

# **A Simple Guide to Modern C++**

*Beginner*

*Intermediate*

*Advanced*

Luca Rengo

The Publisher

# Indice

<b>Indice</b>	<b>2</b>
<b>1 Basi del Linguaggio</b>	<b>7</b>
1.1 Introduzione . . . . .	7
1.2 Linguaggio . . . . .	7
1.3 Preprocessor . . . . .	8
1.4 Compilatore . . . . .	8
1.5 Linker . . . . .	9
1.6 Runtime Vs Compile time . . . . .	9
1.7 C vs C++ . . . . .	10
1.8 Tipi di Dati . . . . .	12
Cast . . . . .	13
1.9 Costanti . . . . .	15
const vs constexpr . . . . .	15
static . . . . .	15
Variabili statiche . . . . .	15
Membri statici delle classi . . . . .	15
static const . . . . .	16
static constexpr . . . . .	16
1.10 Arrays e Matrici . . . . .	16
Arrays . . . . .	16
Matrici . . . . .	17
1.11 Operatori Aritmetici . . . . .	18
1.12 Operatori Relazionali . . . . .	18
1.13 Operatori Bitwise . . . . .	19
1.14 Operatori di Assegnamento e Operatori Unari . . . . .	20
Operatori di Assegnamento . . . . .	20
Operatori Unari . . . . .	21
1.15 Operatori Logici . . . . .	23
1.16 Altri Operatori . . . . .	23
1.17 Condizione If . . . . .	25

If else if else . . . . .	25
Operatore Ternario . . . . .	25
1.18 Switch . . . . .	26
Switch Cases Break . . . . .	26
Default Case . . . . .	27
1.19 Loops . . . . .	28
While . . . . .	28
Do-While . . . . .	28
Continue . . . . .	29
goto . . . . .	29
For . . . . .	30
Foreach . . . . .	30
Cicli annidati . . . . .	30
Cicli Infiniti . . . . .	30
1.20 Enumeratori . . . . .	31
1.21 Puntatori . . . . .	33
Aritmetica dei puntatori . . . . .	33
Puntatori a puntatori . . . . .	34
1.22 References . . . . .	35
References vs Puntatori . . . . .	36
1.23 Stringhe . . . . .	37
Char . . . . .	37
C-string . . . . .	37
char* . . . . .	37
Tabella ASCII . . . . .	38
std::string . . . . .	38
char* vs std::string vs char[] . . . . .	39
Usare char* . . . . .	39
Usare std::string . . . . .	40
Casi in cui preferire char* ad std::string . . . . .	40
Usare char[] . . . . .	40
1.24 Funzioni . . . . .	41
return . . . . .	41
void . . . . .	42
main . . . . .	42
Funzioni ricorsive . . . . .	43
Argomenti passati per valore . . . . .	44
Argomenti passati per referenza . . . . .	44
Funzioni che ritornano puntatori . . . . .	46
1.25 Variables Scope . . . . .	47
Variabili Locali . . . . .	47

Parametri formali . . . . .	48
Variabili Globali . . . . .	48
1.26 Header files . . . . .	49
Only Once Headers   pragma once   ifndef . . . . .	49
Cosa sono le librerie? . . . . .	50
Header files libreria Standard . . . . .	50
Librerie create dagli utenti . . . . .	51
Differenza tra .h vs .hpp . . . . .	51
1.27 Namespaces . . . . .	52
std:: vs using namespace std . . . . .	53
Mai mettere using namespace in un header file! . . . . .	55
1.28 Strutture . . . . .	56
typedef . . . . .	57
Funzioni nelle strutture . . . . .	58
Strutture nelle strutture . . . . .	58
Puntatore ad una struttura . . . . .	59
Array di Strutture . . . . .	60
Strutture come parametri e come ritorno . . . . .	60
Strutture in C vs in C++ . . . . .	63
1.29 Union . . . . .	63
structure vs union . . . . .	65
1.30 Classi . . . . .	66
Costruttori e Distruttori . . . . .	67
Costruttori . . . . .	67
Initialization List . . . . .	67
Distruttori . . . . .	68
Proprietà del distruttore . . . . .	68
Quando viene chiamato il distruttore? . . . . .	69
Access modifiers . . . . .	69
Incapsulamento . . . . .	69
scope resolution operator :: . . . . .	71
Getters & Setters . . . . .	72
Ereditarietà . . . . .	72
this pointer . . . . .	75
Multi-Ereditarietà . . . . .	75
Forward Declaration . . . . .	77
Chiamata a funzione statica e a membro . . . . .	78
Static nelle Classi . . . . .	78
Funzioni e la keyword const . . . . .	79
Class vs Struct . . . . .	80
1.31 Convenzioni del linguaggio . . . . .	81

Generale . . . . .	81
Parentesi Graffe . . . . .	81
Indentazione . . . . .	81
Convenzioni sui nomi . . . . .	82
Ordine Inclusione Header files . . . . .	82
Files . . . . .	82
Types . . . . .	83
Variabili . . . . .	83
Funzioni . . . . .	83
Costanti . . . . .	83
Namespaces . . . . .	83
Header Guards . . . . .	84
Spazi bianchi aggiuntivi . . . . .	84
Linee Guida Miste . . . . .	86
<b>2 Concetti Intermedi</b>	<b>87</b>
2.1 Introduzione . . . . .	87
2.2 Templates . . . . .	88
2.3 std::vector<> . . . . .	90
2.4 Iteratori . . . . .	92
2.5 Virtual . . . . .	96
Virtual functions . . . . .	96
Virtual Destructors . . . . .	98
Virtual Inheritance . . . . .	99
2.6 Polimorfismo . . . . .	101
2.7 Overloading . . . . .	102
Function Overloading . . . . .	102
Operator Overloading . . . . .	102
Overloading vs Overriding . . . . .	104
2.8 Tipi di Casts . . . . .	106
static_cast<> . . . . .	106
const_cast<> . . . . .	106
dynamic_cast<> . . . . .	107
reinterpret_cast<> . . . . .	108
C-style & function-style cast o Regular Cast . . . . .	109
Ricapitolando . . . . .	110
2.9 Lambdas . . . . .	110
2.10 Memoria dinamica . . . . .	113
Memoria Dinamica in C . . . . .	113
new e delete . . . . .	113
new . . . . .	113

array normali vs array con la new . . . . .	114
delete . . . . .	114
Evitare di usare new . . . . .	114
2.11 RAII   Resource Acquisition is initialization . . . . .	115
2.12 Constructor types   Rules of . . . . .	115
Rule of Zero . . . . .	115
Copy Constructor . . . . .	116
Copy Assign . . . . .	117
=default   Defaulted Functions . . . . .	118
=delete   Deleted Functions . . . . .	119
Copy Constructor vs Copy Assignment Operator . . . . .	120
Rule of Three . . . . .	120
Move Constructor . . . . .	121
lvalues references & rvalues references . . . . .	123
Move Assignment Operator . . . . .	124
Rule of Five . . . . .	125
2.13 Move Semantics . . . . .	126
Fallbacks of move semantics . . . . .	127
syntax vs semantics . . . . .	127
Ricapitolando le Move Semantics . . . . .	129
2.14 Classi Astratte . . . . .	130
Pure Virtual Functions . . . . .	130
Abstract Class . . . . .	130
Abstract class vs Interface . . . . .	132
2.15 Eccezioni . . . . .	133
try catch throw . . . . .	133
try . . . . .	133
catch . . . . .	133
std::throw exception . . . . .	134
Errori a compile time ed errori a runtime . . . . .	135
2.16 Operazioni su file . . . . .	136

# Basi del Linguaggio

# 1

---

## *Introduzione*

---

Questa è una semplice e breve guida sul linguaggio C++.

Non insegna a programmare, semplicemente è una collezione di frammenti di codice e spiegazioni delle sintassi del linguaggio e alcune accortezze e good practices.

Questa è una guida per chi ha già familiarità con altri linguaggi di programmazione, come Java, C# e vorrebbe avvicinarsi al C++.

Questa guida volge attorno alla versione C++17, ma vedremo anche alcuni concetti di C++20. Mentre, l'ultima preview mostrata è stata quella del C++23.

---

## *Linguaggio*

---

Il C++ è un linguaggio di programmazione general purpose nato nel 1983 da Bjarne Stroustrup nei Bell Labs come evoluzione del C.

Il nome del linguaggio deriva dal C, ma con l'aggiunta dell'operatore ++ che nel C serve per incrementare di 1. Il che stava a significare che il C++ è come il C, ma migliore, ovvero come suo successore.

---

## *Preprocessor*

---

Il **preprocessore** è composto da delle direttive che danno istruzioni al compilatore di preprocessare prima dell'effettiva compilazione.

Per esempio di includere una libreria standard del linguaggio.

Le direttive del preprocessore iniziano con il #.

```
// Con la direttiva #include diciamo al preprocessore di includere  
    questo file  
// che in questo caso si tratta della libreria standard per l'input ed  
    output  
// del C++  
#include <iostream>
```

---

## *Compilatore*

---

Il C++ è un linguaggio di programmazione, di solito, implementato tramite un **compilatore** (un *traduttore* che converte il *codice sorgente* in *codice macchina* object byte code) invece di un **interprete** (che esegue il direttamente il codice sorgente).



---

## *Linker*

---

Se il compilatore traduce il codice sorgente, il **linker** si occupa di risolvere le referenze ad altri files, per esempio all'inclusione di librerie standard del linguaggio. Quindi il linker linka i files che servono al tuo codice per essere eseguito al tuo codice (che dopo la compilazione sarà probabilmente diventato un .OBJ)

---

## *Runtime Vs Compile time*

---

**Runtime** sono le istruzioni di codice che vengono eseguite mentre il tuo programma si trova in esecuzione.

**Compile time** sono le istruzioni che vengono tradotte dal compilatore nel momento della compilazione.

---

## *C vs C++*

---

C	C++
Sviluppato da Dennis Ritchie tra gli anni 1969 e 1973 agli AT&T Bell Labs.	Sviluppato da Bjarne Stroustrup tra il 1979 ed il 1983.
Non supporta il polimorfismo, incapsulazione, ereditarietà il che significa che il C non supporta la programmazione ad oggetti.	Supporta polimorfismo, incapsulazione ed ereditarietà il che significa che è un linguaggio di programmazione ad oggetti.
C è un sottoinsieme del C++.	Il C++ è un sovrainsieme del C.
Il C contiene 32 keywords.	Il C++ 63 keywords.
Il C supporta la programmazione procedurale.	Il C++ è un ibrido, supporta sia la programmazione procedurale sia i paradigmi della programmazione ad oggetti.
Dati e funzioni sono separati perché è un linguaggio di programmazione procedurale.	Dati e funzioni sono incapsulati in forma di un oggetto.
Non supporta l'information hiding.	I dati sono incapsulati per garantire che vengano usati come inteso.
Tipi incorporati sono supportati dal C. (typedef).	Tipi incorporati e tipi definiti dall'utente sono supportati.
Basata sulle funzioni perché è un linguaggio procedurale.	Basata sugli oggetti perché è un linguaggio ad oggetti.
L'overloading delle funzioni e degli operatori non è supportato.	L'overloading delle funzioni e degli operatori è supportato.
Non si possono definire funzioni dentro le strutture.	Si possono definire funzioni dentro le strutture.
Namespaces non esistono in C. File header di input ed output in C è stdio.h.	Namespace sono presenti in C++. File header di input ed output in C++ è iostream.
Le variabili di referenza non sono supportate.	Le variabili di referenza sono supportate.
Le funzioni virtual e	Le funzioni virtual e

C	C++
friends non sono supportate.	friends sono supportate.
Non supporta l'ereditarietà.	Supporta l'ereditarietà.
Si concentra sulle funzioni più che sui dati.	Si concentra sui dati più che sulle funzioni.
Fornisce malloc(), calloc(), realloc() e free() per l'allocazione/deallocazione dinamica della memoria.	Fornisce gli operatori new e delete (e altro) per l'allocazione/deallocazione dinamica della memoria.
Supporto diretto per occuparsi delle eccezioni non è supportato.	Supporto diretto per occuparsi delle eccezioni è supportato.
scanf() e printf() sono usate per l'input/output in C.	cin e cout sono usate per l'input/output in C++.
Le strutture in C non hanno modificatori d'accesso.	Le strutture in C++ hanno modificatori d'accesso.

---

## *Tipi di Dati*

---

Il C++ possiede diverse tipologie di memorizzazione dei dati:

Type	Size (in bytes)	Range
char	1	-127 to 127 or 0 to 255
unsigned char	1	0 to 255
int	4	-2147483648 to 2147483647
unsigned int	4	0 to 4294967295
short int	2	-32768 to 32767
unsigned short int	2	0 to 65,535
long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
float	4	+/- 3.4e +/- 38 (~7 digits)
double	8	+/- 1.7e +/- 308 (~15 digits)

Figura 1.1: Tabelle dei tipi

E dei wrappers sui tipi **unsigned** (ovvero senza segno, ovvero sempre e solo positivi), come:

<code>size_t</code>	corrisponde ad unsigned int
<code>uint8_t</code>	unsigned char
<code>uint16_t</code>	unsigned int
<code>uint32_t</code>	unsigned long

E altri tipi di dati per lavorare sui caratteri e sulle stringhe:

<code>string</code>	(includere <code>string</code> )
<code>wchar_t</code>	wide char (per caratteri più grandi di 255)

Altri tipi di dati:

<code>bool</code>	Un booleano : o 0, ovvero FALSO/OFF o 1, ovvero TRUE/ON
<code>std::byte</code>	8 bit (definito in <code>&lt;cstdint&gt;</code> header file)
<code>register</code>	un registro
<code>auto</code>	trova la tipologia automaticamente

Esistono anche altri tipi, ma quelli qui riportati sono tra i più comuni.

## Cast

**Definizione:** Il **Cast** è un'operazione che permette di cambiare la tipologia di una variabile o di una determinata operazione matematica.

Ovviamente, questa operazione potrebbe portare ad una perdita di dati, in particolare, quando facciamo un cast da una tipologia più grande (che occupa più spazio, più bits) ad una più piccola (che occupa meno spazio, meno bits) perché quella piccola non ha lo stesso spazio di immagazzinamento di quella grande.

Per esempio se si fa un cast da una tipologia a 32 bit ad una a 8 bit, ovviamente si perderanno dei dati, perché quella da 8 non può contenere gli stessi dati di una da 32.

Per fare un cast bisogna mettere, prima dell'operazione, tra parentesi la tipologia a cui si vuole castare. Esempio: `(float) 5 / 2 = 3.5`

*// Primo esempio:*

```
int a = 7, b = 2;  
float c;
```

```
c = a / b; // Output : 3
```

```
c = (float) a / b; // Output : 3.5
```

*// Secondo esempio:*

```
double d = 36.9;  
float f = 22.2;  
int x;
```

```
x = (int) d; // Output : 36  
x = (int) f; // Output : 22
```

// Ovviamente una variabile intera non può contenere i dati delle variabili con la virgola e quindi le informazioni sulla virgola vengono perse.

Questo è un cast semplice, ci sono altre forme di cast un po' più complesse che vedremo in un altro capitolo.

---

## *Costanti*

---

Una costante è un valore che non cambia mai.

Ci sono diversi tipi di costanti e con significato diverso, per esempio `const` e `constexpr`.

### *const vs constexpr*

<b>const</b>	<b>constexpr</b>
può essere composta da altre variabili a run-time può essere usata solo per non-static member functions	deve essere conosciuta a compile-time può essere usata sia per member e non-member functions e anche costruttori

## *static*

### *Variabili statiche*

Viene allocata per l'intera durata del programma. Anche se la funzione è chiamata molteplici volte, lo spazio allocato per la variabile statica è allocato una volta sola.

### *Membri statici delle classi*

#### **Istanze delle classi come statiche**

I distruttori (funzioni che rimuovono l'allocazione di memoria di un oggetto classe) vengono invocati soltanto dopo la fine del main (funzione principale da cui parte tutto il programma).

#### **Funzioni statiche in una classe**

Queste possono soltanto accedere a dati statici o altre funzioni statiche.

## *static const*

Per quanto riguarda `static const` possono ottenere un valore a compile time o a runtime, proprio come `const`, ma solo accessibili nella data funzione/classe.

## *static constexpr*

```
static constexpr int width = 24;
static constexpr int height = 24;
/* static: Ogni istanza della classe condivide la stessa variabile/costante. Non ne viene creata una copia per ciascuno
 * constexpr (constant expression) significa che questa è una costante e che il valore non verrà cambiato e che è
 * conosciuto al momento della compilazione
 * Qual è la differenza tra const e constexpr?
 * constexpr sarà sempre il valore assegnato (in questo esempio 24); non potrà mai cambiare.
 * const invece vuol dire che una volta inizializzata il valore non può cambiare, ma può avere un valore diverso
 * Esempio: const int right = x + width;
 * Come si può evincere dall'esempio: il valore resterà lo stesso, ma dipenderà dal valore di x e width (in questo
 * esempio) e ogni volta che chiamiamo la funzione , il valore può essere diverso.
 * Mentre con constexpr questo non è possibile!
 */
```

Figura 1.2: Tipi di costanti

---

## *Arrays e Matrici*

---

### *Arrays*

**Definizione:** Gli array sono dei contenitori di dati, una collezione di dati dello stesso tipo.

Il primo elemento di un array, come qualsiasi altra cosa in informatica è l'elemento 0, non l'elemento 1.

