente invocato, anche a costo di generare un errore.

**Exercise 2.3.11:**

```
class String {
  ...
  explicit String(int n);   // Preallocate n bytes
}

String S1 = 10;               // NOT allowed
String S2(10);                // Allowed
String S2 = String(10);       // Allowed
```

Nel caso in cui la classe sia una *classe container* (se deve rappresentare un oggetto composito), per accedere ai suoi dati in maniera ancora migliore è possibile ridefinire l'operatore `[]`, di modo che l'accesso ricordi quello di un array.

**Exercise 2.3.12:**

```
// Even better
int& dbuffer::operator[](unsigned int index) {
  assert(index < mSize);

  return mBuffer[index];
}

const int& dbuffer::operator[](unsigned int index) const {
  assert(index < mSize);

  return mBuffer[index];
}

dbuffer a, b;
a[i] = b[j];   // a.setValue(i, b.getValue(j))
```

Una chiamata di funzione, ovvero una espressione del tipo `function(parameter-list)`, puó essere vista come l'applicazione di un operatore binario avente `function` come operando di sinistra e `parameter-list` come operando di destra. Per tale motivo, anche l'operatore `()` puó essere ridefinito.
L'applicazione piú interessante della ridefinizione di `()` é il poter fornire la usuale sintassi di chiamata di funzione ad oggetti che, in un certo senso, agiscono come una funzione. Oggetti con queste proprietá sono detti **oggetti funzione**, o semplicemente **funtori**. I funtori sono molto importanti perché permettono di passare una funzione come argomento di un'altra funzione.

**Exercise 2.3.13:**

```cpp
#include <iostream>

struct Comparison {
  bool operator()(int a, int b) const {
    return a != b;
  }
};

int main()
{
  Comparison P;
  bool c = P(5, 5);
  std::cout << c << std::endl;
  bool d = P(15, 5);
  std::cout << d << std::endl;
}
```

## 2.4. Eccezioni

Gli errori a runtime vengono gestiti dal linguaggio C++ adoperando il paradigma delle **eccezioni**. L'idea é che una funzione che incontra un errore a runtime che non é in grado di risolvere "lancia" (*throws*) una eccezione, delegando la gestione di tale errore al chiamante. Una funzione in grado di gestire gli errori in questo modo si dice che "cattura" (*catch*) una eccezione.
La gestione degli errori permette di separare due concetti distinti: l'*error reporting*, ovvero notificare al chiamante che c'é stato un problema, e l'*error handling*, ovvero gestire questa evenienza in qualche modo (anche non facendo nulla). Infatti, questi due aspetti sono sotto il controllo di due entitá diverse. Chi scrive una libreria potrebbe essere in grado di determinare dove possono verificarsi gli errori a runtime, ma in genere non ha modo di sapere come poterli gestire. Chi usa una libreria potrebbe essere in grado di poter gestire un errore a runtime, ma in genere non ha modo di sapere dove esattamente tale errore si sia verificato.
Le eccezioni figurano in C++ come **blocco try-catch**, avente la seguente sintassi:

```cpp
try {
  /* ... */
}
catch (/* ... */) {
  /* ... */
}
```

All'interno del blocco `try` viene eseguita la funzione che potrebbe restituire una eccezione. Tra le parentesi tonde della keyword `catch` viene riportata l'eccezione che quel blocco `catch`, anche chiamato **exception handler**, si incarica di gestire. Un blocco `catch` puó trovarsi solamente subito dopo un blocco `try`.
All'interno del blocco `catch` viene riportato cosa fare nel caso in cui si verifichi l'eccezione riportata tra parentesi tonde. Se l'eccezione non avviene, o se avviene una eccezione diversa da quella riportata fra parentesi, il blocco `catch` viene ignorato. Se una funzione puó generare piú eccezioni e le si vuole gestire, é possibile concatenare piú blocchi `catch` ad uno stesso blocco `try`, ciascuno preposto a gestire una eccezione fra queste. Blocchi `catch` possono a loro volta contenere blocchi `try`, ma nella maggioranza dei casi questa é una scelta implementativa da evitare.
Una eccezione viene effettivamente generata mediante `throw`:

```cpp
throw name_of_the_exception;
```

C++ non specifica un formato o un tipo di dato per costruire una eccezione. Queste possono essere di fatto qualsiasi cosa, che sia un tipo di dato custom o un tipo di dato predefinito. Per evitare confusione con i tipi standard, si preferisce utilizzare un tipo di dato custom specifico per ciascuna eccezione, in genere sotto forma di struct o di classe.

**Exercise 2.4.1:**

```cpp
#include <iostream>

struct Division_by_zero {};

int smart_ratio(int n, int d)
{
  if (d == 0) {
    throw Division_by_zero();
  } else {
    return n / d;
  }
}

int main()
{
  int a, b;

  std::cin >> a;
  std::cin >> b;

  try {
    int r = smart_ratio(a, b);
    std::cout << r << std::endl;
  } catch (Division_by_zero) {
    std::cerr << "Attempted division by zero!" << std::endl;
  }

  return 0;
}
```

Definire le eccezioni sotto forma di classi é particolarmente vantaggioso perché permette di costruire una gerarchia di eccezioni, alcune piú generiche ed altre piú specifiche. Ad esempio, le eccezioni della libreria standard sono definite come classi derivate della classe generica `std::exception`.
In una sequenza di `catch`, l'eccezione che viene catturata non é quella che prima si avvicina alla classe dell'eccezione lanciata, ma la prima compatibile. La dicitura `...` indica una qualsiasi eccezione, di qualsiasi tipo o semantica.

**Exercise 2.4.2:**

```cpp
void g() {
  try {
    //
  }
  catch (std::bad_alloc) {
    // Handle only memory allocation errors
  }
  catch (std::exception& e) {
    // Handle any stdlib exception
  }
  catch (...) {
    // Handle anything
  }
}
```

Nel caso in cui una funzione sia in grado di gestire solo parzialmente l'eccezione catturata, é possibile "rilanciare" (*rethrow*) l'eccezione e delegare ulteriormente la gestione dell'eccezione. Una eccezione viene rilanciata chiamando `throw` senza alcun operando.
Se si tenta di rilanciare una eccezione con `throw` ma non vi é alcuna eccezione da rilanciare, viene invocato il metodo `terminate()`. Allo stesso modo, se viene generata una eccezione e questa non viene catturata, viene invocato `terminate()`. Il metodo `terminate()` chiama a sua volta `abort()`, che segnala al sistema che il programma é terminato in maniera anomala.

