Indice

1. Introduzione

1.1. 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, ma quale sia il suo valore non é da cercarsi nel file attuale (in genere questo viene fatto per la definizione di costanti globali in sostituzione a `#define`).
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 é indeterminato, perché dipende dal contenuto della memoria che prima occupava tale variabile.

```
// 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.

```
// 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 che differiscono per il tipo ma hanno lo stesso nome sono ammesse.

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

float y;
char y;                 // 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[1], 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.

```
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.

[1]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.

```
// 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.

1.2. 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**;
- **booleani**;
- **caratteri**;
- **numeri decimali**;
- **enumerativi**;
- `void`.

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.

```
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.

```
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`).

```
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`.

```
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[2].

```
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
```

1.3. Array

Un puntatore é un tipo di dato che contiene un riferimento ad un indirizzo di memoria. Esistono tanti tipi di puntatori quanti sono i tipi primitivi; per quanto contengano un indirizzo di memoria (che é un numero), il loro tipo non é un intero, bensí é specificatamente di tipo "puntatore a...". Puntatori a tipi diversi sono incompatibili. Nonostante questo, la dimensione di un puntatore non dipende dal tipo di dato dato che gli indirizzi di memoria hanno tutti la stessa dimensione. Il valore di default per un puntatore é `NULL` oppure (standard C++11) `nullptr`. Un puntatore viene dichiarato come una normale variabile di tale tipo, ma postponendo `*` al tipo[3]. Anteponendo `&` al nome di una variabile se ne ottiene l'indirizzo di memoria. Per ricavare il valore della cella di memoria a cui un puntatore é associato si antepone `*` al nome della variabile. L'atto di "risalire" al valore a cui un puntatore é legato é una operazione che prende il nome di **dereferenziazione**.

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

[2]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.

[3]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.

```cpp
int* p = nullptr;                   // initialises a pointer p

int s = 10;                         // initialises an integer s

p = &s;                             // p points to the memory address of s

std::cout << *p << std::endl;       // retrieves the value to which p is
                                    // pointing, and prints it

(*p)++;                             // retrieves the value to which p is
                                    // pointing, and increments it by one
```

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

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

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

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

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

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

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

```
int i;
double d;

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

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

Le utilitá dei puntatori sono riassunte di seguito:

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

Un **array** é una sequenza di dati dello stesso tipo, memorizzati in aree di memoria contigue chiamate **celle**, utile per memorizzare dati che fra loro hanno un qualche legame logico. Un array viene dichiarato come una normale variabile di un certo tipo, ma accodando `[]` al nome. La dimensione di un array é fissata, direttamente nel codice riportandola fra le parentesi quadre oppure venendo "dedotta" dal compilatore in base a come l'array viene inizializzato.
Un array viene inizializzato riportando fra parentesi graffe i valori, separati da virgole, che verranno assegnati ordinatamente a ciascuna posizione dell'array. Se vi sono piú valori che posizioni nell'array, il compilatore restituisce un errore. Non é possibile cambiare i valori assegnati alle celle di un array usando la medesima sintassi dell'inizializzazione, ma occorre farlo cella per cella. Riportando il nome dell'array con N fra le parentesi graffe si ottiene il valore nella (N - 1)-esima cella dell'array; gli array partono da 0. Il compilatore non restituisce un messaggio di errore se si cerca di accedere ad una cella di memoria che supera le dimensioni dell'array.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
reference_type& reference_name = variable_to_be_referenced
```

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

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

1.4. Struct, enum, typedef

Similmente agli array, che sono tipi primitivi, le **struct** sono considerate tipi compositi. Di fatto, una struct é un "raggruppamento" di dati anche di tipo diverso, chiamati **campi**. Sono di fatto una forma piú "rudimentale" del concetto di classe. Tutti i dati di una struct sono di default pubblici, quindi liberamente modificabili.

Una struct puó essere inizializzata allo stesso modo di come viene inizializzato un array, dove ogni elemento i-esimo all'interno delle parentesi graffe viene assegnato alla i-esima variabile contenuta nella `struct`. Una `struct` puó anche essere inizializzata parzialmente, ovvero assegnando un valore solamente ai primi n campi.