Quindi l'elemento di indice 0 è il primo elemento, quello di indice 1 è il secondo, quello di indice 2 è il terzo e così via..

```
// tipologia nomeDelArray[ spazioOccupato ];
```

```
double dArray [ 3 ];
```

```
// Qui potremmo usare un loop per definire questi elementi,
// ma li vedremo dopo.
```



```
dArray[0] = 12.4;
dArray[1] = 37.9;
dArray[2] = 19.1;

// Possiamo assegnarli anche quando definiamo l'array.
int array[5] = {7, 9, 12, 4, 11};

// Potremmo anche non definire il size (spazio dell'array).
int array2[] = {3, 6, 9};

// Ma è raccomandabile usare dei contenitori come: std::vector
// oppure una lista.
// Oppure dei puntatori.
// Oppure dei smart pointers.
```

## Matrici

**Definizione:** Le matrici sono degli arrays organizzati su righe e colonne. Questo concetto è per una matrice di 2-dimensioni. La si può pensare proprio come una tabella, formata da righe e da colonne.

```
#include <iostream>

// <iostream> è un file di intestazione (header file) della libreria
// standard per poter lavorare sull'input e sull'output.

// tipologia nome_matrice [righe][colonne];

int matrix[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}}

for(int i = 0; i < 3; i++){
    for(int j = 0; j < 3; j++){
        std::cout << "elemento di riga " << i <<
            " e colonna " << j << matrix[i][j] <<
            std::endl;
    }
}
```

Ovviamente, come in matematica, è possibile eseguire varie operazioni sulle matrici, come: trasposizione, moltiplicazione, somma, sottrazione, ecc..(ma che qui non mostrerò).

---

## *Operatori Aritmetici*

---

**Definizione:** Gli operatori aritmetici permettono di eseguire qualsiasi operazione aritmetica.

- `+` : somma.
- `-` : sottrazione.
- `*` : moltiplicazione.
- `/` : divisione.
- `%` : modulo, restituisce il resto della divisione.

```
int a = 8, b = 3;
```

```
a + b; // Output : 11
```

```
a - b; // Output : 5
```

```
a * b; // Output : 24
```

```
a / b; // Output : 2
```

```
a % b; // Output : 2 (resto della divisione)
```

---

## *Operatori Relazionali*

---

**Definizione:** Gli operatori relazionali servono per controllare la relazione tra due operandi.

- `==` : per l'equivalenza; controllare se due operandi sono uguali.

- `!=` : per controllare se due operandi non sono equivalenti.
- `>` : per controllare se un operando è maggiore dell'altro
- `>=` : per controllare se un operando è maggiore o uguale all'altro.
- `<` : per controllare se un operando è minore di dell'altro.
- `<=` : per controllare se un operando è minore o uguale all'altro.

```
int x = 5, y = 3;  
x == y // Output : FALSE  
x != y // Output : TRUE  
x > y // Output : TRUE  
x >= y // Output : TRUE  
x < y // Output : FALSE  
x <= y // Output : FALSE
```

---

## Operatori Bitwise

---

**Definizione:** Gli operatori bitwise servono per lavorare sui singolo bits di dati.

- **& (bitwise AND)** : permette di fare un AND bit a bit sui due operandi. Il risultato è 1 soltanto se entrambi sono 1.
- **| (bitwise OR)** : permette di fare un OR bit a bit su ogni bit dei due operandi. Il risultato è 1 se almeno uno dei due bits è a 1.
- **^ (bitwise XOR)** : permette di fare uno XOR bit a bit su ogni bit dei due operandi. Il risultato è 1 se i due bits sono differenti.
- **<< (left shift)** : prende due numeri. Shifta a sinistra i bits del primo operando, il secondo operando decide di quanti bits si deve shiftare il primo.
- **>> (right shift)** : prende due numeri. Shifta a destra i bits del primo operando, il secondo operando decide di quanti bits si deve shiftare il primo.

- **(bitwise NOT)** : prende un numero ed inverte tutti i bits.

```
// a = 5 in binario è 00000101, b = 9 in binario è 00001001  
int a = 5, b = 9;
```

```
a & b; // Output : 00000001
```

```
a | b; // Output : 00001101
```

```
a ^ b; // Output : 00001100
```

```
~a; // Output : 11111010
```

```
b << 1; // Output : 00010010
```

```
b >> 1; // Output : 00000100
```

---

## *Operatori di Assegnamento e Operatori Unari*

---

### *Operatori di Assegnamento*

**Definizione:** Gli operatori di assegnamento sono usati per assegnare un valore alle variabili.

- **=** : Operatore di assegnamento di un valore ad una variabile.
- **+=** : Combinazione di = e +, aggiunge l'operando di destra a quello di sinistra e lo assegna a quello di sinistra.
- **-=** : Combinazione di = e -, sottrae l'operando di destra a quello di sinistra e lo assegna a quello di sinistra.
- **\*=** : Combinazione di = e \*, moltiplica l'operando di destra a quello di sinistra e lo assegna a quello di sinistra.
- **/=** : Combinazione di = e /, divide l'operando di destra a quello di sinistra e lo assegna a quello di sinistra.

- `%=` : Combinazione di `=` e `%`, ottiene il resto dall'operando di destra a quello di sinistra e lo assegna a quello di sinistra.
- `<<=` : Combinazione di `=` e `<<`, left shifta l'operando di destra a quello di sinistra e lo assegna a quello di sinistra.
- `>>=` : Combinazione di `=` e `>>`, right shifta l'operando di destra a quello di sinistra e lo assegna a quello di sinistra.
- `&=` : Combinazione di `=` e `&`, bitwise AND sull'operando di destra a quello di sinistra e lo assegna a quello di sinistra.
- `^=` : Combinazione di `=` e `^`, bitwise XOR sull'operando di destra a quello di sinistra e lo assegna a quello di sinistra.
- `|=` : Combinazione di `=` e `|`, bitwise OR sull'operando di destra a quello di sinistra e lo assegna a quello di sinistra.
- `<>=` : Bitwise shift left/right assignment

```
int x = 5; // = è un operatore di assegnamento.
int y;
```

`y += 3;` // `+=` è un operatore di assegnamento ed è la combinazione dell'operatore `=` e l'operatore `+`. Scrivere `y += 3;` è identico a scrivere `y = y + 3;` (ovvero `y` è uguale a se stesso + 3).

`y -= 2;` // identico a `y = y - 2;`

`y *= 4;` // identico a `y = y * 4;` (\* è un per, oppure viene usato nei puntatori)

`y /= 6;` // identico a `y = y / 6;`

## Operatori Unari

**Definizione:** Gli operatori unari operano su un operando per produrre un nuovo valore.

- `-` : nega il valore dell'operando.
- `++nome_variabale` : Incremento di 1 prefix, prima incrementa l'operando prima che venga eseguito.

- **nome\_variabile++** : Incremento postfix, il valore verrà incrementato dopo che è stato usato.
- **--nome\_variabile** : Decremento di 1 prefix, decrementa l'operando prima che venga usato.
- **nome\_variabile--** : Decremento postfix, il valore verrà decrementato dopo che è stato usato.
- **&nome\_variabile** : prima di una variabile, restituisce l'indirizzo di memoria della variabile in questione. In questo caso, NON è l'operatore bitwise AND &.
- **!nome\_variabile** : operatore not, inverte lo stato logico dell'operando. Se è TRUE allora lo modifica in FALSE, se è FALSE allora diventa TRUE.

```
int x = 3;
```

```
-x; // l'operatore - nega il valore dell'operando.
```

```
++x; // identico a scrivere x = x + 1; Questo è chiamato  
      incremento prefisso, perché: in questo modo il valore  
      dell'operando verrà alterato prima che venga usato.
```

```
x++; // identico a scrivere x = x + 1; L'operatore ++ incrementa di  
      1 il valore della variabile in questione. Questo è chiamato  
      incremento postfisso perché: in questo modo il valore verrà  
      modificato dopo che è stato usato.
```

```
--x; // identico a scrivere x = x - 1; Come per il ++ questo è un  
      decremento prefisso.
```

```
x--; // identico a scrivere x = x - 1; Decrementa di 1 il valore  
      della variabile in questione. Decremento post fisso.
```

```
&x; // l'operatore &, prima di una variabile, restituisce l'indirizzo  
      di memoria in cui la variabile risiede.
```

```
bool y = true;
```

```
!y; // Output : y è false; l'operatore ! (not) inverte lo stato logico  
      dell'operando.
```

---

## Operatori Logici

---

**Definizione:** Gli operatori logici servono per combinare due o più condizioni. Il risultato di un'operazione degli operatori logici è un booleano TRUE (VERO) o FALSE (FALSO).

- **&& (logical AND)** : restituisce vero se tutte le condizioni sono vere.
- **|| (logical OR)** : restituisce vero se almeno una delle condizioni è vera.
- **! (logical NOT)** : restituisce vero se la condizione è falsa e restituisce falso se la condizione è vera.
- **!** : Logical negation/bitwise complement

```
int x = 3, y = 6, z = 9;
```

```
(x > y) || (y != z); // Output : TRUE, perché anche se x  
                    > y è FALSA, y != z è VERA.
```

```
(y > x) && (y < z); // Output : TRUE perché entrambe  
                    sono vere.
```

```
!(x > 7); // Output : TRUE, perché x non è maggiore di 7,  
          quindi è falsa, ma il not inverte e quindi essendo la condizione  
          falsa, il not la inverte in VERA.
```

---

## Altri Operatori

---

Ci sono altri operatori come:

- **sizeof** : è usato per ottenere lo spazio che occupa una variabile.

- `,` : la virgola è usata sia come operatore che come separatore. valuta il primo operando e cancella il risultato, valuta il secondo operando e restituisce il suo valore.
- **Operatore Condizionale/Ternario** : condizione ? se vero esegui questo : se falso esegui questo.

```
sizeof(char); // Output : 1
sizeof(int); // Output : 4
sizeof(float); // Output : 4
sizeof(double); // Output : 8
```

```
int a = 0;
double d = 3.69;
```

```
sizeof(a); // Output : 4
sizeof(d); // Output : 4
```

```
sizeof(a + d); // Output : 8
```

```
int y = 2, x = 3; // Output : equivalente a int y = 2; int x =
3;
```

```
x >= 0 ? "x è maggiore o uguale a 0" : "x è
minore di 0"; // Output : "x è maggiore o uguale a 0".
```



---

## Condizione If

---

### *If|else if|else*

**Definizione:** L'if statement permette di decidere se un certo blocco di codice verrà eseguito o no.

L'else statement permette di eseguire un altro blocco di codice, casomai la condizione sia falsa.

else if(condizione) statement permette di fare un'ulteriore controllo dopo al primo if statement.

### *Operatore Ternario*

Un altro modo per valutare una condizione ed eseguire un codice è attraverso l'operatore ternario: condizione ? se è vera esegui questo : altrimenti esegui questo.

L'unica differenza con l'if è che si può eseguire una sola riga di codice sia nel caso la condizione sia vera sia falsa.

```
if (condizione)
{
    // Se (if) la condizione è vera esegui questo blocco di codice.
} else {
    // Altrimenti (else) esegui questo blocco di codice.
}

// Esempio: Cerchiamo il valore maggiore.
int x = 5, y = 3;

if (x > y)
{
    std::cout << "x è maggiore di y" << std::endl
    ;
}
else if (x == y) {
    std::cout << "x ed y sono uguali" << std::endl;
}
```

```
}  
else {  
    std::cout << "x è minore di y" << std::endl;  
}  
  
// Qui invece cerchiamo il valore minore.  
int a = 8, b = 7;  
int min;  
  
min = a < b ? a : b;  
  
std::cout << "Il valore minimo è: " << min << std  
::endl;
```

Se la riga da eseguire è 1 sola, allora si possono anche omettere le parentesi graffe.

---

## Switch

---

### Switch|Cases|Break

**Definizione:** Gli switch statements valutano una data espressione ed in base al valore di quella espressione, eseguono un determinato blocco di codice.

Le possibili espressioni che si possono mettere nello switch sono:

- Un numero intero, `int`
- Un enumeratore, `enum`
- Un carattere, `char` che è un piccolo intero tra -128 e +127.

Le varie scelte sono indicate nel **case**.

I cases son tutti collegati fra loro e quindi per far sì che solo un blocco di codice venga eseguito utilizziamo la keyword **break** per poter uscire dallo switch una volta che il codice è stato eseguito.

Se non mettessimo il **break** allora una volta eseguito un case, il codice che è sequenziale, eseguirebbe il case sotto. Possiamo evitare di metterlo se vogliamo che alcuni case eseguino lo stesso codice.

## *Default Case*

Infine c'è un case **default** nel caso che il valore valutato non sia presente tra i case. Questo case è opzionale, quindi si può anche non includere. Per il **default** non serve il **break** perchè è comunque l'ultimo case, però volendolo si può sempre mettere.

```
int scelta = 3;

switch(scelta){
    case 1:
        std::cout << "Scelta: 1" << std::endl;
        // Blocco di codice per il case 1.
        break;
    case 2:
        std::cout << "Scelta: 2" << std::endl;
        // Blocco di codice per il case 2.
        break;
    case 3:
        std::cout << "Scelta: 3" << std::endl;
        // Blocco di codice per il case 3.
        break;
    default:
        std::cout << "Nessuna scelta o scelta non
            prevista." << std::endl;
        // Blocco di codice per il default case.
        break;
}
```

---

## Loops

---

**Definizione:** I loops (cicli) ci permettono di ripetere un dato blocco di codice per un determinato o indeterminato numero di volte.

### While

I while loops ci permettono di eseguire un ciclo quando non conosciamo esattamente il numero di iterazioni.

La condizione del while viene valutata, se possibile entra dentro al loop altrimenti lo salta ed esegue il codice dopo.

Ad ogni iterazione la condizione viene controllata, se vera il ciclo continua, se falsa il ciclo viene interrotto.

```
while (condizione) {  
    // Blocco di codice da eseguire.  
}  
  
int x = 3;  
  
while (x < 5) {  
    std::cout << "Ciao per la " << x << "a volta "  
    << std::endl;  
    x++; // indentico a x = x + 1  
}
```

### Do-While

Nel Do while rispetto al singolo while, si entra almeno una volta all'interno del ciclo, poi come nel while viene controllata la condizione e se vera il ciclo continua altrimenti verrà interrotto.

```
do {  
    // Blocco di codice da eseguire.  
} while (condizione); // da notare il ; dopo il while.  
  
int x = 2;
```

```
do{
    std::cout << "Hello World!" << std::endl;
    x++;
}while(x < 1);
```

## Continue

La keyword **continue** è simile alla keyword **break**, ma invece di terminare l'esecuzione (del loop, dello switch, ecc..) , passa alla prossima iterazione del loop.

```
int a = 5;
do {
    if(a == 10){
        a++;
        continue;
    }
    std::cout << "Valore di a: " << a << std::endl;
    a++;
} while( a < 20);
```

## goto

La keyword **goto** permette di fare un salto incondizionato verso una label (etichetta).

Potrebbe essere utile per uscire dai cicli annidati (nested loops). L'uso del **goto** è scoraggiato ed è considerato una *bad practice* perché porta a quello che è definito *spaghetti code*, ovvero ad un codice destrutturato e difficile da mantenere.

```
int a = 10;

LOOP:do {
    if( a == 15) {
        // skip the iteration.
        a = a + 1;
        goto LOOP;
    }
    cout << "value of a: " << a << endl;
```

```
        a = a + 1;
    }
    while( a < 20 );
```

## *For*

Il for loop è composto da tre parti: l'inizializzazione della variabile contatore, la condizione, ed aggiornamento della variabile contatore.

A differenza del while loop, in questo conosciamo già a priori quanti cicli faremo.

Ad ogni iterazione del ciclo, la variabile contatore viene aggiornata.

```
for( inizializzazione; condizione; aggiornamento
    variabile ){
    // Codice da eseguire.
}

int n = 4;

for( int i = 0; i < n; i++){
    // Codice da eseguire
}
```

## *Foreach*

Questo loop è un po' più complicato e si avvale degli iteratori che verranno spiegati più avanti in un altro capitolo.

È definito nel file di intestazione `#include <algorithm>`.

## *Cicli annidati*

Si può inserire un loop dentro ad un altro loop (cicli annidati o nested loops).