## 2.5. Template e funtori

Una **funzione templata** é una funzione costruita in maniera speciale di modo che possa avere uno o piú argomenti aventi qualsiasi tipo. La sintassi per dichiarare una funzione templata é la seguente:

```
template <typename T1, ... typename Tn> return_type fun_name(t1 n1, ..., tN nN)
```

La keyword `typename` specifica uno o piú tipi *generici*, che verranno trattati come "segnaposto"; se uno degli argomenti viene dichiarato generico, il suo effettivo tipo sará scelto dal compilatore in base a come la funzione viene chiamata. Un tipo generico viene trattato come un tipo a tutti gli effetti, ed é pertanto possibile usarlo per dichiarare variabili interne alla funzione, operare dei cast ed usarlo per specificare degli argomenti.
Fra le parentesi angolate viene specificato il tipo che, per quella particolare chiamata, deve essere usato come tipo effettivo al posto del tipo generico.

```
fun_name <type1, ... typeN>(arg1 , ..., argN)
```

Naturalmente, se piú argomenti di una funzione templata sono stati dichiarati dello stesso tipo generico, una chiamata a tale funzione deve rispettare lo stesso tale vincolo di tipo.

**Exercise 2.5.1:**

```cpp
template <typename T> void swap(T& a, T& b) {
  T temporary = a;
  a = b;
  b = temporary;
}

int main() {
  int a = 1, b = 2;
  double c = 3, d = 4;

  swap <int>(a, b);       // Allowed
  swap <double>(c, d);    // Allowed
  swap <double>(c, a);    // NOT allowed
}
```

Quando il compilatore deve compilare una funzione templata, ne compila tante versioni quante sono le combinazioni di tipi di dato utilizzate per chiamare tale funzione. Ciascuna di queste "versioni" prende il nome di **specializzazione**. Si noti come le funzioni template non introducano un overhead, perché una specializzazione equale in termini di risorse usate alla sua controparte non templata.
Sebbene un tipo di dato generico possa essere passato ad una funzione in qualsiasi modo (per copia, per puntatore o per referenza), in genere é preferibile evitare di passare dati di tipo generico per copia. Questo perché non é possibile sapere in anticipo la dimensione del dato, ed un passaggio per valore potrebbe essere molto esoso in termini di memoria. Ove possibile, il passaggio per copia di tipi generici dovrebbero venire sostituiti da passaggi per referenza.

**Exercise 2.5.2:**

```
template <typename T> T example1(T a) {
  T b = a;
  ...
  return b;   // Correct, but ambiguous. Must use <>
}

template <typename T> void example2() {
  T temp;     // Correct, but ambiguous. Must use <>
  ...
}
```

La keyword `template` non fa parte della signature della funzione, ed é pertanto possibile fare overloading su una funzione templata, anche con una funzione non templata. Questo ha particolarmente senso nel caso in cui si vuole avere una versione "speciale" di una funzione templata che funzioni in un modo diverso per tipi di dato specifici. In questo caso, é particolarmente sensato definire comunque una funzione templata specifica, anche se non definisce alcun tipo generico, perché tale funzione viene effettivamente compilata solamente se necessario.
Un pattern molto comune prevede di utilizzare `typedef` per riferirsi ad una specializzazione che viene usata molto piú frequentemente delle altre (questo é il caso della classe `String` della libreria standard, ad esempio).
Una funzione templata non puó essere prima dichiarata e poi definita in un secondo momento, ma deve necessariamente venire dichiarata e definita insieme. Questo significa che non é possibile, come in genere viene fatto, riportare la firma della classe in un file `.h` e specificarne il corpo in un file `.cxx`. In genere, le funzioni template vengono riportate in file a parte con estensione `.hxx` ("a metá" fra un file header ed un file sorgente).
Si noti come una funzione templata introduca delle difficoltá di debugging che le funzioni normali non hanno. In particolare, il compilatore potrebbe non essere in grado di individuare degli errori nel codice della funzione templata fino a quando questa non viene specializzata con dei tipi "problematici".

**Exercise 2.5.3:**

```
#include <iostream>
#include <array>
#include <string>

template<typename T> custom_print(const T&)
{
  std::cout << T << std::endl;   // Allowed
}

int main()
{
  int x = 10;
  std::string s = "Hello, World!";
  std::array<int, 5> a = {1, 2, 3, 4, 5};

  custom_print<int>(x);                 // Ok
  custom_print<std::string>(s);         // Ok
  custom_print<std::array<int, 5>>(a);  // Error: std::array has no operator<<

  return 0;
}
```

Oltre alle funzioni, anche le classi possono essere definite template. Una classe templata é una classe in cui figura un tipo generico, come attributo e/o in uno o piú dei suoi metodi:

```
template<class C1, ... class Cn> class class_name {
  /* ... */
}
```

Istanziare una classe templata non é dissimile dall'istanziare una classe normale:

```
class_name<C1, ..., Cn> object(arg1, ..., argN);
```

Cosí come per le funzioni, una classe templata deve venire subito sia dichiarata e definita.