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

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

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

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

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

```
struct_name.field
```

```
pointer_to_a_struct->field
```

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

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

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

La memoria occupata da una `struct` dipende dalla politica di allocazione della memoria del compilatore. In genere, viene prediletta una allocazione di memoria che ottimizza l'accesso piuttosto che la dimensione. Per tale motivo, per ottenere la massima efficienza in termini di spazio occupato é preferibile disporre i dati all'interno in ordine decrescente di grandezza, di modo che piú dati possano venire "accorpati" in un'unica `word`.
`typedef` permette di associare un alias ad un tipo di dato giá esistente. É utile per riferirsi ad un tipo avente un nome molto lungo con un alias piú corto. Puó essere utile anche per "mascherare" valori veri con nomi di comodo, di modo che da fuori da una classe i dati appaiano con nomi piú semplici da comprendere.

```
typedef old_name new_name;
```

```
typedef unsigned long int uli;

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

`const` é un modificatore che, posto davanti alla definizione di una variabile, la rende immutabile, ovvero non é piú possibile modificarne il valore in un secondo momento. Permette di creare delle costanti, ovvero valori che devono imprescindibilmente assumere uno ed un solo valore[4]. Se si tenta di aggiungere `const` ad una variabile che non viene inizializzata quando viene dichiarata viene restituito un messaggio di errore.

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

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

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

pi = 3.1415;                 // NOT Allowed

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

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

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

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

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

[4] `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.

```
int i = 200;

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

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

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

1.5. Stringhe

Una **stringa** é un tipo di dato che permette di memorizzare informazioni alfanumeriche. In C, le stringhe sono degli array di `char` il cui ultimo carattere é il carattere speciale `\0`. Quando a `cout` viene fornito un array di `char` con queste caratteristiche, vengono ordinatamente stampati tutti i caratteri dell'array ad eccezione di `\0`. Un puntatore a `char` viene interpretato come una stringa, pertanto non é possibile, a meno di usare una sintassi particolarmente convoluta, riferirsi ad una stringa tramite un puntatore. Le stringhe del C hanno dei metodi che si trovano nell'header file `string.h` (o `cstring`).

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

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

```
#include <string>

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

Le stringhe C sono ancora utilizzate come argomenti dalla riga di comando. Infatti, la sintassi standard[5] della funzione `main` completa é la seguente:

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

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

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

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

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

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

1.6. Casting

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

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

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

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

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

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

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

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

1.7. Ciclo di vita delle variabili

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

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

Una variabile é globale se non appartiene a nessuna funzione o classe. Tale variabile é accessibile da qualsiasi punto dell'unitá di compilazione corrente, e se dichiarata con il modificatore `extern` é accessibile anche da qualsiasi altra unitá di compilazione che la importa. Esistono all'interno della memoria fintanto che il programma é in esecuzione.
Definire una variabile globale che deve essere accessibile da ogni singola unitá di compilazione del codice é una delle poche situazioni in cui puó avere senso avere un file `main.h`.
Una variabile é automatica se viene allocata e deallocata automaticamente nello/dallo stack. Sono le variabili comunemente intese, che si trovano all'interno di una funzione e che esistono solamente fintanto che tale funzione é in esecuzione. Sono (potenzialmente) accessibili solamente dal blocco in cui sono state definite. Naturalmente, le variabili automatiche dichiarate all'interno di `main()` esistono fino alla fine dell'esecuzione del programma.

```cpp
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**[6].

```
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.

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

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

Obj* o = new Obj;

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

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

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

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

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

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

array[3][2] = 7;

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

delete[] array;
```

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

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

```
name_s::entity
```

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

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

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

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

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

```
namespace First
{
  int s;
}

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

1.8. Funzioni

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

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

Ad eccezione degli array statici, che possono essere passati solo e soltanto tramite puntatore, qualsiasi dato può essere passato in tutti e tre i modi, pertanto scegliere una modalità piuttosto che l'altra dipende (quasi) soltanto da

cosa si vuole ottenere. Il passaggio per valore è da preferirsi per 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.
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`.

```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 può ritornare dati di qualsiasi tipo, ad eccezione degli array, dei quali è possibile ritornare esclusivamente un puntatore che vi si riferisce.
è 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.
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. Naturalmente, due funzioni con

la stessa identica signature (stesso namespace, stesso nome, stesso nome, tipo e ordine dei parametri) non sono ammesse, ed il compilatore restituisce un messaggio d'errore.
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.

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

```
#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.

2. Appendice

2.1. 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...
*/
```

```
/**
  @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);
```