## *Cicli Infiniti*

Bisogna fare attenzione a non creare cicli infiniti, che come dice la parola vanno ad oltranza, rallentano e bloccano il programma.

```
for( ; ; ){
    std::cout << "Loop Infinito" << std::endl;
}
```

---

## Enumeratori

---

**Definizione:** Gli enumeratori sono dei tipi di dati definiti dagli utenti ed usati per assegnare nomi a delle costanti intere, il che rende il codice semplice da leggere. Il primo elemento di un enum è di indice 0, ammeno che non lo si cambi, se lo si cambia, di conseguenza, cambiano anche tutti gli altri sotto.

```
enum Days { Lunedì , Martedì , Mercoledì , Giovedì ,  
           Venerdì , Sabato , Domenica };  
Days day = Venerdì ;  
  
if (day == Venerdì) {  
    std::cout << "Oggi è venerdì!" << std::endl;  
}  
  
enum Year {  
    GENNAIO = 1 ,  
    FEBBRAIO ,  
    MARZO ,  
    APRILE ,  
    MAGGIO ,  
    GIUGNO ,  
    LUGLIO ,  
    AGOSTO ,  
    SETTEMBRE ,  
    OTTOBRE ,  
    NOVEMBRE ,  
    DICEMBRE  
};  
  
Year mese = FEBBRAIO ;  
std::cout << "Siamo nel " << mese << "o mese dell  
'anno" << std::endl ;  
  
enum Colors {  
    ROSSO ,  
    BLU ,
```

```
    VERDE,  
    GIALLO,  
    ARANCIONE,  
    GRIGIO,  
    VIOLA,  
    ROSA,  
    NERO,  
    BIANCO  
};  
  
Colors colore = Colors.ARANCIONE;  
std::cout << "Colore di indice: " << colore <<  
    std::endl;  
  
// Per poter vedere il nome dell'enum e non il suo valore, bisogna  
    utilizzare una mappa o uno switch o altro.
```



---

## *Puntatori*

---

**Definizione:** Un puntatore è una variabile che contiene l'indirizzo di memoria di un'altra variabile. Si può dire che questa variabile punta ad un'altra. Per creare un puntatore usiamo l'operatore `*` che è lo stesso usato anche per la moltiplicazione.

Usiamo l'operatore `&` (indirizzo) per ottenere l'indirizzo di una variabile. Per ottenere il valore della variabile a cui il puntatore punta usiamo l'operatore `*` (dereferenza).

```
int x = 5;
int* ptr; // puntatore ad intero

ptr = &x; // il puntatore ptr punta alla variabile x

std::cout << var << std::endl; // Output : 5
std::cout << ptr << std::endl; // Output : indirizzo di
    x
std::cout << *ptr << std::endl; // Output : 5
```

## *Aritmetica dei puntatori*

Sui puntatori è possibile eseguire delle operazioni aritmetiche:

- Incremento|Decremento di un puntatore.
- Addizione di un intero ad un puntatore.
- Sottrazione di un intero ad un puntatore.
- Sottrazione di due puntatori dello stesso tipo.

```
// Primo esempio
#define MAX 3
int var[MAX] = {3, 6, 9};
int *ptr1, *ptr2;
```

```

ptr1 = &var[MAX - 1];

while ( ptr <= &var[MAX - 1] ) {
    std::cout << "Address of var[" << i << "] = "
    ;
    std::cout << ptr << endl;

    std::cout << "Value of var[" << i << "] = ";
    std::cout << *ptr << endl;

    // point to the previous location
    ptr++;
    i++;
}

// Secondo esempio

int x[10];
int *p1, *p2;
int i;

p1 = &x[3]; // P1 punta a x[3]
p2 = p1 + 2; // P2 punta a x[5]

p1 += 6; // P1 punta ad x[9];

p2 = p1 - 3; // P2 punta ad x[6];
p1 -= 7; // P1 punta ad x[2];

i = p2 - p1; // i: 6 - 2 = 4
i = p1 - p2; // i: 2 - 6 = -4

```

### ***Puntatori a puntatori***

Come esistono i puntatori ad una variabile, esistono anche dei puntatori ad altri puntatori.

Creiamo un puntatore ad un puntatore semplicemente aggiungendo un ulteriore \* al singolo puntatore.

```

int var = 369;
int *ptr;

```

```
int **pptr;

ptr = &var;

pptr = &ptr;

std::cout << "Valore di var: " << var << std::
    endl; // Output: Valore di var: 369
std::cout << "Valore di *ptr: " << *ptr << std::
    endl; // Output: Valore di *ptr: 369
std::cout << "Valore di **pptr: " << **pptr << std::
    endl; // Output: Valore di **pptr: 369
```

---

## References

---

**Definizione:** Una reference è come un alias, ovvero un altro nome per una variabile che già esiste. Come i puntatori è implementata attraverso la memorizzazione dell'indirizzo di memoria della suddetta variabile. Definiamo una reference attraverso l'operatore **&** prima del nome della variabile. Se facciamo qualcosa alla reference, all'alias, di conseguenza lo facciamo anche alla variabile a cui si riferisce.

```
int x = 10;

int& ref = x; // Questa è una reference alla variabile x.

ref = 20;
std::cout << "x: " << x << std::endl; // Output: 20

x = 30;
std::cout << "ref: " << ref << std::endl; //
    Output: 30
```

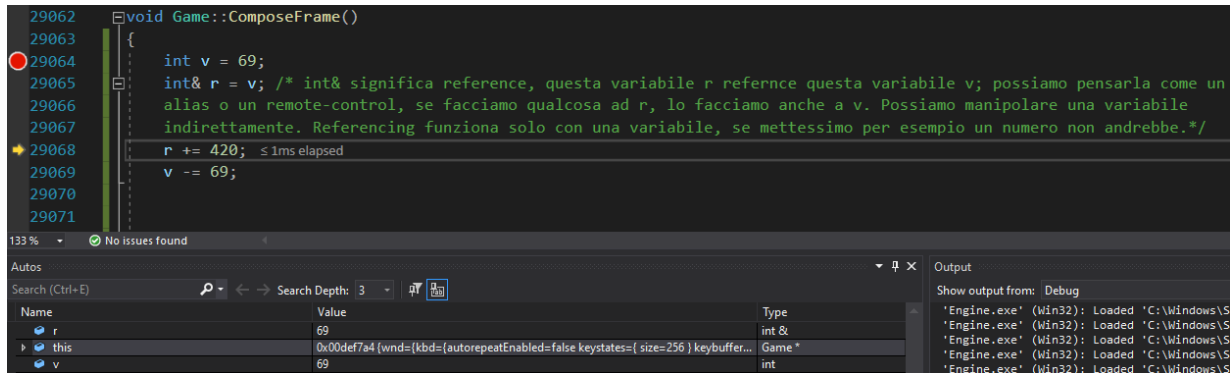


Figura 1.3: Reference

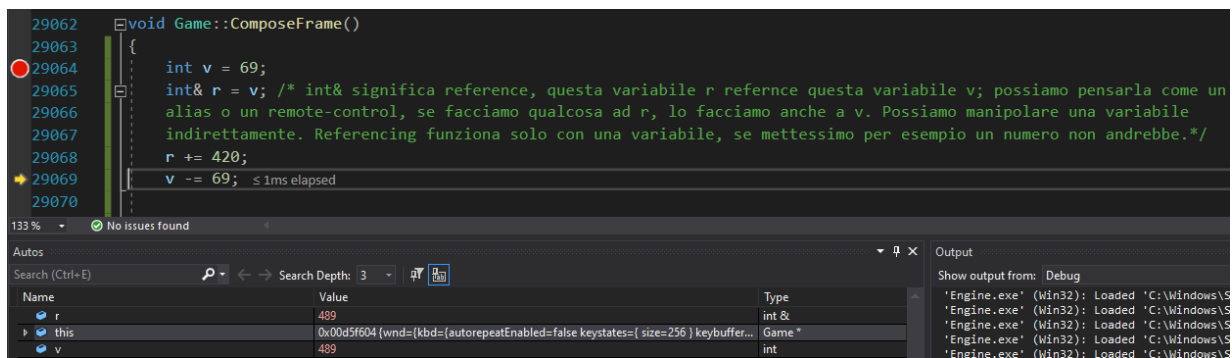


Figura 1.4: Reference

## References vs Puntatori

Reference	Pointers
Riferiscono ad una variabile con un altro nome	Memorizzano un indirizzo di una variabile
Non possono avere un valore <b>NULL</b> .	Possono avere un valore <b>NULL</b> .
Deve essere inizializzata alla dichiarazione	Può anche non essere inizializzata alla dichiarazione
Condivide la stessa memoria con la variabile originale, ma prende anche dello spazio nello stack.	Ha un proprio spazio e indirizzo di memoria sullo stack.
Non può essere riassegnato.	Può essere riassegnato.
Hanno un solo livello di indirezione.	Si possono avere puntatori a puntatori per livelli extra di indirezione.
Non c'è la aritmetica delle references	C'è l'aritmetica dei puntatori.

---

## Stringhe

---

### Char

**Definizione:** Un `char` è usato per memorizzare un singolo carattere  
Alternativamente, si possono usare i valori ASCII per indentificare le lettere

```
char linguaggio = 'C';
```

```
char linguaggio = 67; // 67 corrisponde a C nella tabella  
ASCII.
```

### C-string

Per creare una stringa in C facciamo un array (contenitore di dati dello stesso tipo) di `char`.

Il `'\0'` è il **NUL terminator** che denota la fine di una C-stringa.

```
char s[] = "prova";
```

```
// Oppure possiamo scriverlo:
```

```
char s[] = { 'p', 'r', 'o', 'v', 'a', '\0' };
```

```
// '\0' è il NUL terminator, denota la fine di una stringa.
```

### char\*

Un puntatore a `char` memorizza la locazione iniziale di una C-string (una stringa in C).

```
char s = "prova";
```

```
// Possiamo far puntatore al puntatore la prima cella dell'array  
così..
```

```
char* p = &(s[0]);
```

```
// ..oppure in maniera più coincisa così:
```

```
char *p = s;
```

## Tabella ASCII

**Definizione:** La tabella ASCII (*American Standard Code for Information Interchange*) è un codice per la codifica di caratteri.

Inizialmente era basata su codici di 7 bit, quindi per un totale di  $2^7 = 128$  caratteri. Venne poi estesa ad 8 bit, per un totale di  $2^8 = 256$  caratteri.

ASCII control characters			ASCII printable characters			Extended ASCII characters		
00	NULL	(Null character)	32	space	64	@	96	`
01	SOH	(Start of Header)	33	!	65	A	97	a
02	STX	(Start of Text)	34	"	66	B	98	b
03	ETX	(End of Text)	35	#	67	C	99	c
04	EOT	(End of Trans.)	36	\$	68	D	100	d
05	ENQ	(Enquiry)	37	%	69	E	101	e
06	ACK	(Acknowledgement)	38	&	70	F	102	f
07	BEL	(Bell)	39	'	71	G	103	g
08	BS	(Backspace)	40	(	72	H	104	h
09	HT	(Horizontal Tab)	41	)	73	I	105	i
10	LF	(Line feed)	42	*	74	J	106	j
11	VT	(Vertical Tab)	43	+	75	K	107	k
12	FF	(Form feed)	44	,	76	L	108	l
13	CR	(Carriage return)	45	-	77	M	109	m
14	SO	(Shift Out)	46	.	78	N	110	n
15	SI	(Shift In)	47	/	79	O	111	o
16	DLE	(Data link escape)	48	0	80	P	112	p
17	DC1	(Device control 1)	49	1	81	Q	113	q
18	DC2	(Device control 2)	50	2	82	R	114	r
19	DC3	(Device control 3)	51	3	83	S	115	s
20	DC4	(Device control 4)	52	4	84	T	116	t
21	NAK	(Negative acknowl.)	53	5	85	U	117	u
22	SYN	(Synchronous idle)	54	6	86	V	118	v
23	ETB	(End of trans. block)	55	7	87	W	119	w
24	CAN	(Cancel)	56	8	88	X	120	x
25	EM	(End of medium)	57	9	89	Y	121	y
26	SUB	(Substitute)	58	:	90	Z	122	z
27	ESC	(Escape)	59	;	91	[	123	{
28	FS	(File separator)	60	<	92	\	124	
29	GS	(Group separator)	61	=	93	]	125	}
30	RS	(Record separator)	62	>	94	^	126	~
31	US	(Unit separator)	63	?	95	_		
127	DEL	(Delete)						
128	Ç		160	á		192	Ł	Ó
129	ü		161	í		193	ł	ô
130	é		162	ó		194	Ţ	õ
131	â		163	ú		195	ţ	ö
132	ä		164	ñ		196	—	ø
133	à		165	Ñ		197	†	ő
134	å		166	ª		198	ä	µ
135	ç		167	º		199	Å	þ
136	ê		168	¿		200	Ĺ	ð
137	ë		169	®		201	Š	ú
138	è		170	¬		202	Œ	û
139	ĩ		171	½		203	Ť	ü
140	î		172	¼		204	ƒ	ý
141	ï		173	¾		205	ƒ	ÿ
142	Ā		174	«		206	†	·
143	Ă		175	»		207	‡	ˆ
144	É		176	≈		208	đ	≡
145	æ		177	×		209	Đ	±
146	Æ		178	÷		210	É	²
147	ö		179	×		211	Ê	¼
148	õ		180	×		212	Ë	½
149	ô		181	×		213	Ĭ	¾
150	ù		182	×		214	İ	÷
151	û		183	×		215	Í	×
152	ÿ		184	×		216	Ĵ	°
153	Ō		185	×		217	Ĵ	ˆ
154	Ū		186	×		218	Œ	˙
155	ø		187	×		219	Œ	ˆ
156	£		188	×		220	Œ	˙
157	Ø		189	×		221	Œ	˙
158	×		190	×		222	Œ	˙
159	ƒ		191	×		223	Œ	˙
								nbsp

Figura 1.5: Tabella ASCII

## std::string

**Definizione:** Il C++ ha una propria definizione per rappresentare una sequenza di caratteri come un oggetto di una classe. Questa classe è chiamata

`std::string`. Questa memorizza i caratteri come una sequenza di bytes con la funzionalità di poter accedere al singolo carattere byte.

La classe `std::string` ha diverse funzioni, come:

Funzione	Definizione
<b>length()</b>	restituisce la lunghezza della stringa.
<b>capacity()</b>	restituisce la capacità allocata alla stringa che può essere più o meno la lunghezza.
<b>resize()</b>	cambia la grandezza della stringa che può essere aumentata o diminuita.
<b>shrink_to_fit()</b>	diminuisce la grandezza della stringa e la rende uguale al minimo della capacità della stringa. Utile per salvare ulteriore memoria se siamo sicuri di non dover aggiungere altri caratteri.

Queste sono solo alcune delle funzioni della classe `string`.

```
std::string str = "Ciao a tutti";

str.resize(4);

std::cout << "Stringa dopo resize: " << str <<
std::endl; // Output: Stringa dopo resize: Ciao

std::cout << "Capacità della stringa: " << str.
capacity() << std::endl; // Output: Capacità della
stringa: 12

std::cout << "Lunghezza della stringa: " << str.
length() << std::endl; // Output: Lunghezza della
stringa: 4
```

### *char\* vs std::string vs char[]*

*Usare*

*char\**

```
char *str = "prova";
```

CONS	PROS
In C va bene, ma in C++ è deprecato, perché in C le stringhe sono array di char, mentre in C++ sono array di char costanti.	Basta un singolo puntatore per l'intera stringa. È efficiente a livello di memoria.
Non possiamo modificare la stringa dopo, possiamo semplicemente far puntare ad un'altra stringa.	Non c'è bisogno di dichiarare la lunghezza della stringa all'inizializzazione.

**Usare*****std::string***

```
std::string s = "prova";
```

CONS	PROS
	Con C++ std::string è la migliore via, perché ha delle funzioni di ricerca, rimpiazzo e manipolazione migliori.

***Casi in cui preferire char\* ad std::string***

- Quando si ha a che fare con livelli bassi di accesso, come interagire con il sistema operativo. Anche se std::string::c\_str dovrebbe occuparsi di quello.
- Compatibilità con del vecchio codice in C (Anche se la funzione std::string::c\_str dovrebbe già in largo modo occuparsi di questo).
- Per risparmiare memoria (std::string sicuramente occupa di più).

**Usare*****char[]***

```
// In realtà ci bastano 5 spazi nell'array, però se poi dopo vogliamo
// fare
// concatenazioni o manipolazioni sulle altre stringhe, ci servirà
// altro spazio.
char stringa[128] = "prova";
```

CONS	PROS
È un array allocato staticamente che consuma spazio nello stack.	Possiamo modificare la stringa anche in un altro stage del programma.
Dobbiamo utilizzare array di grandi dimensioni per poter concatenare o manipolare le altre stringhe, visto che lo spazio dell'array è fissato dall'inizio.	



---

## Funzioni

---

**Definizione:** Una funzione è un blocco di codice che esegue una specifico compito e può essere richiamato quando si vuole.

Le funzioni sono composte da: un tipo di dato di ritorno che è ciò che la funzione restituisce dopo esser stata eseguita, un nome, degli eventuali parametri ed racchiusa tra due parentesi graffe il corpo, il blocco di codice.

Per eseguire la funzione basta richiamarla col suo nome e passare gli eventuali parametri.