**Exercise 2.5.4:**

```
template<class C> class String {
private:
  Struct Srep;
  Srep* rep;
public:
  String();
  String(const C*);
  String(const String&);

  C read(int i) const;
  /* ... */
}
```

Nel chiamare una fuzione templata é possibile omettere le parentesi angolate solamente se il tipo generico non genera alcuna ambiguitá, ovvero quando il compilatore é in grado di dedurre dal contesto quale debba essere il tipo in questione. Se non esiste alcuna specializzazione della funzione templata che possa accordarsi con i tipi di dato passati, o se esistono piú specializzazioni egualmente plausibili, il compilatore restituisce un messaggio d'errore. D'altro canto, nell'istanziare una classe templata non é mai possibile omettere le parentesi angolate.
Le stesse regole della risoluzione fra tipi per le funzioni standard vengono adottate per la risoluzione fra tipi delle funzioni template. In sostanza, ogni chiamata ad una funzione templata dove le parentesi angolate sono state omesse viene risolta scegliendo l'insieme di tipi di dato che meglio si accorda con i valori passati come argomenti, dopodiché vengono applicate le classiche regole per risolvere le ambiguitá nell'overloading. Nello specifico:

- Vengono considerate tutte le possibili specializzazioni di tutte le funzioni templati i cui tipi si accordano con i tipi passati come argomento;
- Se piú funzioni template sono valide, ma alcune sono piú specializzate di altre, vengono scelte sempre quelle piú specializzate;
- Viene gestito l'overloading delle funzioni template e delle funzioni standard con tipi medesimi. Se il tipo di dato passato alla funzione templata é stato dedotto anziché riportato esplicitamente, non sono ammesse conversioni implicite;
- Se una funzione standard ed una funzione templata specializzata con gli stessi tipi sono candidate, viene sempre preferita la funzione standard;
- Se vi sono zero o piú di un candidato con la stessa probabilitá di essere la specializzazione scelta, viene restituito un messaggio di errore.

**Exercise 2.5.5:**

```
template <typename T> void test(const T& parameter) {}   // For any type
void test(const double& parameter) {}                    // For doubles

test(2.5);   // Will ALWAYS call the second version
```

Si noti come la deduzione fatta dal compilatore, per quanto certamente corretta sintatticamente, potrebbe non dare il risultato voluto.

Particolare attenzione va prestata ai costruttori delle classi template. Questo perché assegnare un tipo generico potrebbe essere sia un assegnamento bit-a-bit (come per i tipi primitivi) oppure richiedere memoria dinamica, e non é noto esplicitamente quale delle due situazioni si verifica. É bene pertanto gestire questa situazione con un blocco `try catch`.

**Exercise 2.5.6:**

```
dbuffer(size_type size, const value_type& value) : size(0), buffer(nullptr)
{
  buffer = new value_type[size];

  try {
    for (size_type i = 0; i < size; ++i) {
      buffer[i] = value;
    }
    this->size = size;
  } catch(...) {
    delete[] buffer;
    buffer = nullptr;
    throw;   // Very important!
  }
}
```

## 2.6. Iteratori

Gli **iteratori** forniscono una astrazione all'accesso degli elementi delle classi container. In questo modo, un algoritmo puó funzionare su container diversi senza doverlo modificare e senza dover conoscere i dettagli implementativi della classe. Allo stesso tempo, una classe container puó fare uso di iteratori per permettere un accesso ai suoi dati standardizzato.
Un **iteratore** é un oggetto che generalizza il concetto di puntatore ad un elemento di una sequenza. Non esiste in sé e per sé il tipo di dato "iteratore"; qualsiasi classe che implementi (almeno) determinate caratteristiche, riportate nell'interfaccia `std::iterator`, é di fatto un iteratore. Nello specifico, é definibile iteratore qualsiasi implementazione di `std::iterator` che possieda almeno queste nozioni:

- Una nozione di "l'elemento che sta venendo puntato";
- Una nozione di "elemento successivo a quello puntato";
- Una nozione di uguaglianza.

Con "sequenza" si intende qualsiasi struttura dati lungo la quale sia possibile spostarsi dal suo primo elemento al suo ultimo elemento mediante la nozione di elemento successivo. Ad esempio, gli array, le liste, gli alberi e i vettori sono considerabili sequenze, ed é pertanto possibile associarvi degli iteratori.
Un iteratore non é un puntatore templato, é invece una generalizzazione del concetto di puntatore, e che quindi potrebbe anche non manipolare direttamente gli indirizzi di memoria. Infatti, non esiste un "iteratore nullo": per testare se un iteratore sta effettivamente puntando a qualcosa oppure no, il metodo standard é di testare se stia puntando all'ultimo elemento della sequenza[12].
L'interfaccia `std::iterator` specifica cinque sottotipi di iteratori:

- **Forward iterator**, che permette di muoversi solamente in avanti di un elemento alla volta (é l'iteratore piú semplice possibile);
- **Bidirectional iterator**, che permette di muoversi sia in avanti che all'indietro di un elemento alla volta;
- **Random access iterator**, che permette di muoversi di un numero arbitrario di elementi sia in avanti che all'indietro (i puntatori sono di fatto dei random access iterator);
- **Input iterator**;
- **Output iterator**.

La classificazione viene fatta in base a quali operazioni tali iteratori sono in grado di supportare in maniera efficiente (in tempo lineare):

[12] Questo permette di semplificare diversi algoritmi, che altrimenti richiederebbero un caso speciale quando devono gestire la fine del loro input.

| Categoria | Output | Input | Forward | Bidirectional | Random Access |
|---|---|---|---|---|---|
| Read | | `*i`, `i->` | `*i`, `i->` | `*i`, `i->` | `*i`, `i->`, `[]` |
| Write | `*i`, `i->` | | `*i`, `i->` | `*i`, `i->` | `*i`, `i->`, `[]` |
| Iteration | `++` | `++` | `++` | `++`, `--` | `++`, `--`, `+`, `-`, `+=`, `-=` |
| Comparison | | `==`, `!=` | `==`, `!=` | `==`, `!=` | `==`, `!=`, `<`, `>`, `<=`, `>=` |