### *return*

La keyword **return** permette di restituire un valore/oggetto dalla funzione.

```
// tipologia nome_funzione(parametri)
{
    // Blocco di codice della funzione.
}

// Questa funzione restituisce un intero, si chiama somma, prende
// due parametri interi a e b e restituisce la somma tra a e b.
int somma(int a, int b){
    return a + b;
}

// Fuori dalla funzione
int x = 3, y = 5;
// chiamiamo la funzione somma, gli passiamo i parametri e il
// valore di ritorno lo assegniamo alla variabile intera z.
int z = somma(x, y); // Output z : 8
```

Tutto ciò che è creato all'interno della funzione è locale alla funzione e quindi non accessibile da fuori.

I nomi dei parametri sono soltanto dei placeholders. Potremmo anche non metterli e lasciare solo le tipologie, ma poi per poterli referenziare nella funzione non sapremmo come fare.

I parametri che vengono passati alla funzione sono anch'essi locali, a meno che non li si passano attraverso dei puntatori.

I parametri passati, a meno che con puntatori, sono delle copie delle variabili passate come parametro, e qualsiasi modifiche di queste copie non ha un effetto sui parametri passati.

```
// Questa è una funzione che restituisce un boolean (0 o 1 (VERO o FALSO)), chiamata isGreater che prende due variabili intere a e b come parametri e restituisce se true se la variabile a è maggiore della variabile b, altrimenti false.
```

```
// Questa funzione si potrebbe scrivere così..
```

```
bool isGreater(int a, int b)
{
    if(a > b){
        return a;
    } else {
        return b;
    }
}
```

```
//.. Oppure si potrebbe anche scrivere così.
```

```
bool isGreater(int a, int b)
{
    return a > b;
}
```

```
// Detto questo, per questo tipo di operazioni, ci sono già delle funzioni della libreria Standard ben più ottimizzate. Per questo esempio si potrebbe usare std::max.
```

## *void*

Se non volessimo ritornare niente dovremmo usare la tipologia **void**, questo tipo di funzione (che non ritorna niente) è chiamata **procedura**.

## *main*

Il **main** è la funzione principale di qualsiasi programma in C/C++, da esso parte il tutto, ha origine tutto.

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

## *Funzioni ricorsive*

Le funzioni ricorsive sono delle funzioni che richiamano se stesse per raggiungere un risultato.

```
// Il fattoriale di un numero, o anche scritto  $n!$  è  $n * (n - 1)$   
//  $4! = 4 * 3 * 2 * 1 = 24$ 
```

```
int fattoriale(int n)  
{  
    if ((n == 0) || (n == 1))  
        return 1;  
    else  
        return n * fattoriale(n - 1);  
}
```

```
// Questa funzione si potrebbe anche scrivere
```

```
int fattoriale(int n)  
{  
    // Le parentesi nella condizione non servirebbero, le lascio  
    // per chiarezza  
    return ((n == 0) || (n == 1)) ? 1 : n *  
        fattoriale(n - 1);  
}
```

```
// Fibonacci è una serie in cui i due primi elementi sono 1 e dove  
// ogni elemento è uguale alla somma dei due termini precedenti.
```

```
int fibonacci(int x)  
{  
    // Le parentesi nella condizione non servirebbero, le lascio  
    // per chiarezza  
    return ((x == 1) || (x == 0)) ? x : fibonacci  
        (x - 1) + fibonacci(x - 2);  
}
```

```
int main()  
{
```

```

int x = 4;
std::cout << "Fattoriale di " << x << " e\' :
" << fattoriale(x) << std::endl; // Output :
Fattoriale di 4 è 24.

int y = 15;
std::cout << "Fibonacci di " << y << " e\' "
<< fibonacci(15) << std::endl; // Output :
Fibonacci di 15 è 610.
return 0;
}

```

### *Argomenti passati per valore*

Quindi, quando passiamo dei valori (e non degli indirizzi di memoria alle variabili), si dice che passiamo gli argomenti **per valore**, quindi una copia delle variabili passate viene creata ed usata nelle funzioni.

Quindi noi non operiamo direttamente sulle variabili passate, ma sulle loro copie. Questo non ci permette di poter modificare le variabili passate.

```

int sottrazione(int a, int b)
{
    return a - b;
}

// Fuori dalla funzione
int x = 5, y = 3;
int z = sottrazione(x, y); // Output: 2

```

### *Argomenti passati per referenza*

Per poter effettivamente modificare le variabili che abbiamo passato per argomento, dobbiamo passarle con i puntatori, dobbiamo passare i loro indirizzi di memoria. Questo si chiama passare argomenti **per referenza**.

Se, per esempio, volessimo sostituire i valori di due variabili e li passassimo per valore, non riusciremmo.

```

// Parte 1: Usare argomenti passati per valore.
void swap(int a, int b)
{
    int temp = a;

```

```
    a = b;
    b = temp;
}

// Fuori dalla funzione
int x = 5, y = 3;
swap(x, y);
std::cout << "Valore di x dopo lo swap: " << x <<
    std::endl; // Output: 5
std::cout << "Valore di y dopo lo swap: " << y <<
    std::endl; // Output: 3
// Non funziona, noi vorremmo cambiare i valori di x ed y, ma così
// non funziona, perchè stiamo lavorando sulle copie delle
// variabili, non sulle variabili stesse.

// Parte 2: Usare argomenti passati per referenza
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Fuori dalla funzione
int x = 5, y = 3;
swap(x, y);
std::cout << "Valore di x dopo lo swap: " << x <<
    std::endl; // Output: 3
std::cout << "Valore di y dopo lo swap: " << y <<
    std::endl; // Output: 5

// Ha funzionato, perché abbiamo agito sulle variabili passate e
// non sulle loro copie.

// Quello visto prima era un modo per poter fare la funzione swap
// in C che funziona anche in C++, ma c'è anche un altro modo
// ovvero utilizzando le references.

void swap(int &a, int &b)
{
    int temp = a;
```

```

        a = b;
        b = temp;
    }

    // Fuori dalla funzione
    int x = 5, y = 3;
    swap(x, y);
    std::cout << "Valore di x dopo lo swap: " << x <<
        std::endl; // Output: 3
    std::cout << "Valore di y dopo lo swap: " << y <<
        std::endl; // Output: 5

    // Comunque c'è una funzione della libreria Standard std::swap()
    per questo.

```

## *Funzioni che ritornano puntatori*

Si possono, naturalmente, ritornare i puntatori dalle funzioni.

```

int* func()
{
    static int a = 11; // static così rimane sempre in
        memoria anche quando non si chiama la funzione
    return &a;
}

int *p;

p = func();

std::cout << p << std::endl; // Output: indirizzo di p
std::cout << *p << std::endl; // Output: 11

```

---

## *Variables Scope*

---

Variables Scopes, o in italiano, la portata delle variabili, significa fino a dove una variabile può essere utilizzata, fino a dove esiste, vale, la possiamo usare. La portata è una regione del programma, ci sono all'incirca 3 principali posti in cui le variabili possono essere dichiarate ed in base a questo le variabili assumono diversi nomi:

- **Locali** : dentro ad una funzione o ad un blocco di codice (racchiuso tra le graffe).
- **Parametri formali** : ovvero nella definizione della funzione, nei suoi parametri.
- **Globale** : fuori dalle funzioni.

## *Variabili Locali*

Le variabili create all'interno di una funzione o un blocco di codice, sono locali a quella funzione, possono essere utilizzate solo all'interno di quella funzione e non all'esterno. Una volta che la funzione termina, quella variabile cessa di esistere.

```
void funzione ()
{
    int a = 5;
    std::cout << "Valore variabile locale a: " <<
        a << std::endl;
}

// Fuori dalla funzione
funzione(); // Output : Valore variabile locale a: 5

std::cout << "Valore variabile locale a: " << a
    << std::endl; // Errore la variabile a non esiste!
```

## *Parametri formali*

Sono i parametri della funzione, esistono soltanto finchè la funzione esiste.

```
void funzione(int a)
{
    std::cout << "Valore variabile a: " << a <<
        std::endl;
}

int main()
{
    int x = 8;
    funzione(x); // Output Valore variabile a: 8

    std::cout << "Valore variabile a: " << a <<
        std::endl; // Errore non esiste in questo scope.

    return 0;
}
```

## *Variabili Globali*

Esistono per tutta la durata del programma, posso essere utilizzate anche all'interno delle funzioni e il loro valore non viene perso una volta che la funzione smette.

```
int x = 10;

void funzione()
{
    std::cout << "Valore variabile x: " << x <<
        std::endl;
}

int main()
{
    funzione(); // Output Valore variabile x: 10
    return 0;
}
```



---

## Header files

---

**Definizione:** Gli header files, o file di intestazione in italiano, sono dei files con l'estensione **.h** o **.hpp** che contengono le dichiarazioni delle funzioni e definizione di macro e tipi.

Sono un modo per organizzare il codice, possiamo includere gli elementi di questi files nel nostro codice attraverso la direttiva **#include** che informa il preprocessore di cercare questo file prima di continuare ad eseguire il codice. Esistono due tipi di header files: quelli standard del linguaggio/compiler e quelli creati dall'utente programmatore.

Per includere le librerie standard usiamo **#include <nomelibreria>** perché il compilatore sa dove si trovano queste librerie, mentre per le librerie definite dall'utente usiamo **#include "nomelibreria.h"** e passiamo anche il percorso di dove si trova. (ammesso che non si trova nella stessa cartella in cui si trova il nostro codice, in quel caso basta mettere il nome della libreria)

### *Only Once Headers | **pragma once** | **ifndef***

**Definizione:** Se un file header viene incluso due volte, il compilatore lo processerà il suo contenuto due volte, il che risulterà in un errore. Per evitare questo c'è una procedura standard da scrivere all'interno del file di intestazione.

```
#ifndef NOME_HEADER_FILE_H
#define NOME_HEADER_FILE_H
```

```
// Tra queste c'è il codice dell'header file.
```

```
#endif
```

La direttiva **#ifndef** controlla che il file non sia già stato aggiunto, se non è mai stato aggiunto, allora lo aggiunge, altrimenti salta il contenuto così che non verrà aggiunto due volte.

Inoltre, per fare questa stessa operazione, ma più semplice e corta esiste una direttiva non-standard: **#pragma once**.

```
#pragma once
```

```
// Contenuto dell'header.
```

## *Cosa sono le librerie?*

Le librerie sono collezioni di risorse non volatili usate dai programmi. La libreria Standard è una collezione di classi, funzioni, macros, costanti, ecc.. che sono state scritte in C++ stesso. Ci sono una grande lista di header files che contengono i contenuti della libreria Standard.

## *Header files libreria Standard*

Qui, una lista degli header files della libreria standard più comuni (alcuni anche del C):

<b>#include &lt;stdio.h&gt; :</b>	per l'input ed output (dal C).
<b>#include &lt;iostream&gt; :</b>	input ed output fondamentali.
<b>#include &lt;string&gt; :</b>	fornisce le standard classi string e template.
<b>#include &lt;math.h&gt; :</b>	per operazioni matematiche (dal C).
<b>#include &lt;limits&gt; :</b>	usata per descrivere proprietà di tipi numerici fondamentali.
<b>#include &lt;time.h&gt; :</b>	per funzioni legate al tempo (dal C).
<b>#include &lt;chrono&gt; :</b>	fornisce elementi di tempo, come <code>std::chrono::duration</code> e <code>std::chrono::time_point</code> ed altri.
<b>#include &lt;algorithm&gt; :</b>	fornisce la definizione di molti container algoritmici.
<b>#include &lt;iterator&gt; :</b>	fornisce templates e classi per lavorare con gli iteratori.
<b>#include &lt;sstream&gt; :</b>	fornisce delle classi per la manipolazione di stringhe.
<b>#include &lt;vector&gt; :</b>	fornisce la classe di template container <code>std::vector</code> , un array dinamico.
<b>#include &lt;random&gt; :</b>	facilita la generazione di numeri (pseudo-)casuali e distribuzioni.
<b>#include &lt;numeric&gt; :</b>	operazioni numeriche generalizzate.
<b>#include &lt;functional&gt; :</b>	fornisce diverse oggetti funzionali da usare con gli standard algorithm.
<b>#include &lt;stdexcept&gt; :</b>	classi per le eccezioni.
<b>#include &lt;memory&gt; :</b>	per la gestione della memoria.
<b>#include &lt;optional&gt; :</b>	per gli opzionali.
<b>#include &lt;ranges&gt; :</b>	per i ranges e per i lazy evaluated adaptors. (C++20)
<b>#include &lt;concepts&gt; :</b>	fornisce la libreria fondamentale concepts. (C++20)
<b>#include &lt;thread&gt; :</b>	fornisce classi e namespaces per lavorare sui threads.

Inoltre, tutti gli headers dalla libreria standard del C sono inclusi nella libreria standard del C++

Ci sono tanti altri headers file e ognuno usato per qualcosa..

## *Librerie create dagli utenti*

Gli utenti si possono creare le proprie librerie, creando un file **.h** con le sole definizioni di funzioni e un file chiamato come l'header file, con le implementazioni di queste, ma con l'estensione **.cpp**.

Per includere queste librerie, usiamo **#include "nome\_libreria.h"**, al posto di **#include <nomelibreria.h>**, perché non una libreria standard e quindi il compilatore non sa dove cercarla e quindi gli dobbiamo specificare noi dove si trova la nostra libreria.

```
// Nel file header nomelibreria.h  
int somma(int a, int b);
```

```
// Nel file .cpp nomelibreria.cpp  
#include "nomelibreria.h"
```

```
int somma(int a, int b){  
    return a + b;  
}
```

## *Differenza tra .h vs .hpp*

In C++ l'estensione del file non è importante. L'uso di **.h**, **.hpp**, **.hxx**, **.hh**, **.tpp** o nessuna estensione sono tutte convenzioni.

<b>.h</b>	<b>.hpp</b>
Sia per il C che per il C++ Dal punto di vista del C++, il codice C verrà definito come <i>extern "C"</i> Esprime l'intento che si usa il C (o perlomeno si può pensare così) Dal punto di vista del C, il codice C sarà visibile, mentre quello del C++ sarà invisibile.	È solo per C++ Non funzionerà con il C.  Esprime l'intento che si usi C++ (o perlomeno si può pensare così)

---

## Namespaces

---

**Definizione:** Gli **namespaces** ci permettono di raggruppare varie entità che altrimenti si troverebbero nello scope globale. Permettono una migliore organizzazione e strutturazione del codice.

Se avessimo per esempio due funzioni con lo stesso nome, sarebbe difficile differenziarle e quindi i namespaces ci permettono di separarle.

Per creare una namespace adoperiamo la keyword **namespace**.

```
namespace primo_spazio {
    void func()
    {
        std::cout << "Dentro al namespace:
        primo_spazio" << std::endl;
    }
}

namespace secondo_spazio {
    void func()
    {
        std::cout << "Dentro al namespace:
        secondo_spazio" << std::endl;
    }
}

int main()
{
    // Chiamo la funzione func del primo spazio.
    primo_spazio::func();
    // Output: Dentro al namespace: primo_spazio

    // Chiamo la funzione func del secondo spazio.
    secondo_spazio::func();
    // Output: Dentro al namespace: secondo_spazio
    return 0;
}
```

Il namespace più usato è quello della libreria Standard del linguaggio, ovvero il namespace `std` che raggruppa tutte le funzioni e classi della libreria Standard.

Ogni qualvolta che usiamo una funzione, classe della libreria Standard ci riferiamo a quel namespace. Usiamo il nome del namespace e i due punti `::` per indicare che quello che stiamo usando fa parte di quel namespace. C'è un modo, però per evitare ogni volta di scrivere `std::`, ed è attraverso la riga **using namespace std**;. Con questo non abbiamo più bisogno di scrivere `std::`, perché lo da già per scontato, o meglio, li prende direttamente dalla libreria Standard.

```
// Accediamo al namespace std.
std::string s = "Hello World!";

// Qui invece facciamo la stessa cosa, ma senza dover riscrivere
// ogni volta std::
using namespace std;

string s = "Hello World!";
```

### *std:: vs using namespace std*

Usare **using namespace std** è considerato una **bad practice**, probabilmente ci sono diversi motivi per questo, ma qui ne elenco alcuni:

- Come abbiamo detto prima, se noi per esempio abbiamo due namespaces con due funzioni con lo stesso nome, se noi usiamo *using namespace nome\_del\_namespace* allora avremmo un conflitto, o meglio, avremmo due namespaces con una funzione con lo stesso nome, il che creerebbe confusione. (e questo non vale solo per le funzioni, ma anche per le classi, costanti, ecc..). Il programma ancora compilerebbe, ma potrebbe chiamare la funzione sbagliata.
- Usare *using namespace std* importerebbe nel nostro programma l'intero namespace `std` anche quando a noi serve solo una parte del namespace. **Non** è un problema di performance, ma solo di chiarezza del codice e di evitare ambiguità.
- Scrivere invece **std::** ogni volta rende chiaro il codice, perchè si capisce subito da quale namespace stai prendendo quella data funzione e/o altro.

Quindi, per rendere il codice più chiaro è meglio usare **std::** al posto del **using namespace std**;

```

// Se rimostrassimo il codice di prima

namespace primo_spazio {
    void func()
    {
        std::cout << "Dentro al namespace:
        primo_spazio" << std::endl;
    }
}

namespace secondo_spazio {
    void func()
    {
        std::cout << "Dentro al namespace:
        secondo_spazio" << std::endl;
    }
}

using namespace primo_spazio;
using namespace secondo_spazio;

int main()
{
    // Il codice diventa ambiguo!
    func();
    func();
    return 0;
}

```

Un modo per evitare questa ambiguità sarebbe usando la keyword `typedef` che permette essenzialmente di rinominare una keyword.

```

// Questo eviterebbe in parte l'ambiguità, ma comunque rimane
// meglio mettere nome_del_namespace::funzione.
typedef primo_spazio::func() primo_func();
typedef secondo_spazio::func() secondo_func();

int main()
{
    // Chiamo la funzione func del primo spazio.
    primo_func();
    // Output: Dentro al namespace: primo_spazio
}

```

```
    // Chiamo la funzione func del secondo spazio.  
    secondo_func();  
    // Output: Dentro al namespace: secondo_spazio  
    return 0;  
}
```

Al posto di importare l'intero namespace `std`, si potrebbe troncare e portare solo una parte del namespace `std`.

```
using std::cout;  
  
std::string s = "Hello World!"  
  
cout << s << std::endl;
```

Comunque, in generale è meglio usare `std::`.

### ***Mai mettere using namespace in un header file!***

Un altro importante problema che può capitare con `using namespace std` è quello di includerlo in un header file. È DA NON FARE!

Mettere lo `using namespace` in un header file costringe chiunque voglia utilizzare la tua libreria ad usare anche `using namespace`, il che crea un problema quando per esempio l'utente crea una funzione che si trova anche nel namespace.

```
using namespace std; // NON mettere lo using namespace std qui, in un header file, MAI metterlo  
// Perché poi quando l'header file viene incluso in altri file, avremo "inquinato" gli altri file  
// Non sappiamo chi potrebbe usare quel codice e in che modo  
// NON mettere neppure qualcosa come using std::chrono::steady_clock perché anche questo inquinerebbe gli altri file  
// QUINDI NON mettere mai using ... negli header file (.h)
```

Figura 1.6: Mai mettere using in un header file

Quindi, meglio mettere `using namespace` nei files `.cpp`, ma in generale, come abbiamo detto prima è meglio non utilizzarli.

Quindi non mettere `using namespace` in un header file, neanche `using namespace std`.

---

## Strutture

---

**Definizione:** Le strutture sono dei tipi di dati definiti dall'utente per raggruppare oggetti di tipi diversi.

Sono usati per rappresentare un record.

La keyword **struct** è usata per creare una struttura.

In questo modo semplicemente creiamo la struttura, ma non instanziamo nessun oggetto, per creare una istanza servirà richiamare il nome della struttura e poi il nome dell'istanza.

Per accedere ai campi della struttura si potrà usare l'operatore **.** (punto).

```
struct nome_struttura {  
    // campi della struttura  
    int x;  
    double d;  
    char stringa[128];  
}; // Da notare il ; alla fine
```

// Esempio una struttura per immagazzinare le coordinate di un punto.

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

// In realtà in C++ non è necessario usare la keyword struct per creare un'istanza, a differenza del C.

```
struct Point p1;  
p1.x = 0;  
p1.y = 1;
```

In realtà in C++ non è necessario usare la keyword struct per creare un'istanza, a differenza del C.

Inoltre è anche possibile assegnare dei valori di default ai campi della struttura.

```
struct Point {  
    int x = 0;  
    int y = 1;
```



```
};

struct Point p1;
std::cout << p1.x << std::endl; // Output : 0
std::cout << p1.y << std::endl; // Output : 1
```

## *typedef*

**Definizione:** La keyword **typedef** è usata essenzialmente per rinominare una tipologia.

Possiamo usare la parola chiave **typedef** per evitare di scrivere ogni volta **struct nome\_della\_struttura** per istanziare:

```
typedef struct Point Punto;
// Ora possiamo semplicemente scrivere Punto
// nome_della_istanza per creare una nuova istanza, al posto di
// dover scrivere struct Point nome_della_istanza.
// In questo caso, potrebbe non sembrare molto, ma per strutture
// con nomi più lunghi è una manna dal cielo.
```

```
Punto p1;
p1.x = 3;
p1.y = 2;
```

Inoltre, ci sono diversi modi per creare una struttura apparte il modo visto prima, due di questi è attraverso il **typedef**:

```
// Altro modo 1
typedef struct Point {
    int x;
    int y;
} Point;
```

```
Point p1;
p1.x = 1;
p1.y = 0;
```

```
// Altro modo 2
struct Point {
    int x;
    int y;
} typedef Point;
```

```
Point p1;  
p1.x = 4;  
p1.y = 5;
```

## *Funzioni nelle strutture*

A differenza del C, nelle strutture del C++ è possibile inserire delle funzioni.

```
struct Rettangolo {  
    int x;  
    int y;  
    int area() {  
        return x * y;  
    }  
};  
  
int main()  
{  
    typedef struct Rettangolo Rettangolo;  
    Rettangolo r1;  
    r1.x = 3;  
    r1.y = 2;  
    std::cout << "Area rettangolo: " << r1.area() << std::endl; // Output: Area rettangolo: 6  
    return 0;  
}
```

## *Strutture nelle strutture*

È possibile includere delle strutture all'interno di una struttura, come una matrisca.

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

// Ovviamente la definizione della struttura Point deve essere fatta prima della struttura Rettangolo se la vogliamo includere in Rettangolo.

```

struct Rettangolo{
    Point p;
    int area(){
        return p.x * p.y;
    }
};

typedef struct Rettangolo Rettangolo;
Rettangolo r1;
r1.p.x = 3;
r1.p.y = 2;
std::cout << "Area rettangolo: " << r1.area() <<
    std::endl; // Output: Area rettangolo: 6

```

### ***Puntatore ad una struttura***

È possibile far puntare un puntatore ad una struttura.

Per assegnare un valore ad uno specifico campo della struttura possiamo sia avvalerci dell'operatore . sia dell'operatore -> che in questo caso fa esattamente la stessa cosa.

```

struct Book {
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
};

typedef struct Book Book;
Book *pBook;

// Possiamo sia fare così..
(*pBook).title = "Learn C++";

//.. sia fare così
pBook->title = "Learn C++";

```

## *Array di Strutture*

Ovviamente è possibile creare un array di strutture, dove ogni elemento dell'array è una struttura.

```
struct Cliente {
    int id;
    char nome[128];
};

typedef struct Cliente Cliente;

Cliente clienti[2] = {{0, "Gigi"}, {1, "Pippo"}};
// Oppure
clienti[0].id = 0;
clienti[0].nome = "Gigi";

clienti[1].id = 1;
clienti[1].nome = "Pippo";

// Oppure si potrebbe anche fare così
struct Cliente {
    int id;
    char nome[128];
} cliente1, cliente2;

cliente1.id = 0;
cliente1.nome = "Gigi";

cliente2.id = 1;
cliente2.nome = "Pippo";
```

## *Strutture come parametri e come ritorno*

Ovviamente, si possono passare anche le strutture come parametri. Da fare attenzione che se non serve, meglio non copiare un'intera struttura quando la si passa come parametro.

```
struct Book {
    char title[50];
    char author[50];
    char subject[100];
};
```

```

    int    book_id;
};

void stampaLibro(struct Book* book){
    std::cout << "Titolo: " << book->title << std
        ::endl;
    std::cout << "Autore: " << book->author <<
        std::endl;
    std::cout << "Soggetto: " << book->subject <<
        std::endl;
    std::cout << "Id: " << book->book_id << std::
        endl;
}

struct Book book1;
std::strcpy( book1.title , "Learn C++ Programming"
    );
std::strcpy( book1.author , "Chand Miyan");
std::strcpy( book1.subject , "C++ Programming");
book1.book_id = 3;

stampa(&book1);
// Output : Titolo: Learn C++ Programming
// Output : Autore: Chand Miyan
// Output : Soggetto: C++ Programming
// Output : Id: 3

```

Al tempo stesso, si possono restituire strutture dalle funzioni.

```

#define MATERIE 3

struct studente{
    int matricola;
    char nome[128];
    char cognome[128];
    int voti[MATERIE];
    int media;
};

typedef struct studente Studente;

// std::cin serve per l'input dell'utente.

```

```

Studente createStudente() {
    Studente s;
    std::cout << "Inserisci matricola: \n";
    std::cin >> s.matricola;

    std::cout << "Inserisci nome: \n";
    std::cin >> s.nome;

    std::cout << "Inserisci cognome: \n";
    std::cin >> s.cognome;

    int sum = 0;

    for(int i = 0; i < MATERIE; i++){
        std::cout << "Inserisci voto: \n";
        std::cin >> s.voti[i];
        sum += s.voti[i];
    }

    s.media = sum / MATERIE;

    return s;
}

int main()
{
    Studente s = createStudente();
    // Output: quelli inseriti
    std::cout << "Nome: " << s.nome << std::endl;
    std::cout << "Cognome: " << s.cognome << std::endl;
    std::cout << "Voto1: " << s.voti[0] << std::endl;
    std::cout << "Voto2: " << s.voti[1] << std::endl;
    std::cout << "Voto3: " << s.voti[2] << std::endl;
    std::cout << "Media: " << s.media << std::endl;
    return 0;
}

```

}

### *Strutture in C vs in C++*

Strutture in C	Strutture in C++
Sono permessi solo membri dati, non funzioni.	Sono permessi sia dati sia funzioni membri.
Non può avere membri statici.	Può avere membri statici.
Non possiamo avere un costruttore.	Possiamo avere un costruttore.
L'inizializzazione diretta dei membri non è possibile.	L'inizializzazione diretta dei membri è possibile.
È necessario usare la keyword struct per dichiarare una variabile di tipo struct.	Non è necessario usare la keyword struct.
Non supporta access modifiers.	Supporta gli access modifiers. (public, private, protected, ecc..)
Sono permessi soltanto i puntatori alle strutture.	Sono permessi sia i puntatori sia le references.
L'operatore sizeof() genererà 0 per una struttura vuota.	L'operatore sizeof() genererà 1 per una struttura vuota.
Il Data Hiding non è possibile.	Il Data Hiding è possibile.

---

## *Union*

---

**Definizione:** La **union** è un tipo di struttura dove l'ammontare di memoria è una fattore chiave.

- Come le strutture, le union possono contenere diversi tipologie di variabili.
- Ogni qualvolta che una nuova variabile è inizializzata dall'union in C sovrascrive quella vecchia, ma in C++ usiamo quella locazione di memoria e non abbiamo bisogno di quella parola chiave.
- È utile quando i dati passati ad una funzione sono sconosciuti, utilizzare una **union** che contiene tutti i possibili tipi può essere il rimedio a questo problema.
- Utilizziamo la keyword **union** per crearne una.

```
union nome_della_union {  
    // tipi di dati  
}; // Da notare il ; proprio come nelle strutture.  
  
union var {  
    int iVar;  
    char cVar;  
    float fVar;  
};  
  
int main()  
{  
    // In C++ non serve la keyword union.  
    union var V1, V2, V3;  
  
    V1.iVar = 33;  
    V2.cVar = 33;  
    V3.fVar = 33.33;  
  
    std::cout << "V1 var: " << V1.iVar << std::  
        endl; // Output: V1 var: 33  
    std::cout << "V2 var: " << V2.cVar << std::  
        endl; // Output: V2 var: !  
    std::cout << "V3 var: " << V3.fVar << std::  
        endl; // Output: V3 var: 33.33  
  
    return 0;  
}
```



***structure vs union***

<b>Structure</b>	<b>Union</b>
Usa la keyword <b>struct</b>	Usa la keyword <b>union</b>
Quando una variabile è associata con una struttura il compilatore alloca la memoria per ogni membro. Lo spazio occupato dalla struttura è maggiore o uguale alla somma dello spazio dei suoi membri.	Quando una variabile è associata con una union il compilatore alloca memoria considerando lo spazio occupato dal membro più grande.
Per ogni membro della struttura è assegnato uno spazio di allocazione unico.	La memoria allocata è condivisa con i membri individuali dalla union.
Modificare un membro della struttura non modificherà gli altri membri.	Modificare un membro della union modificherà gli altri membri.
I membri individuali possono essere acceduti ad ogni momento.	Solo un membro alla volta può essere acceduto.
Si possono inizializzare diversi membri alla volta.	Solo il primo membro della union può essere inizializzato.

---

## Classi

---

**Definizione:** La classe è il concetto fondamentale, il muro portante, la pietra miliare della programmazione ad oggetti. È un tipo di dato definito dall'utente che contiene i propri membri dati e membri funzioni che possono essere acceduti creando un'istanza.

Una classe è come uno stampino, un modello per creare oggetti.

È la differenza sostanziale del C++ con il C, l'avere le classi rendendo il linguaggio: un linguaggio a programmazione di oggetti.

Ogni classe rappresenta un oggetto che possiede degli attributi, delle caratteristiche (dati) e dei comportamenti stabiliti dalle funzioni che possiede.

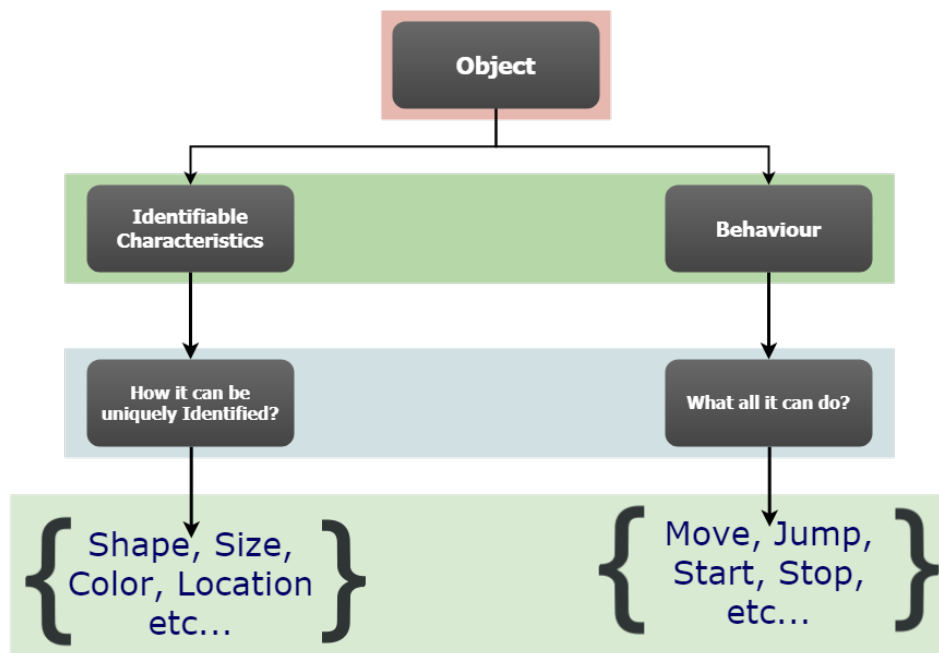


Figura 1.7: Dati e comportamenti di una classe

Per creare una classe si utilizza la keyword **class**. Lo spazio di memoria non è allocata quando la classe viene definita, ma quando viene istanziata.

Per creare un'istanza della classe, si chiama il nome della classe e poi il nome dell'istanza.

## Costruttori e Distruttori

### Costruttori

**Definizione:** Il **costruttore** è una speciale funzione membro (della classe) che inizializza gli oggetti di una classe. Il costruttore è chiamato automaticamente quando un'istanza della classe viene creata. È una funzione speciale perché non tipi di ritorno, o meglio il tipo di ritorno è la classe stessa.

Il nome di questa funzione **costruttore** è identico al nome della classe stessa.

Se non specifichiamo un **costruttore**, uno di default verrà creato dal compilatore (senza parametri e con il corpo della funzione vuoto).

### Initialization

### List

**Definizione:** La **Initialization List** è usata per inizializzare i dati membri della classe. Per fare questo aggiungiamo un **:** (due punti) dopo il costruttore e inizializziamo le variabili e le separiamo da delle virgole.

Può essere utile per:

- Chiamare il costruttore della classe base.
- Inizializzare i membri prima che il costruttore venga eseguito.
- Per inizializzare i membri **const** non statici.
- Per inizializzare dei membri referenze
- Per inizializzare gli oggetti membri che non hanno un default constructor.
- Per inizializzare i membri della classe base.
- Quando il nome del costruttore è lo stesso del dato.
- Per questioni di performance.

// Questo codice non andrebbe, perché mVal è const. Non possiamo cambiare il valore di una const nel costruttore perché è segnato come const.

```
class Demo {  
    // Costruttore  
    Demo(int& val)  
    {  
        mVal = val;  
    }  
};
```

```

    }

    const int& mVal;
};

// Questo invece è possibile:
class Demo {
    // Quindi puoi usare la lista di inizializzazione per fare
    // questo.
    // Costruttore : initialization list
    Demo(int& val) : mVal(val)
    {
    }

    const int& mVal;
};

```

## ***Distruttori***

**Definizione:** Il **distruttore**, come dice la parola, è una funzione membro della classe che viene invocata automaticamente quando un oggetto (istanza della classe) viene distrutto/eliminato. Il che significa che il distruttore è l'ultima funzione ad essere chiamata.