La scelta di quale iteratore utilizzare dipende dalla logica della classe. Non esiste un iteratore che sia il migliore per qualsiasi esigenza.
Un iteratore viene implementato come sottoclasse di una classe, che espone metodi per generarli e restituirli. Idealmente, l'iteratore restituito da una classe dovrebbe puntare al "primo" elemento della classe. Una classe deve fornire un iteratore speciale finale che indica che non esiste più alcun elemento oltre ad esso; questo non è un iteratore davvero accessibile.
Ogni iteratore deve ridefinire gli operatori `*` e `->` per permettere l'accesso ai suoi membri, l'operatore `++` per spostarsi da un elemento all'elemento successivo e l'operatore `==` per confrontare due iteratori e vedere se puntano allo stesso elemento. Inoltre, ogni operatore deve esistere in due versioni, una non-costante che permette di accedere ai suoi membri in lettura e scrittura e una costante che permette di accedere in sola lettura. Infine, la classe che implementa gli iteratori deve supportare due metodi, `begin()` e `end()`, che restituiscono iteratori rispettivamente all'inizio e alla fine della sequenza dei suoi elementi.
Si noti come un iteratore sia un oggetto a tutti gli effetti, ed è pertanto raccomandato implementare, oltre a tutto quanto citato sopra, anche tutti i metodi fondamentali di una classe (default constructor, copy constructor, destructor, ecc...). Tecnicamente questo esula dall'interfaccia standard, ma é comunque buona pratica farlo.
Per sapere come poter manipolare i dati puntati da un iteratore é necessario che questo metta a disposizione delle informazioni relative ai tipi di dato a cui si riferisce. L'interfaccia di `std::iterator` riporta tali informazioni secono il seguente formato:

```
template<class Iter> struct iterator_traits {
  typedef typename Iter::iterator_category iterator_category;
  typedef typename Iter::value_type value_type;
  typedef typename Iter::difference_type difference_type;
  typedef typename Iter::pointer pointer;
  typedef typename Iter::reference reference;
}
```

Ogni classe iteratore deve quindi implementare una sua versione di `iterator_traits`. Il significato dei campi é riportato di seguito:

- `iterator_category` indica il sottotipo di iteratore (forward iterator, random access iterator, ecc...);
- `value_type` indica il tipo di dato a cui l'iteratore punta (in genere, il tipo di dato per cui la classe container é costruita);
- `difference_type` indica il tipo di dato con cui esprimere la differenza fra due iteratori;
- `pointer` indica il tipo di dato ritornato dall'applicare `operator->` sull'iteratore;
- `reference` indica il tipo di dato ritornato dall'applicare `operator*` sull'iteratore.

Algoritmi differenti richiedono diversi tipi di iteratori come argomenti. Inoltre, uno stesso algoritmo puó venire implementato con diversi gradi di efficienza utilizzando iteratori diversi. Per questo motivo, la libreria standard organizza i sottotipi di iteratori in una gerarchia di classi. In questo modo, una funzione puó avere una forma per un iteratore generico ed una o piú forme per sottotipi specifici di iteratore. La gerarchia é la seguente:

```
  struct input_iterator_tag {};
  struct output_iterator_tag {};
  struct forward_iterator_tag : public input_iterator_tag {};
  struct bidirectional_iterator_tag : public forward_iterator_tag {};
  struct random_access_iterator_tag : public bidirectional_iterator_tag {};
```

Gli iteratori, per convenzione, devono essere sottoclassi chiamate `iterator` (lettura e scrittura) oppure `const_iterator` (sola lettura). A seconda delle necessitá implementative, é possibile avere solo `iterator`, solo `const_iterator` oppure entrambe. Le due classi devono essere distinte, ciascuna con i propri metodi, in genere praticamente identici:

```
Class Example {
  /* ... */
public:
  class iterator {
    /* ... */
  }
  class const_iterator {
    /* ... */
  }
}

int main()
{
  Example::iterator i;
  Example::const_iterator c;
}
```

Naturalmente, le classi iteratore devono essere riportati in una sezione pubblica della classe principale, altrimenti non sarebbe possibile richiederli.
La sezione privata di una classe iteratore é in genere costituita da un puntatore ad un elemento della classe principale e da un costruttore che permette di creare un iteratore a partire da tale puntatore. Tale costruttore dovrebbe essere privato, ma al contempo `friend` della classe principale, di modo che possano essere costruiti solamente passando dalla classe che contiene l'iteratore, che puó quindi filtrare l'accesso.
Nella maggior parte dei casi, un iteratore non alloca memoria dinamica, pertanto non é necessario scrivere esplicitamente il copy constructor, il destructor e l'operatore assegnamento. D'altra parte, avendo scritto un costruttore privato accessibile alla classe che lo contiene, la classe iteratore deve disporre di un default constructor scritto esplicitamente.
L'operatore `*` applicato ad un iteratore deve ritornare il valore contenuto nel nodo della classe container, non il nodo in sé. Questo perché gli iteratori hanno il preciso scopo di astrarre i dettagli implementativi della classe e poter accedere ai suoi valori in maniera naturale. L'operatore `*` non modifica l'iteratore in sé, pertanto puó essere ridefinito `const`. Naturalmente, l'operatore per `iterator` deve permette l'accesso con modifica, mentre l'operatore per `const_iterator` l'accesso in sola lettura.

```
value_type& operator*() const
{
  return p->item();
}
```

L'operatore `->` applicato ad un iteratore deve ritornare l'indirizzo di memoria a cui il dato si riferisce. Anche in questo caso, l'operatore `->` puó essere ridefinito `const`.

```
value_type* operator->() const
{
  return &**this;
}
```

L'operatore `++` permette all'iteratore di spostarsi lungo gli elementi della sequenza, astraendo il concetto di "prossimo elemento" dietro la stessa operazione. L'operatore `++` esiste in due forme: *pre* e *post*. Dato che l'unica differenza fra i due é il momento in cui avviene l'operazione (per prima o per ultima), é possibile definire l'uno in funzione dell'altro.

```
// pre-incremento
iterator& operator++()
{
  /* Raggiunto l'elemento successivo */
  return *this;
}
```

```
// post-incremento
iterator operator++(int dummy)
{
  iterator tmp = *this;
  ++(*this);
  return tmp;
}
```

É poi necessario dotare gli iteratori di un operatore `==` che valuti se due iteratori sono uguali e l'operatore `!=` che valuti se sono diversi. Questo permette, ad esempio, di capire se un iteratore si trova alla fine della sequenza, comparandolo con l'iteratore di fine. Si noti come sia necessario implementare sia una versione per quando i tipi

sono gli stessi (due `iterator` o due `const_iterator`) sia per quando non lo sono (un `iterator` confrontato con un `const_iterator` e viceversa).
Infine, é necessario introdurre un costruttore per la classe `const_iterator` che abbia un `iterator` come input, di modo che sia possibile convertire il secondo nel primo. Dato che si vuole che tale costruttore operi una conversione, occorre espressamente non dichiararlo `explicit`.
Per un forward iterator, gli operatori e i costruttori sopra citati sono sufficienti. Per un bidirectional iterator, é necessario ridefinire anche l'operatore `--` sulla falsa riga di quanto fatto per `++`. Per un random access iterator, oltre a `--` é necessario ridefinire anche `[]` per potersi spostare lungo la sequenza a partire da un indice, gli operatori di disuguaglianza per confrontare la posizione di iteratori diversi e lo "shifting" di $n$ posizioni, mediante `+`, `-`, `+=` e `-=`.