Per definire un **distruttore** si crea una funzione con lo stesso nome della classe, ma prima del nome deve essere accompagnata dal simbolo `~` (tilde).

## ***Proprietà del distruttore***

- Il distruttore è invocato automaticamente quando gli oggetti sono distrutti.
- Non può essere dichiarato **static** o **const**.
- Il **distruttore** non ha argomenti.
- Non ha tipi di ritorno, nemmeno **void**.
- Un oggetto della classe con un distruttore non può diventare membro di una **union**.
- Un distruttore dovrebbe essere dichiarato nella sezione **public**.
- Il programmatore non può accedere all'indirizzo del **distruttore**.

***Quando viene chiamato il distruttore?***

- La funzione finisce.
- Il programma termina.
- Un blocco contenente le variabili cessa.
- Un operatore **delete** viene chiamato.

```
class MyClass {
    // Costruttore
    MyClass () {
        // Corpo del costruttore.
    }
    // Distruttore
    ~MyClass () {
        // Corpo del distruttore.
    }
}
```

***Access modifiers***

**Definizione:** Gli **Access Modifiers** in una classe sono usati per assegnare l'accessibilità ai membri della classe. Questo permette una importante feature della programmazione ad oggetti, ovvero la **Data Hiding** che permette di prevenire l'accesso diretto dei dati da parte delle funzioni del programma.

Ci sono 3 tipi di **access modifiers**:

Access Modifier	Definizione
<b>public</b>	accessibile a tutti.
<b>private</b>	accessibile solo all'interno della classe stessa.
<b>protected</b>	accessibile solo alla classe, alle sue sottoclassi (ereditarietà) ed alle classi amiche (friend class).

***Incapsulamento***

**Definizione:** L'**incapsulamento** è un concetto di programmazione ad oggetti che mette assieme i dati e le funzioni che manipolano i dati per mantenerli sicuri da interferenze esterne e da un uso improprio.

L'**incapsulamento** dei dati è un meccanismo di impacchettamento di dati e delle funzioni che li usano.

La **Data abstraction** è un meccanismo che espone solo le interfacce e nasconde i dettagli dell'implementazione dall'utente.

```
class Sommatore {  
    // con public sono accessibili da tutti.  
public:  
    // Costruttore.  
    Sommatore(int i = 0) { // i = 0 vuol dire che  
        // assegniamo un valore di default, casomai l'utente non  
        // voglia inserirne uno.  
        totale = i;  
    }  
  
    // Interfaccia al mondo esterno.  
    void aggiungiNumero(int numero) {  
        totale += numero;  
    }  
  
    // Interfaccia al mondo esterno.  
    int getTotale() {  
        return totale;  
    }  
  
private:  
    // Dati nascosti al mondo esterno.  
    int totale;  
};  
  
int main() {  
    Sommatore s;  
  
    s.aggiungiNumero(3);  
    s.aggiungiNumero(6);  
    s.aggiungiNumero(9);  
  
    std::cout << "Totale: " << s.getTotale() <<  
        std::endl; // Output: Totale: 18  
    return 0;  
}
```

### *scope resolution operator ::*

L'operatore **scope resolution** indicato con `i ::` (doppi due punti) può essere usato per definire delle funzioni della classe fuori dalla stessa.

Può essere usato per accedere ad una variabile globale quando c'è anche una variabile locale con lo stesso nome.

Può essere usato quando si ha la definizione di una classe all'interno di un'altra classe.

```
#include <iostream>
int weight = 33;

class MyClass {
public:
    MyClass() {
        num = 66;
    }

    void display();

    int get_num() {
        return num;
    }

private:
    int num;
};

void MyClass::display() {
    std::cout << "Il valore di num e\': " << get_num
        () << std::endl;
}

int main() {
    int weight = 99;
    MyClass istanza;
    istanza.display(); // Output: Il valore di num è: 66

    std::cout << "Valore della variabile weight
        locale: " << weight << std::endl;
    std::cout << "Valore della variabile globale: "
```

```

        << ::weight << std::endl;
    return 0;
}

```

## Getters & Setters

Per via dell'incapsulamento, per poter recuperare (getter) o impostare (settare) le variabili private usufruiamo dei **getters & setters** che sono due funzioni, una per recuperare il dato (getter) e l'altro per impostarlo (setter).

```

class MyClass {
public:
    MyClass() {
        // Costruttore
    }

    // Recupera il valore della variabile number.
    int get_number() {
        return number;
    }

    // Imposta un nuovo valore alla variabile number.
    void set_number(int number_t) {
        number = number_t;
    }

private:
    int number;
};

```

## Ereditarietà

**Definizione:** L'ereditarietà è la capacità di derivare le proprietà e le caratteristiche da un'altra classe. È una delle feature più importanti della programmazione ad oggetti.

La classe che deriva è chiamata **derived class** o **sub class**, mentre quella che viene derivata è chiamata **base class** o **super class**.

Per implementare l'ereditarietà usiamo l'operatore `:` quando andiamo a definire la classe derivata. Questa è chiamata la **initialization list** serve per chiamare



la classe base e per inizializzare le variabili membri prima che il costruttore venga eseguito.

```
class nome_classe_derivata : modalità_di_accesso
    nome_classe_base {
    // Corpo della subclass/ derived class
}
```

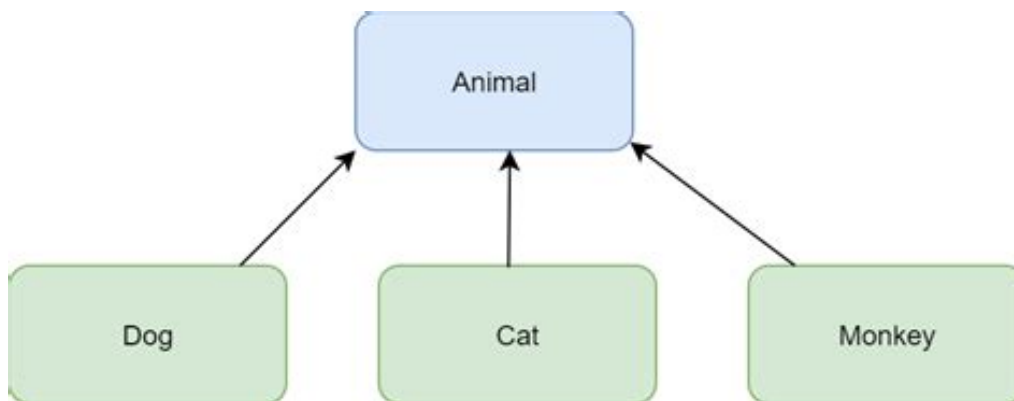


Figura 1.8: Concetto dell'ereditarietà

```
class Animal {
public:
    Animal() {
        std::cout << "Animal Constructor" << std::endl;
    }

    void eat() {
        std::cout << "gnam gnam.." << std::endl;
    }

    void sleep() {
        std::cout << "Sleeping zzz.." << std::endl;
    }
};

class Dog : public Animal {
public:
    Dog(std::string name, int weight){
```

```

std::cout << "Dog Constructor" << std::
    endl;
// Qui stiamo assegnando i valori dei parametri alle
// nostre variabili nella classe (quelle in private).
// Per evitare confusioni potremmo anche chiamare i
// parametri del costruttore in maniera diversa (tipo:
// nomeparametro_t per differenziarlo oppure
// _nomeparametro) oppure per differenziare le
// variabili della classe potremmo aggiungerci la
// keyword this.
name = name;
weight = weight;
}

void bark() {
    std::cout << "Wuuf Wuuf" << std::endl;
}

std::string get_name() {
    return name;
}

int get_weight() {
    return weight;
}

private:
    std::string name;
    int weight;
};

class Cat : public Animal {
public:
    Cat(std::string name, int weight) {
        std::cout << "Cat Constructor" << std::
            endl;
        // Qui usiamo il puntatore this per far riferimento alle
        // variabili membre della classe al posto di quelle
        // passate come parametro al costruttore.
        this->name = name;
        this->weight = weight;
    }
};

```

```

    }

    void meow() {
        std::cout << "Meow Meow" << std::endl;
    }

    std::string get_name() {
        return name;
    }

    int get_weight() {
        return weight;
    }

private:
    std::string name;
    int weight;
};

int main() {
    Dog floki { "floki", 36 };
    std::cout << floki.bark() << std::endl;
    std::cout << floki.get_name() << std::endl;
    std::cout << floki.get_weight() << std::endl;

    // Output: Animal Constructor
    // Output: Dog Constructor
    // Output: floki
    // Output: 36
    return 0;
}

```

***this******pointer***

**Definizione:** La keyword `this` serve per riferirsi all'oggetto in cui ci troviamo.

## ***Multi-Ereditarietà***

Il c++ permette l'**ereditarietà multipla**, quindi una classe può derivare da più classi base. Non è presente invece l'implementazione di interfacce.

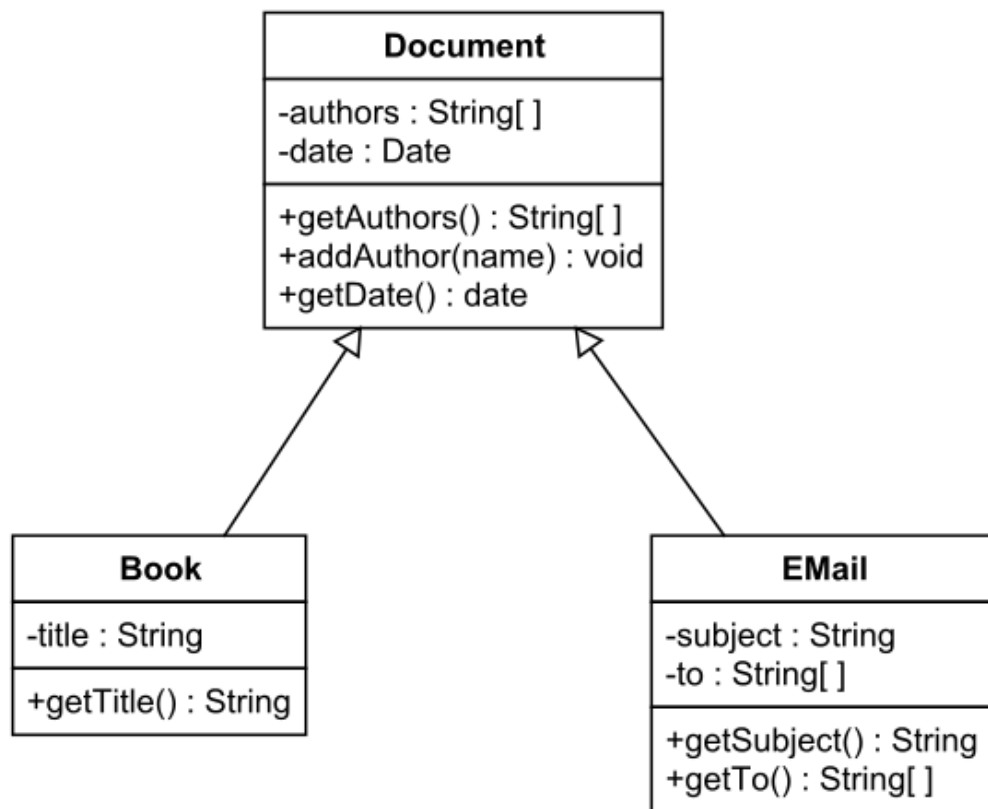


Figura 1.9: Ereditarietà in un diagramma UML

```
class A
{
    public:
    A() { cout << "A's constructor called" <<
        endl; }
};

class B
{
    public:
    B() { cout << "B's constructor called" <<
        endl; }
};

class C: public B, public A // Da notare l'ordine.
{
```

```

    public:
    C() { cout << "C's constructor called" <<
        endl; }
};

int main()
{
    C c;
    // Output: B's constructor called
    // Output: A's constructor called
    // Output: C's constructor called
    return 0;
}

```

## Forward Declaration

**Definizione:** La **forward declaration** è quando prima dichiariamo una funzione, una classe, eccetera.. con la premessa che da qualche parte nel codice più in là ci sarà una definizione di questa funzione, classe, eccetera..

Può essere utile per aiutare il compilatore per assicurarsi che non ci sono stati errori di spelling o di numero sbagliato di argomenti da passare.

Può essere utile per ridurre il tempo di *build* del programma.

Può essere utile per rompere il ciclo delle referenze dove due definizioni si usano a vicenda.

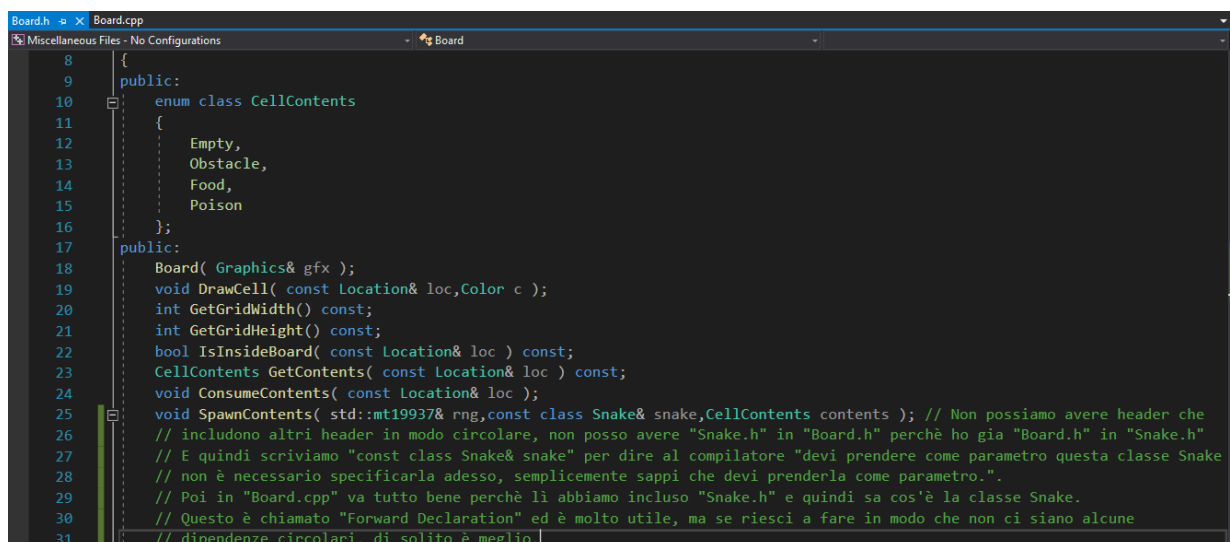


Figura 1.10: Forward Declaration

## *Chiamata a funzione statica e a membro*

*Static*

*nelle*

*Classi*

**Definizione:** Possiamo definire un membro della classe statica attraverso la keyword **static**. Questo significa che non importa quante istanze della classe vengano create, c'è una sola copia del membro statico.

Un membro statico è condiviso da tutti gli oggetti della classe. Se non è presente un'inizializzazione al membro statico, il suo valore di default sarà 0.

Per accedere a questa funzione statica o membro o altro **non** possiamo utilizzare l'operatore `.`, ma dobbiamo usufruire dell'operatore `::`.

```
class MyClass {  
    public:  
        MyClass () {  
            // Costruttore  
        }  
  
        // Questo è un esempio, non ho messo l'implementazione  
        della funzione.  
        static int calcola_qualcosa () {}  
};  
  
int main () {  
    MyClass oggetto;  
    // Non posso fare oggetto.calcolaQualcosa();  
    // Devo fare MyClass::calcola_qualcosa();  
    MyClass::calcola_qualcosa ();  
    return 0;  
}
```

```

if (left < 0)
{
    x_mobile = 5;
}
else if (right >= Graphics::ScreenWidth)
{
    x_mobile = gfx.ScreenWidth - 6; // Avrei potuto scrivere Graphics::ScreenWidth ; Graphics è il nome della classe e
    // ScreenWidth è una costante intera statica, ed è proprio perchè è statica che posso usare la notazione
    // Graphics::ScreenWidth
    // Guarda sopra nell'if
}

if (top < 0)
{
    y_mobile = 5;
}
else if (bottom >= gfx.ScreenHeight)
{
    // ...
}

```

Graphics.h

```

75 static constexpr int ScreenWidth = 800;
76 static constexpr int ScreenHeight = 600;
77 };

```

Figura 1.11: Chiamata a membro statico

## Funzioni e la keyword *const*

Ci sono vari significati che la keyword **const** assume e fa assumere alla funzione quando si trova in essa.