## 2.7. Ereditarietá

L'**ereditarietá** permette di estendere una classe a partire dai metodi e dagli attributi di un'altra classe. Un'alternativa é la **composizione**, inserire in una classe dei sotto-oggetti di supporto per fare operazioni specifiche.
Con la composizione, la classe mantiene una propria esistenza autonoma, nel caso dell'ereditarietá la classe ha una "doppia" identitá: é sia una istanza della sua classe, sia una istanza della classe da cui deriva.
Una classe viene ereditata mediante la keyword `public`:

```
class derived_class_name : public base_class_name
{
  /* ... */
}
```

Se una classe ne eredita un'altra, puó utilizzare tutti i metodi `public` o `protected` della classe ereditata ed accedere a tutti gli attributi `public` o `protected` della classe ereditata. Tutto ció che nella classe ereditata é dichiarato `private` non puó essere usato dalla classe che eredita.
Piú classi possono essere ereditate dalla stessa classe:

```
class derived_class_name : public base_class_name1, ..., public base_class_nameN,
{
  /* ... */
}
```

Ereditare piú classi in una sola volta puó portare a dei conflitti, che C++ non é in grado di individuare in fase di compilazione, ed é pertanto consigliato farlo con parsimonia.
Ereditando una classe mediante `public` é possibile usare i metodi `public` e `protected` della classe ereditata anche al di fuori della classe che la eredita. Una classe puó peró viene ereditata anche mediante la keyword `private`:

```
class derived_class_name : private base_class_name
{
  /* ... */
}
```

Anche in questo caso, la classe che eredita puó usare tutti i metodi non `private` della classe ereditata ed accedere a tutti gli attributi non `private` della classe ereditata. La differenza é che tali metodi e attributi non possono essere usati al di fuori della classe che eredita attraverso di essa, anche se nella classe ereditata non erano `private`.
Se la classe che eredita ridefinisce o effettua overloading su un metodo della classe ereditata, il metodo originale non é accessibile direttamente a partire dalla classe che eredita, anche nel caso in cui la firma sia diversa. Se si desidera usare il metodo originale, é necessario specificarlo mediante `::`.

**Exercise 2.7.1:**

```
class BaseClass
{
public:
  void f() {/* ... */}
  void f(int a, int b) {/* ... */}
};

class SubClass : public BaseClass
{
public:
  void f(int) {/* ... */}
};

int main()
{
  SubClass S;

  S.g(10);                // Allowed
  S.g();                  // NOT allowed, who is g()?
  S.g(1, 2);              // NOT allowed, who is g(int, int)?
  S.BaseClass::g(1, 2);   // Allowed
}
```

Nel costruire una classe derivata, viene seguita la stessa gerarchia delle classi. Ovvero, prima viene costruita la parte di classe relativa alla classe ereditata, poi vengono inizializzati i dati membro della classe che eredita ed infine viene eseguito il costruttore della classe che eredita.
Nello scrivere un costruttore per una classe derivata é necessario gestire anche la costruzione della parte di classe ereditata. Se si vuole che tale sezione venga costruita usando un costruttore specifico, é possibile farlo richiamando il suddetto costruttore nella initialization list del costruttore della classe che eredita. Se questo non viene fatto, viene automaticamente richiamato il costruttore di default della classe ereditata.
Se una classe derivata non ha definito esplicitamente un costruttore di default, il compilatore ne genera uno automatico che prima richiama il costruttore di default della classe ereditata, poi vengono richiamati i costruttori di default relativi ai dati membro della classe che eredita ed infine viene generato un costruttore di default vuoto per la classe che eredita.

**Exercise 2.7.2:**

```
class Member
{
public:
  Member() {/* ... */}
  Member(int x) {/* ... */}
};

class Base
{
public:
  Base() {/* ... */}
  Base(int x) {/* ... */}
};

class Derived : public Base
{
private:
  Member m;
public:
  Derived(int x) {/* ... */}
  Derived(int a, int b) : Base(100), m(0) {/* ... */}
};

int main()
{
  Derived d(1);      // 1.   Base::Base()
                     // 2.   Member::Member()
                     // 3.   Derived::Derived(int x)

  Derived d(1, 2);   // 1.   Base::Base(int x)
                     // 2.   Member::Member(int x)
                     // 3.   Derived::Derived(int a, int b)

  return 0;
}
```

I distruttori di una classe derivata vengono richiamati in ordine diverso rispetto alla gerarchia delle classi. Ovvero, prima viene richiamato il distruttore della classe che eredita, poi vengono richiamati i distruttori relativi ai dati membro della classe che eredita ed infine viene richiamato il distruttore della parte di classe relativa alla classe ereditata.
Se una classe derivata non ha definito esplicitamente un copy constructor, il compilatore ne genera uno automatico che prima richiama il copy constructor di default della classe ereditata, poi vengono richiamati i copy constructor dei dati membro della classe che eredita ed infine viene generato un copy constructor di default vuoto per la classe che eredita.
Nello scrivere esplicitamente un copy constructor per una classe derivata é necessario gestire anche la copia della parte di classe ereditata. Per fare questo, é necessario richiamare i relativi copy constructor nella initialization list del copy constructor della classe che eredita. A tali costruttori viene direttamente passato `other` (o un attributo di `other`), senza che questo generi un errore: tale operazione prende il nome di **upcasting**.

**Exercise 2.7.3:**

```
class Member
{
public:
  Member() {/* ... */}
  Member(const Member& other) {/* ... */}
};

class Base
{
public:
  Base() {/* ... */}
  Base(const Base& other) {/* ... */}
};

class Derived : public Base
{
private:
  Member m;
public:
  Derived(const Derived& other) : Base(other), m(other.m) {/* ... */}
};

int main()
{
  Derived d1;       // 1.   Base::Base()
                    // 2.   Member::Member()
                    // 3.   Derived::Derived()

  Derived d2(d1);   // 1.   Base::Base(const Base& other)
                    // 2.   Member::Member(const Member& other)
                    // 3.   Derived::Derived(const Derived& other)

  return 0;
}
```

Se una classe derivata non ha ridefinito esplicitamente l'operatore assegnamento, il compilatore lo ridefinisce sulla falsariga di come viene generato un copy constructor automatico. Se lo si volesse ridefinire esplicitamente, occorre copiare anche le sezioni di classe della classe ereditata.

**Exercise 2.7.4:**

```
class Member
{
public:
  Member& operator=(const Member& other) {/* ... */}
};

class Base
{
public:
  Base& operator=(const Base& other) {/* ... */}
};

class Derived : public Base
{
private:
  Member m;
public:
  Derived& operator=(const Derived& other)
  {
    if (this != &other) {
      Base::operator=(other);
      this->m = other.m;
    }

    return *this;
  }
};
```

L'upcasting permette di risalire una gerarchia di classi, usando una classe derivata come fosse la classe base. É possibile avere piú classi che derivano dalla stessa classe eppure avere funzioni che operano sulla classe base passando oggetti della classe derivata.

**Exercise 2.7.5:**

```
class Vehicle {/* ... */};

class Car : public Vehicle {/* ... */};
class Motorbike : public Vehicle {/* ... */};

void register(Vehicle& v) {/* ... */}

int main()
{
  Car c();
  Motorbike m();

  register(c);    // Allowed
  register(m);    // Allowed
}
```

L'upcasting questo puó essere fatto solamente tramite puntatori o reference, non per valore.

```
DerivedClass d;
BaseClass* b = &d;
```

```
DerivedClass d;
BaseClass& b = d;
```

Quando un oggetto viene upcastato alla classe ereditata i suoi attributi ed i suoi metodi non presenti in questa esistono ancora, ma sono "oscurati". Quando si cerca di fare un upcasting per valore ció che accede davvero é

uno **slicing**, ovvero dove gli attributi ed i metodi propri della classe che eredita non sono oscurati, bensí sono eliminati del tutto.

## 2.8. Polimorfismo

Una classe si dice **polimorfa** se puó essere usata su una classe base mantenendo delle caratteristiche della classe derivata. A differenza di altri linguaggi di programmazione ad oggetti (es. Java), una classe C++ non é polimorfa di default, ma solo se si specifica che debba esserlo. Questo perché introdurre il polimorfismo richiede risorse aggiuntive da mettere a disposizione a runtime.
Per rendere polimorfo un metodo é possibile utilizzare la keyword `virtual`:

```
virtual ret_value polymorph_function(arg1, ..., argN)
```

Una metodo dichiarato `virtual` é automaticamente `virtual` anche per tutte le classi derivate da quella che lo contiene. Una classe polimorfa é quindi una classe che contiene almeno un metodo polimorfo (definito con `virtual`).
Un metodo dichiarato `virtual` puó essere ridefinito come non piú `virtual` (omettendo la keyword) in una classe derivata. Questa particolare forma di overloading é detta **overriding**.

**Exercise 2.8.1:**

```
class Instrument
{
public:
  Instrument(void) {/* ... */}
  virtual void play() {/* ... */}
}

class Guitar : public Instrument
{
public:
  Guitar(void) {/* ... */}
  void play() {/* ... */}   // Override
}

class Drums : public Instrument
{
public:
  Drums(void) {/* ... */}
  void play() {/* ... */}   // Override
}

void play_instrument(Instrument& b)   // Has Instrument as input
{
  b.play();
}

int main()
{
  Guitar g;
  play_instrument(g);   // Calls Guitar::play(), not Instrument::play()

  Drums d;
  play_instrument(d);   // Calls Drums::play(), not Instrument::play()

  return 0;
}
```

É possibile forzare un metodo `virtual` ad essere “puramente” `virtual`, senza implementazione, dichiarandolo uguale a 0:

```
virtual ret_value polymorph_function(arg1, ..., argN) = 0;
```

Una classe che contiene uno o piú metodi "puramente" `virtual` viene detta **classe astratta**. Tale classe non puó essere istanziata, puó solamente essere ereditata da un'altra classe che ne implementi (del tutto o in parte) l'interfaccia. Se la classe che eredita una classe astratta fa un override di tutti i metodi `virtual` "puri" che eredita, allora diviene effettivamente istanziabile; se invece fa override soltanto di una parte dei metodi `virtual` puri che eredita, é a sua volta considerata classe astratta.

**Exercise 2.8.2:**

```
class Instrument
{
public:
  Instrument(void) {/* ... */}
  virtual void play() = 0;   // Pure virtual; Instrument is abstract
}

class Guitar : public Instrument
{
public:
  Guitar(void) {/* ... */}
  void play() {/* ... */}   // Override, can now be instantiated
}

class Drums : public Instrument
{
public:
  Drums(void) {/* ... */}
  void play() {/* ... */}   // Override, can now be instantiated
}

void play_instrument(Instrument& b)   // Has Instrument as input
{
  b.play();
}

int main()
{
  Instrument i;         // NOT allowed

  Guitar g;
  play_instrument(g);   // Calls Guitar::play(), not Instrument::play()

  Drums d;
  play_instrument(d);   // Calls Drums::play(), not Instrument::play()

  return 0;
}
```

Oltre all'upcasting é anche possibile compiere **downcasting**, ovvero passare da una classe piú generica ad una classe piú specifica. A differenza dell'upcasting, che é una operazione sempre sicura, il downcasting é quasi sempre insicuro, perché le informazioni che la classe che vuole fare il downcasting ha rispetto alla classe bersaglio sono in genere parziali o assenti. Cosí come per l'upcasting, il downcasting é possibile solo rispetto a puntatori o reference.
L'unico caso in cui il downcasting é effettivamente sicuro si ha quando la classe in questione é polimorfa. In particolare, il downcasting per classi polimorfe viene eseguito mediante `dynamic_cast`:

```
BaseClass b;
DerivedClass& d = dynamic_cast<DerivedClass&>(b);
```

Il downcasting mediante `dynamic_cast` é considerato sicuro perché, in caso di fallimento dell'operazione, viene restituito un errore a runtime. In particolare, se il downcasting é stato fatto su un puntatore, viene restituito il

puntatore nullo, mentre se é stato fatto su una reference viene lanciata l'eccezione apposita `std::bad_cast()`. Cercando di usare `dynamic_cast` su una classe non polimorfa o su qualcosa che non sia una classe comporta un errore di compilazione.