Mettendo **const** nei parametri della funzione, ciò significa che i parametri di quella funzione non possono essere cambiati, perché sono costanti.

```

void Poo::ProcessConsumption(const Dude& dude) // Meglio passarlo per referenza, per evitare di copiarlo e occupare spazio
// Qui passiamo dude non per cambiare i suoi valori, ma solo per leggerli e quindi possiamo mettere const così non possiamo
// neanche accidentalmente modificare i valori.
const int duderight = dude.x + dude.width;
const int dudebottom = dude.y + dude.height;
const int pooright = x + width;
const int poobottom = y + height;

if (duderight >= x &&
    dude.x <= pooright &&
    dudebottom >= y &&
    dude.y <= poobottom)
{
    isEaten = true;
}

```

Figura 1.12: Const come parametro

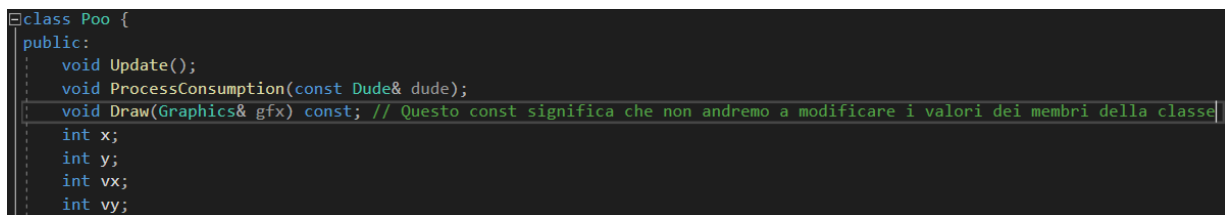
Mentre, la keyword **const** alla fine della funzione (Const member function in inglese) significa che l'oggetto chiamato da questa funzione non può essere modificato, questo previene modifiche accidentali all'oggetto.

```
int val = 5;
```

```

// Se aggiungessimo una riga per modificare il valore, otterremmo
// un errore.
// Inoltre mettere quel const lì esprime l'intento di non cambiare
// l'oggetto
// della funzione.
int getValue() const
{
    return val;
}

```



```

class Poo {
public:
    void Update();
    void ProcessConsumption(const Dude& dude);
    void Draw(Graphics& gfx) const; // Questo const significa che non andremo a modificare i valori dei membri della classe
    int x;
    int y;
    int vx;
    int vy;
}

```

Figura 1.13: Const function class members

Infine, c'è restituire **const** come valore di ritorno di una funzione, ma non sembra di molta utilità, tranne per le move-semantics, per lo meno se lo si ritorna per valore, mentre ritornarlo per reference protegge il valore di ritorno dall'essere modificato.

## *Class vs Struct*

In C++ le classi e le strutture sono simili, ma con alcune differenze:

Class	Struct
I membri della classe sono privati da default.	I membri di una struttura sono pubblici da default.
L'allocazione della memoria avviene nell'heap.	L'allocazione della memoria avviene sullo stack.
È un tipo di dato per reference.	È un tipo di dato per valore.
Si dichiara usando la keyword <b>class</b> .	Si dichiara usando la keyword <b>struct</b> .



---

## Convenzioni del linguaggio

---

**Definizione:** Le **convenzioni** sono delle linee guida di un linguaggio che raccomandano un certo stile di programmazione. Queste permettono un codice più chiaro, più leggibile e rende il codice di un software più semplice da mantenere.

Inoltre, sia il codice che i commenti dovrebbero essere in inglese a differenza di come ho fatto io in questa guida.

Potete trovare tutte le convenzioni del linguaggio nelle **C++ Core Guidelines** : [CppCoreGuidelines](#)

Qui anche una versione più corta (non ufficiale): [CppStyleAndConventions](#)

Comunque ne elencherò qualche d'una:

### Generale

- Lunghezza della riga limitata a 80 caratteri
- Indentazione con 4 spazi.
- I files dovrebbero usare le newlines stile Unix `\n`.

### Parentesi Graffe

Utilizzare *Allman Style* brackets (parentesi). Le parentesi graffe sono sulla loro linea allo stesso livello con lo statement sopra.

```
if ( condition )  
{  
  
}  

```

Anche gli if con una sola linea dovrebbero avere le parentesi graffe.

### Indentazione

Il contenuto in un **namespace** dovrebbe essere allo stesso livello di indentazione del namespace stesso.

Il contenuto in una **classe**, **struct**, **funzioni**, **if**, **loop**, **switch**, **cases** e **labels** (dei **goto**) dovrebbe essere indentato.

## *Convenzioni sui nomi*

- Non c'è nè prefisso nè suffisso a nessun nome.
- Gli acronimi dovrebbero essere dello stesso case.

*// Correct.*

```
SomeBMCType someBMCTVariable = bmcFunction();
```

## *Ordine Inclusione Header files*

Inclusione degli headers in un header file:

- headers locali
- librerie c
- librerie cpp

Inclusione degli headers in un source file (.cpp):

- source.hpp (se applicabile)
- headers locali
- librerie c
- librerie cpp

In ordine alfabetico.

## *Files*

- Gli headers C++ dovrebbero finire in *.hpp*. Gli headers C dovrebbero finire in *.h*.
- I files dovrebbero essere chiamati nel modo (case) *lower\_snake\_case*.

## *Types*

- Preferire *using* a *typedef*.
- Le strutture, classi, enums dovrebbero essere tutti in UpperCamelCase.
- Preferire gli scope namespaces al posto di nomi con lunghi prefissi.
- Un alias di una singola parola con una struct / class dovrebbe essere in minuscolo, ma un alias a più parole dovrebbe essere UpperCamelCase.
- Eccezioni: Una libreria API potrebbe usare il modo `lower_snake_case` per accordarsi alle convenzioni STL o ad una libreria C. Application APIs dovrebbero tutte essere UpperCamelCase.
- Eccezione: Per convenienza un tipo di una classe template potrebbe finire in `_t` per accordarsi alle convenzioni STL.

## *Variabili*

Le variabili dovrebbero tutte essere lowerCamelCase, inclusi i membri delle classi, senza underscore (trattini bassi).

## *Funzioni*

- Le funzioni dovrebbero essere lowerCamelCase.
- Eccezione: Una libreria API potrebbe usare `lower_snake_case` in accordo con le convenzioni STL o di una sottostante libreria in C che sta astraendo. Application API dovrebbero tutte essere lowerCamelCase.

## *Costanti*

- Costanti e i membri delle enums dovrebbero essere chiamati come le variabili in lowerCamelCase.

## *Namespaces*

- I namespaces dovrebbero essere `lower_snake_case`.
- Top-level namespace dovrebbe essere chiamato sulla base della repository che lo contiene.

- Favorisci un namespace chiamato 'details' o 'internal' per indicare l'equivalente di un namespace 'private' in un header file e namespaces anonimi in un file C++.

## Header Guards

Preferire `#pragma` allo stile `#ifndef`.

## Spazi bianchi aggiuntionali

- Segui lo stile di dichiarazione del C++

```
foo(T& bar, const S* baz); // Correct.
foo(T &bar, const S *baz); // Incorrect.
```

- Usa gli spazi bianchi moderatamente.
- Inserisci uno spazio bianco prima e dopo if e loops.

```
if (...)
while (...)
for (...)
```

- Aggiungi spazio bianco attorno agli operatori binari per leggibilità

```
foo((a-1)/b,c-2); /// Incorrect.
foo((a - 1) / b, c - 2); /// Correct.
```

- Non inserire spazi bianchi dopo gli operatori unari.

```
a = * b;    /// Incorrect.
a = & b;    /// Incorrect.
a = b -> c;  /// Incorrect.
if (! a)    /// Incorrect.
```

- Non inserire spazi bianchi nè prima nè dopo una chiamata a funzione e ai parametri.

```
foo(x, y); /// Correct.  
foo ( x , y ); /// Incorrect.
```

```
do (...) // Correct.  
{  
} while(0); /// 'while' qui è strutturato come una chiamata a  
funzione.
```

- Preferire una linea a capo dopo gli operatori per mostrare la continuazione.

```
if (this1 == that1 &&  
this2 == that2) /// Correct.
```

```
if (this1 == that1  
&& this2 == that2) /// Incorrect.
```

- Le linee lunghe dovrebbero avere la continuazione che inizi allo stesso livello delle parentesi o tutti gli oggetti all'interno delle parentesi dovrebbero essere al secondo livello di indentazione.

```
reallyLongFunctionCall(foo ,  
bar ,  
baz); // Correct.
```

```
reallyLongFunctionCall(  
foo ,  
bar ,  
baz); // Also correct.
```

```
reallyLongFunctionCall(  
foo , bar , baz); // Similarly correct.
```

```
reallyLongFunctionCall(foo ,  
bar ,  
baz); // Incorrect.
```

### *Linee Guida Miste*

- Usare sempre `size_t` o `ssize_t` per cose come contatori, pesi, ecc.. C'è bisogno di un forte motivo razionale per usare un tipo come `uint8_t` quando `size_t` può fare lo stesso lavoro.
- Usa `uint8_t`, `int16_t`, `uint32_t`, `int64_t` quando è importante per l'interazione coll'hardware. Non usarli senza una buona motivazione, quando le interazioni coll'hardware non sono coinvolte; preferire `size_t` o `int` piuttosto.

# Concetti Intermedi

# 2

---

## *Introduzione*

---

In questo capitolo, tratterò argomenti non necessariamente più complicati, ma che non considererei basi.

In questo capitolo vedremo ulteriori concetti riguardo le classi, il polimorfismo, le varie tipologie di costruttori, la programmazione generale, le lambdas, la programmazione funzionale e molto altro ancora..

---

## Templates

---

Immaginiamo di avere un codice, esempio questo:

```
const int& max(const int& a, const int& b)
{
    return a > b ? a : b;
}
```

Però ora se noi volessimo utilizzare questa funzione per i double, dovremmo copiarla e cambiare la tipologia da int a double.

```
const int& max(const int& a, const int& b)
{
    return a > b ? a : b;
}

const double& max(const double& a, const double&
    b)
{
    return a > b ? a : b;
}
```

C'è un problema, se ora volessimo usare la stessa funzione, ma con i float? o con i char? Certo potremmo fare dei casts, ma così perderemmo dei dati, ma soprattutto ripeteremmo lo stesso codice più e più volte semplicemente per avere la stessa identica funzione, ma per tipologie diverse.

Inoltre, fare questo, continuare a ripetere lo stesso codice, violerebbe un'importante principio in programmazione, ovvero **DRY**: *Don't repeat yourself*, in italiano, non ripeterti.

Vogliamo cercare di ripetere lo stesso codice **il meno possibile** e cercare di **riutilizzare** codice che già abbiamo per altre funzionalità. Quindi, c'è un modo migliore? Possiamo evitare di ripetere di scrivere lo stesso codice più e più volte? Sì e Sì! E facciamo questo attraverso i **templates**!

**Definizione:** I **templates** sono la fondazione della programmazione generale che riguarda lo scrivere codice che è indipendente dalla tipologia.



Quindi un **template** ti permette di creare uno stampino che funziona con qualsiasi tipo di variabile.

Come facciamo a dire al compilatore che vogliamo usare una variabile generica? Usiamo **typename** per dire che l'identificatore che segue è una tipologia e lo mettiamo all'interno del "diamantino", ovvero <>.

```
template <tipologia> tipoDiRitorno
    nomeDellaFunzione(lista dei parametri)
{
    // corpo della funzione.
}

// Quindi usiamo una tipologia generica e la indichiamo con T, ma
// avremmo potuto usare qualsiasi altra lettera.
template<typename T>
const T& max(const T& a, const T& b)
{
    return a > b ? a : b;
}

int x = 5, y = 3;
std::cout << "Max tra due int: " << max(a, b) <<
    std::endl; // Output: Max tra due int: 5

double d1 = 3.69, d2 = 7.89;
std::cout << "Max tra due double: " << max(a, b)
    << std::endl; // Output: Max tra due double: 7.89

// Fate attenzione che se state usando 'using namespace std', avrete
// due funzioni chiamate max, una della libreria standard e l'altra
// questa in questo esempio.
// In quel caso vi conviene rinominare la vostra funzione in
// qualcos'altro o semplicemente con la m MAIUSCOLA (Max).
```

Questo, può naturalmente essere fatto anche con le classi ed altro..  
Questa è una funzionalità, come abbiamo potuto vedere in questo semplice esempio, di quanto possono essere utili i templates.

```

template<typename T> // Questo è un template, così potremmo creare una classe "DynamicArray" di qualsiasi tipologia.
class DynamicArray // Quindi mettiamo T dove invece andrebbe la tipologia della variabile.
{
    // Quando invece andremo a creare un'istanza della classe DynamicArray dovremo dichiararla nel seguente modo:
    DynamicArray(T size); // DynamicArray<int> dal ( 5 ); Nel "diamantino" (ovvero maggiore e minore) < > specifichiamo
    : // la tipologia , ovvero in questo esempio int, ma avremmo potuto anche mettere altro, tipo float, o altro..
    size ( size ), // DynamicArray<float> dal ( 5.0f); e così ne abbiamo dichiarata una (di istanza) di tipo float.
    pArray( new T[size])
    {}
private:
    T size = 0;
    T* pArray = nullptr;
};

class DynamicIntArray // Per esempio se volessimo creare questa classe per i float dovremmo copiare il codice e modificare
{
    // i nomi e le funzioni e le variabili da int a float, ma il codice rimarrebbe più o meno identico e quindi per questo
public: // ci conviene usare i template
    DynamicIntArray() = default;
    DynamicIntArray(const DynamicIntArray& source)

```

Figura 2.1: Template

---

## *std::vector<>*

---

**Definizione:** I **vectors** sono un contenitore rappresentante una array che può cambiare in size (spazio). Sono degli array dinamici.

I vectors memorizzano i dati in locazioni contigue di memoria e permettono l'accesso diretto a qualsiasi elemento usando l'operatore []. Supportano la riduzione e l'ampiamiento dello spazio a runtime (ovvero eseguite mentre il tuo programma è in esecuzione).

La classe vector fa uso dei template così che possiamo eseguirla con qualsiasi tipo. Per poterla usare avremo bisogno di importare `#include <vector>`.

```

#include <iostream>
#include <vector>

```

```

std::vector<int> v{ 1, 3, 7, 8};
std::vector<int> v2 = v; // Oppure potevamo scrivere
std::vector<int> v2(v);

```

```

v2.push_back(9); // Aggiungiamo un elemento.

```

```

std::cout << "v size: " << v.size() << std::endl;
//Output: v size: 4
std::cout << "v2 size: " << v2.size() << std::endl;
//Output: v2 size: 5

```

Inoltre, la classe *vector* mette a disposizione tante altre funzioni per la loro manipolazione.

P.S.: Da non confondere con i vettori in matematica|fisica.

---

## *Iteratori*

---

**Definizione:** Gli **iteratori** sono degli oggetti (come puntatori) che puntano ad un elemento all'interno di un contenitore. Usiamo gli **iteratori** per muoverci nel contenitore

Ci sono diversi tipi di iteratori:

- **Input Iterators** : Sono i più deboli fra tutti per via delle loro limitate funzionalità. Può essere usato solo in algoritmi single-pass ovvero quelli che processano il contenitore in modo sequenziale.
- **Output Iterators** : Anch'essi sono molto limitati. Possono essere usati negli algoritmi single-pass, ma non per accedere agli elementi, ma per essere assegnati agli elementi.
- **Forward Iterator** : Sono più in alto nella gerarchia rispetto agli input ed output e possiedono tutte le funzionalità di questi ultimi due, ma possono anche muoversi in avanti ed anch'essi di una posizione alla volta.
- **Bidirectional Iterators** : Possiedono tutte le funzionalità degli forward iterators, ma possono muoversi in entrambe le direzioni.
- **Random-Access Iterators** : Sono gli iteratori più potenti. Non sono limitati dal solo poter muoversi in modo sequenziale, ma possono accedere in maniera casuale a qualsiasi elemento dentro ad un contenitore. Sono quelli che hanno le stesse funzionalità dei puntatori.

ITERATORS	PROPERTIES				
	ACCESS	READ	WRITE	ITERATE	COMPARE
Input	->	= *i		++	==, !=
Output			*j=	++	
Forward	->	= *i	*j=	++	==, !=
Bidirectional		= *i	*j=	++, --	==, !=, <, >
Random-Access	->, []	= *i	*j=	++, --, ++, --, +, -	==, !=, <, >, <=, >=

È sempre meglio usare gli **iteratori** per iterare tra i contenuti di un contenitore così da evitare di usare l'operatore `[]` per accedere agli elementi. Inoltre per ottenere la fine di un contenitore con gli **iteratori** possiamo semplicemente usare la funzione **end()** al posto di utilizzare lo spazio occupato.

Possono essere utili per la riusabilità del codice, visto che anche se cambiamo vettore, il codice riguardante gli **iteratori** non dovrebbe cambiare.

Gli **iteratori** ci permettono una manipolazione dinamica dei contenitori, permettendoci di aggiungere e rimuovere elementi in modo dinamico a nostro piacimento.

Per poter usare gli iteratori è necessario includere `#include <iterator>`.

- **begin()** : Restituisce la posizione iniziale del contenitore.
- **end()** : Restituisce la posizione finale del contenitore.

```
#include <iostream>
#include <iterator>
```

```
std::vector<int> v = { 9, 6, 3};
```

```
// Dichiaro un iteratore.
std::vector<int>::iterator it;
for(it = v.begin(); it < v.end(); it++)
{
    std::cout << "Elemento: " << *it << std::endl
    ;
}
```

// Output: stampa uno ad uno gli elementi del vettore.