# 3. Appendice

## 3.1. Doxygen

**Doxygen** é uno strumento che permette di generare documentazione del codice C++ in maniera automatica a partire da dei commenti propriamente formattati. Tali commenti é best practise riportarli nei file header in cui sono riportate le firme dei metodi.
Doxygen riconosce un commento speciale tipo perché é un commento multilinea che inizia con un doppio asterisco:

```
/**

Comment to be generated goes here...

*/
```

Doxygen formatta la documentazione per mezzo di *direttive*, parole chiave che iniziano con `@`. Una prima direttiva che é bene porre é `@brief`, con cui viene specificata una descrizione sommaria (lunga una sola riga) della funzione e del suo significato. Un commento piú descrittivo puó essere riportato lasciando una riga vuota.

```
/**
  @brief brief description goes here...

  longer description goes here...
*/
```

La direttiva `@param` permette di descrivere un parametro di una funzione; una direttiva `@param` va riportata per ogni parametro della funzione. La direttiva `@return` descrive il valore di ritorno della funzione.

```
/**
  @param first parameter is this... and does that...
  @param second parameter is this... and does that...

  @param nth parameter is this... and does that...

  @return return value is this... and does that...
*/
```

**Exercise 3.1.1:**

```
/**
  @brief Computes the Euclidean distance

  Computes the Euclidean distance between two points

  @param x1 the x-coordinate of the first point
  @param x2 the x-coordinate of the second point
  @param y1 the y-coordinate of the first point
  @param y2 the y-coordinate of the second point

  @return the Euclidean distance
*/

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

La direttiva `@pre` permette di descrivere una *precondizione* necessaria affinchè un metodo o una classe funzioni correttamente. Ha particolare valore per documentare il senso degli `assert`. La direttiva `@post` permette di descrivere una *postcondizione*.

**Exercise 3.1.2:**

```
/**
  @brief Computes the Pearson correlation coefficient

  @pre sigmaX != 0
  @pre sigmaY != 0
*/

  float pearson_coefficient(float covariance, float sigmaX, float sigmaY)
  {
    assert(sigmaX != 0);
    assert(sigmaY != 0);

    ...
  }
```

## 3.2. Make e Makefile

La compilazione avviene invocando un compilatore per il linguaggio C++, come `gcc` o `clang`. Per generare un file oggetto `file.o` a partire da un file sorgente `file.cxx` è necessario invocare il compilatore con il flag `-c`:

```
$ gcc -c file.cxx -o file.o
```

Il flag `-o`, opzionale, serve ad esplicitare il nome che avrà il file oggetto generato. Se non riportata, come nome viene scelto automaticamente il nome del file originale con l'estensione cambiata in `.o`.
Per linkare i file oggetto `file1.o`, `file2.o` e `file3.o` in un file eseguibile non è necessario alcun flag obbligatorio:

```
$ gcc file1.o file2.o file3.o -o file.exe
```

Se il flag `-o` non viene riportato, come nome di default un file eseguibile avrà sempre `a.out`.
I file header non vanno compilati, pertanto non è necessario notificare il compilatore della presenza di tali file. Tuttavia, è possibile notificare al compilatore di considerare dei percorsi extra nella ricerca degli header file con il flag `-I`. Mediante `:` è possibile includere più di un percorso:

```
$ gcc -I/extra/path1:/extra/path2:/extra/path3 -c file.cxx -o file.o
```

Un progetto complesso è in genere costituito da più cartelle e sottocartelle, di modo da organizzare i file in maniera più chiara. Spesso, i file sorgente si trovano in una cartella chiamata `src` ed i file header in una cartella chiamata `include`. L'unico file che si trova nella cartella principale è `main.cxx`.

```
.
|--- main.cxx
|--- src
|    |--- file1.cxx
|    |--- file2.cxx
|    |--- ...
|--- include
     |--- file1.h
     |--- file2.h
     |--- ...
```

**Exercise 3.2.1**: Si consideri il seguente progetto C++:

```
.
|--- main.cxx
|--- src
|    |--- Point.cxx
|    |--- Rectangle.cxx
|    |--- Circle.cxx
|--- include
     |--- Point.h
     |--- Rectangle.h
     |--- Circle.h
```

Potrebbe venire compilato in questo modo:

```
$ gcc -c -Iinclude main.cxx -o main.o
$ gcc -c -Iinclude src/Point.cxx -o Point.o
$ gcc -c -Iinclude src/Rectangle.cxx -o Rectangle.o
$ gcc -c -Iinclude src/Circle.cxx -o Circle.o
$ gcc main.o Point.o Rectangle.o Circle.o -o main
```

Modificare un file di un progetto potrebbe influire o non influire su altri file, e di conseguenza potrebbe essere o non essere necessario ricompilarli. Per avere una migliore idea del modo in cui i file dipendono gli uni dagli altri è possibile rappresentare questa interdipendenza mediante un **grafo delle dipendenze**. Tale grafo è composto da tre sottografi:

- **Dipendenze di inclusione**. Rappresentano la relazione che intercorre tra i file che vengono inclusi l'uno nell'altro, in genere header files. Se un file `A.cxx` (o `A.h`) necessita di importare un file `B.h` (o `B.cxx`), allora esiste un arco che unisce `A.cxx` (o `A.h`) e `B.h` (o `B.cxx`);
- **Dipendenze di compilazione**. Rappresentano la relazione fra i file oggetto ed i file sorgente necessari alla loro compilazione. Se un file oggetto `A.o` necessita di un file sorgente `B.cxx` per poter essere compilato, allora esiste un arco che unisce `A.o` e `B.cxx`;
- **Dipendenze di linking**. Rappresentano la relazione che intercorre tra un eseguibile ed i file oggetto necessari alla sua costruzione. Se un eseguibile `A` necessita di un file oggetto `B.o`, allora esiste un arco che unisce `A` e `B.o`.

Sfruttare il grafo delle dipendenze permette di evitare ricompilazioni inutili, ricompilando quando necessario solamente i file che è effettivamente necessario ricompilare. Tuttavia, è possibile evitare di dover controllare manualmente il grafo mediante il programma `make`.
`make` è uno strumento in grado di determinare in maniera del tutto automatica quando un file di un progetto C++[13] necessita di essere ricompilato e quali sono i file che, a sua volta, devono essere ricompilati per introdurre il cambiamento. `make` fa uso di un file speciale, chiamato `Makefile`, che contiene una rappresentazione testuale del grafo delle dipendenze del progetto.
Un `Makefile` è costituito da una serie di **target**. Ogni target è riportato su una riga, seguita dal carattere `:` e da uno o più file separati da spazi. Il target rappresenta un nodo del grafo delle dipendenze, mentre i file alla destra di `:` rappresentano i nodi con il quale tale nodo ha un arco.

```
target: file1 file2 ... fileN
```

Al di sotto di ciascun target ed indentato di un tab sono riportate tutte le azioni che è necessario compiere per compilare tale target.

```
target: file1 file2 ... fileN
<TAB> command1
<TAB> command2
<TAB> ...
<TAB> commandN
```

Il comando `make`, se eseguito senza parametri, cerca di eseguire le azioni specifiche per il target che si trova più in alto nel file che si chiama `Makefile`.

[13] `make` può essere utilizzato per gestire qualsiasi progetto che faccia uso di un linguaggio compilato, come `C` o `LaTeX`.

```
$ make
```

Per indicare a `make` di scegliere un target specifico, è sufficiente riportarlo di seguito a `make`. Più target possono essere riportati; in tal caso, verranno eseguite le relative azioni ordinatamente.

```
$ make target1 target2 ... targetN
```

Le azioni relative ad un certo target vengono eseguite solamente se uno o più file fra quelli riportati a destra di `:` sono stati modificati. Se uno fra questi file è un target a sua volta, anche tale target viene gestito. In questo modo, la risoluzione delle interdipendenze fra i target viene gestita in maniera ricorsiva.
Per determinare se è necessario eseguire le azioni relative ad un target, `make` ne controlla il *timestamp*: se almeno uno dei file che si trovano alla destra di `:` è stato modificato per l'ultima volta dopo l'ultima modifica del target, allora `make` esegue le azioni relative al target. Si noti come le azioni relative al target sono eseguite a prescindere dall'entità della modifica: anche se un file a destra di `:` viene modificato lasciando intatta la semantica (ad esempio, aggiungendo un commento), le azioni sono eseguite comunque.
Se il `Makefile` non ha per nome "`Makefile`", occorre specificarne il nome invocando `make` con il flag `-f`.

```
$ make -f makefile_filename target1 target2 ... targetN
```

Di norma, appena un target fallisce per un qualsiasi motivo, `make` fallisce. Specificando il flag `-k` è possibile spingere `make` a non fermarsi al primo errore ma proseguire comunque.
Oltre ai target, in un `Makefile` è possibile definire delle variabili che verranno poi valutate quando tale viene processato da `make`. Le variabili di un `Makefile` non hanno tipo, e vengono dichiarate con la medesima sintassi del C++. Per convenzione, sono in maiuscolo:

```
VARIABLE = values
```

Per riferirsi ad una variabile all'interno del `Makefile` è sufficiente racchiuderla fra parentesi tonde e accodare `$` in testa:

```
$(VARIABLE)
```

Nello specifico caso dei `Makefile` per il C++, variabili utili da definire sono le seguenti:

- `CXX`, che indica quale compilatore C++ da utilizzare;
- `CXXFLAGS`, che indica quali flag è necessario passare al compilatore;
- `LDFLAGS`, che indica quali flag è necessario passare al linker.

**Exercise 3.2.2:**

```
CXX = g++
CXXFLAGS = -Wall -g -Iinclude
LDFLAGS = -Wl

main.exe: Point.o Rectangle.o main.o
  $(CXX) $(LDFLAGS) -o main.exe Point.o Rectangle.o main.o

main.o: main.cxx
  $(CXX) $(CXXFLAGS) -c main.cxx -o main.o

Rectangle.o: Rectangle.cxx
  $(CXX) $(CXXFLAGS) -c src/Rectangle.cxx -o Rectangle.o

Circle.o: Circle.cxx
  $(CXX) $(CXXFLAGS) -c src/Circle.cxx -o Circle.o

Point.o: Point.cxx
  $(CXX) $(CXXFLAGS) -c src/Point.cxx -o Point.o
```

Oltre alle variabili definite dall'utente, è possibile utilizzare delle variabili che `make` interpreta in maniera automatica. Le più importanti sono le seguenti:

- `$@`, che indica il target corrente;
- `$%`, che indica il target corrente, membro di un archivio;
- `$^`, che indica tutte le dipendenze del target corrente, separate da spazi;

- `$<`, che indica la prima dipendenza del target corrente (quella più a sinistra);
- `$?`, che indica le dipendenze del target corrente che sono state aggiornate più di recente;
- `$+`, che indica tutte le dipendenze del target corrente, con i duplicati in ordine.

Talvolta potrebbe essere necessario avere dei target che non sono associati ad alcun file reale. Target di questo tipo possono essere indicati mediante `.PHONY`:

```
.PHONY: target_name
target_name:
<TAB> command1
<TAB> command2
<TAB> ...
<TAB> commandN
```

Uno degli utilizzi più comuni di `.PHONY` è quello di avere un target che permetta di eliminare tutti i file oggetto, di modo da essere assolutamente certi di ripartire dalla situazione iniziale. Tale target è in genere chiamato `clean`:

```
.PHONY: clean
clean:
  rm -rf *.o *.exe
```