- **advance()** : Incrementa la posizione dell'iteratore fino all'argomento passato come parametro.
- **next()** : Restituisce un nuovo iteratore dopo aver avanzato di tot posizioni menzionate nell'argomento.
- **prev()** : Restituisce un nuovo iteratore dopo essere retrocesso di tot posizioni menzionate nell'argomento.
- **inserter()** : Per inserire elementi ad qualsiasi posizione nel contenitore. Prende due argomenti: il contenitore e l'iteratore alla posizione in cui l'elemento deve essere inserito.

```
#include <iostream>
#include <iterator>
```

```
std::vector<int> v = { 9, 6, 3};
std::vector<int> v2(2, 5, 8);
```

```
std::vector<int>::iterator it = v.begin();
```

```
std::advance(it, 2);
```

```
std::cout << "Elemento dell'iteratore dopo
advance: " << *it << std::endl; // Output:
Elemento dell'iteratore dopo advance: 3
```

```
std::prev(it, 2);
std::cout << "Elemento dell'iteratore dopo prec:
" << *it << std::endl; // Output: Elemento
dell'iteratore dopo prec: 9
```

```
std::next(it, 1);
std::cout << "Elemento dell'iteratore dopo next:
" << *it << std::endl; // Output: Elemento
dell'iteratore dopo next: 6

// Copio gli elementi di 1 vettore nell'altro usando inserter
// Inserisco gli elementi di v2 in v alla posizione a cui puntava
l'iteratore it.
std::copy(v2.begin(), v2.end(), std::inserter(v,
it));

for(int &x : v)
{
    std::cout << "Elemento: " << x << std::endl;
}

// Output: Gli elementi del vettore con gli elementi aggiunti.
```

---

## Virtual

---

### Virtual functions

**Definizione:** Una funzione **virtuale** è una funzione dichiarata in una classe base che può essere ri-definita (*overridden*) da una classe derivata. Le funzioni **virtuali** ci assicurano che la corretta versione della funzione venga eseguita.

Alcune regole per le **funzioni virtuali**:

- Non possono essere statiche.
- Può essere una **friend function** di un'altra classe.
- Bisognerebbe accedergli attraverso un puntatore o referenza di un tipo alla classe base per ottenere *runtime polymorphism*.
- Il prototipo della funzione dovrebbe essere lo stesso sia nella classe base sia nella classe derivata.
- Sono sempre definiti nella classe base e ridefiniti nella classe derivata. Non è obbligatorio che la classe derivata ri-definisca la funzione, può anche soltanto utilizzare quella della classe base.
- Una classe può avere un **virtual destructor**, ma non un **virtual constructor**.

```
#include <iostream>
```

```
class Base {  
    public:  
        virtual void print()  
        {  
            std::cout << "print in base class" <<  
                std::endl;  
        }  
  
        void show()  
        {
```



```
        std::cout << "show in base class" <<
            std::endl;
    }
};

class Derived : public Base {
public:
    void print() override // override non
        servirebbe, ma aiuta per la manutenzione del codice
        ed indica che la funzione è stata "overridata".
    {
        std::cout << "print in derived class"
            << std::endl;
    }

    void show()
    {
        std::cout << "show in derived class"
            << std::endl;
    }
};

int main()
{
    Base* bPtr;
    Derived d;
    bPtr = &d;

    // Chiamo la funzione virtuale.
    bPtr->print(); // Output: print in derived class

    // Chiamo la funzione non virtuale.
    bPtr->show(); // Output: show in base class

    return 0;
}
```

## Virtual Destructors

**Definizione:** Per rimuovere una classe derivata, la classe base dovrebbe essere definita con un **distruttore virtuale**. Cancellare una classe derivata usando un puntatore alla classe base senza un distruttore virtuale risulta in un comportamento indefinito (*undefined behaviour*).

```
#include <iostream>

class A {
public:
    A()
    {
        std::cout << "Constructor in base
class" << std::endl;
    }

    virtual ~A()
    {
        std::cout << "Destructor in base
class" << std::endl;
    }
};

class B : public A {
public:
    B()
    {
        std::cout << "Constructor in derived
class" << std::endl;
    }

    ~B()
    {
        std::cout << "Destructor in derived
class" << std::endl;
    }
};

int main()
{
```

```
B* bPtr = new B();  
A* aPtr = bPtr;  
  
delete aPtr;  
  
// Output:  
// Constructor in base class  
// Constructor in derived class  
// Destructor in derived class  
// Destructor in base class  
return 0;  
}
```

In linea di massima, se si ha una funzione virtuale, allora è da mettere anche il distruttore virtuale.

## *Virtual Inheritance*

**Definizione:** La Ereditarietà virtuale è usata per risolvere il problema del DDD (*Dreadful Diamond on Derivation*), ovvero quando una classe deriva molteplici classi che derivano dalla stessa classe.

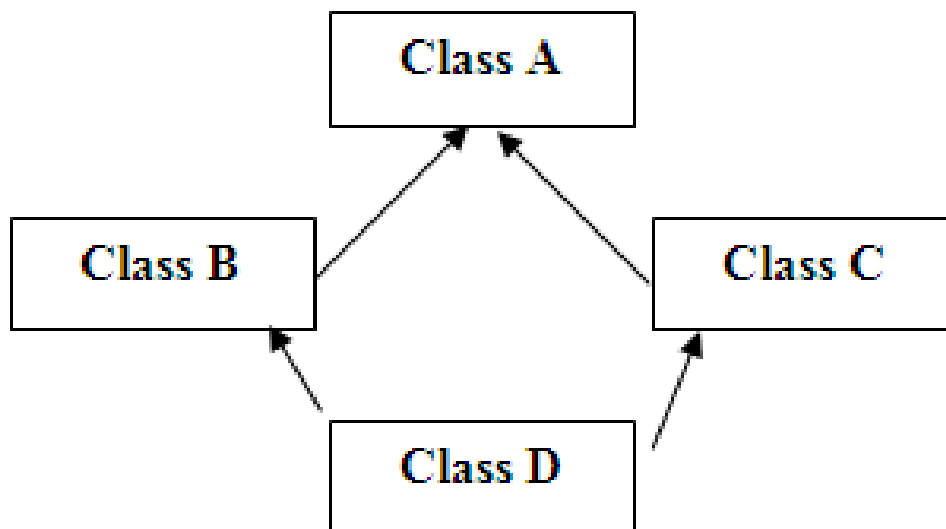


Figura 2.2: Problema del diamante

Come possiamo notare i dati e le funzioni della **classe A** è ereditata due volte dalla **classe D**, una volta per via della **classe B** e una volta per via della **classe C**.

Quando qualsiasi dato o funzione della **classe A** viene acceduto dalla **classe D**, nasce dell'ambiguità su quale dato/funzione chiamare. Quella ereditata da **B** o da **C**? Questo confonde i compilatori e mostrano errori.

Per risolvere questa ambiguità quando la **classe A** è ereditata sia dalla **classe B** sia dalla **classe C**, è dichiarata come **classe base virtuale** (Fare riferimento all'immagine: 2.2 a pag.99).

```
#include <iostream>

class A {
    public:
        void show()
        {
            std::cout << "Show from A" << std::endl;
        }
};

class B : public virtual A {
};

class C : public virtual A {
};

class D : public B, public C {
};

int main()
{
    D d;
    d.show(); // Output: Show from A
}
```

La keyword **virtual** può essere posta sia prima che dopo **public**.

---

## *Polimorfismo*

---

**Definizione:** La parola **polimorfismo** significa *avere molte forme*, questo occorre quando c'è una gerarchia di classi e queste sono correlate attraverso l'ereditarietà.

Ci sono due tipi principali di polimorfismo:

- **Compile time Polymorphism** : si ottiene dall'*overloading* di funzioni o di operatori.
- **Runtime Polymorphism** : si ottiene dall'*overriding* delle funzioni (con la keyword *virtual*).

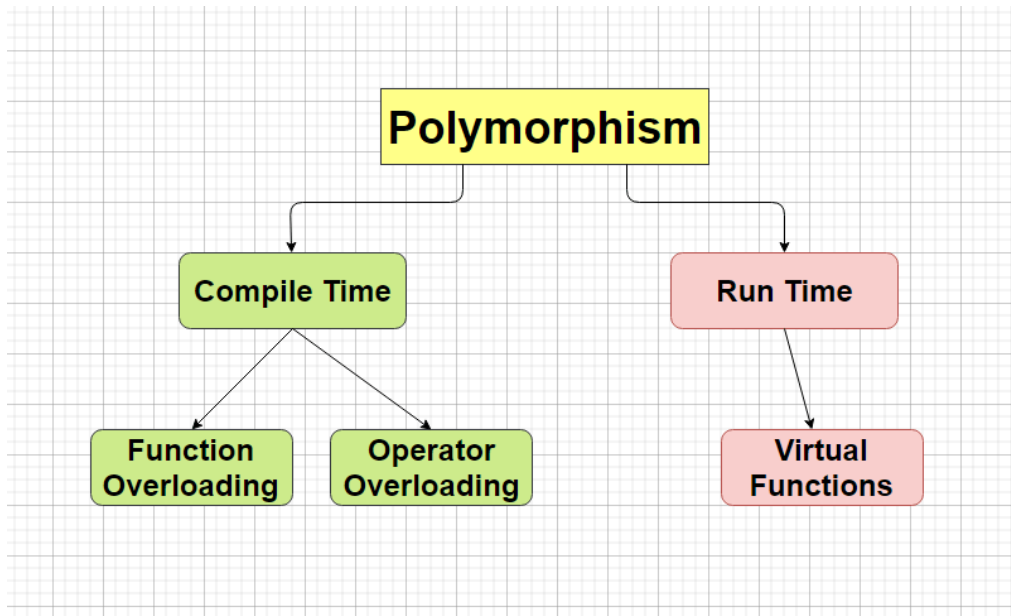


Figura 2.3: Polimorfismo

---

## Overloading

---

**Definizione:** L'**overloading** permette di ridefinire una funzione o un operatore con lo stesso nome e nello stesso scope, ma con una differente implementazione.

### Function Overloading

Si può definire una funzione con lo stesso nome di un'altra purchè abbia argomenti diversi.

```
void func(int x)
{
    std::cout << "Valore di x: " << x << std::
        endl;
}

void func(double x)
{
    std::cout << "Valore di x: " << x << std::
        endl;
}

void func(float x)
{
    std::cout << "Valore di x: " << x << std::
        endl;
}
```

### Operator Overloading

Possiamo ridefinire degli operatori per eseguire delle operazioni nel modo che vogliamo noi.

Utilizziamo la keyword **operator** ed il simbolo dell'operatore per *overload*arlo.

```
class Vec {
public:
```

```

    Vec() {}

    Vec(int x, int y)
    {
        this->x = x;
        this->y = y;
    }

    Vector operator+(const Vec& v)
    {
        Vec vec;
        vec.x = this->x + v.x;
        vec.y = this->y + v.y;
        return vec;
    }

    int getX()
    {
        return this->x;
    }

    int getY()
    {
        return this->y;
    }

private:
    int x;
    int y;
};

int main()
{
    Vec v1(3, 2);
    Vec v2(1, 0);

    Vec v3 = v1 + v2;

    std::cout << "v3.x: " << v3.getX() << "; v3.
        getY(): " << v3.y << std::endl;
    // Output: v3.x: 4; v3.y: 2

```

```
// perché facciamo la x di v1 che è 3 + la x di v2 che è 1
// quindi 4 e
// la y di v1, ovvero 2 + la y di v2, ovvero 0 quindi 2
// quindi v3 ha membri (4,2).
return 0;
}
```

Non tutti gli operatori si possono *overloadare*.  
Gli operatori che non si possono *overloadare* sono: . (punto), ::, ?:(operatore ternario), sizeof.

## *Overloading vs Overriding*

L'**overloading** è la creazione di molteplici definizioni di una funzione cambiando la **signature**: il numero di parametri, la tipologia dei parametri. Il tipo di ritorno non gioca alcun ruolo.

Può essere fatta sia nelle classi basi che in quelle derivate.

L'**overriding** è la ridefinizione di una funzione di una classe base in una classe derivata con la stessa **signature**, stesso tipo di ritorno e parametri.

Può essere fatta solo nelle classi derivate.

Differenza tra **function overloading** e **function overriding**:



Overloading	Overriding
Nessuna keyword è usata.	Keyword <b>override</b> .
Il prototipo cambia in base ai parametri.	Il prototipo non cambia.
Occorre durante compile time.	Occorre durante runtime.
I costruttori possono essere "overloadati".	
I distruttori non possono essere "overloadati".	I distruttori possono essere "overridati".
	Le funzioni virtuali non possono essere "overridate".
Può essere usato per ottenere <i>early binding</i> .	Overriding è anche conosciuto come <i>late binding</i> .
La funzione chiamata viene determinata dal numero di parametri.	La funzione overridden è preceduta dalla keyword <b>virtual</b> nella classe base.
Le funzioni verrebbero ridefinite con lo stesso nome, ma differente numero o tipo di parametri.	
	L'indirizzo dell'oggetto della classe è assegnato al puntatore la cui funzione è chiamata dal puntatore.
	Quando la funzione è definita viene preceduta dalla keyword <b>virtual</b> nel main.

---

## *Tipi di Casts*

---

**Definizione:** Il **casting** è un'operazione che permette la conversione di un valore in un altro. In C++ ci sono diversi tipi di casting:

### ***static\_cast<>***

- **static\_cast<>** : Quello che fa è un cast implicito tra tipi (come int a float, o puntatore a void\*) e può anche chiamare funzioni esplicite per la conversione.

```
float f = 3.69;
int x = static_cast<int>(f);
std::cout << "x: " << x << std::endl; // Output: x:
3
```

### ***const\_cast<>***

- **const\_cast<>** : Serve per aggiungere o rimuovere il **const** ad una variabile. Se la variabile che stiamo cercando di modificare era già const allora questo produce un valore indefinito. Se lo si usa per qualcosa che non era dichiarato come const allora è safe (sicuro farlo, non ci saranno problemi).

```
#include <iostream>
```

```
void print( char* str)
{
    std::cout << str << '\n';
}
```

```
int main()
{
    const char* c = "testo";
    // Ci serve per poter passare un puntatore a char const ad
    // una funzione che prende un puntatore a char senza const.
```

```

    print( const_cast<char*>(c) ); // Output: testo
    return 0;
}

```

### *dynamic\_cast<>*

- **dynamic\_cast<>** : Serve esclusivamente per i casts riguardanti il polimorfismo. Puoi castare un puntatore o una reference a qualsiasi altro tipo di classe. Non solo si può fare un casting verso il basso, ma anche in alto e a lato. Il **dynamic\_cast** cercherà di ritorna l'oggetto desiderato se possibile, altrimenti ritornerà **nullptr** in caso di un puntatore e **std::bad\_cast** nel caso di una reference.
- Ha delle limitazioni. Non funzionerà nel caso in cui diversi oggetti ereditano tutti dallo stessa classe. (il famoso problema del *dreaded diamond*.) e non stai usando l'ereditarietà **virtual**.
- Inoltre può soltanto funzionare con l'ereditarietà pubblica, fallirà con l'ereditarietà **protected** o **private**. Comunque questi tipi di ereditarietà sono rare.

```

// C++ programma per dimostrare che se non c'è
// alcuna funzione virtuale nella Base classe.
#include <iostream>

// Base class declaration
class Base {
    void print()
    {
        std::cout << "Base" << std::endl;
    }
};

// Derived Class 1 declaration
class Derived1 : public Base {
    void print()
    {
        std::cout << "Derived1" << std::endl;
    }
};

```

```

// Derived class 2 declaration
class Derived2 : public Base {
    void print()
    {
        std::cout << "Derived2" << std::endl;
    }
};

// Driver Code
int main()
{
    Derived1 d1;

    // Base class pointer hold Derived1
    // class object
    Base* bp = dynamic_cast<Base*>(&d1);

    // Dynamic casting
    Derived2* dp2 = dynamic_cast<Derived2*>(bp);
    if (dp2 == nullptr)
        std::cout << "null" << std::endl;

    // Output: null, in realtà errore.
    return 0;
}

```

### ***reinterpret\_cast<>***

- **reinterpret\_cast<>** : È quello più pericoloso di tutti e quindi bisogna utilizzarlo con moderazione. Trasforma un tipo direttamente in un altro come cast da un puntatore ad un altro o memorizzare un puntatore in un int, ecc..
- L'unica cosa garantita con questo tipo di cast è che se torni indietro al tipo originale riotterrai lo stesso valore (non succederà se il tipo era più piccolo del tipo originale.)

```

class A {
public:
    int x;
}

```

```
};

class B {
public:
    int x;
};

A *a = new A;
B *b = reinterpret_cast<*B>(a);

a->x = 5;
std::cout << "b: " << b->x << std::endl; // Output:
b: 5
std::cout << "a: " << a->x << std::endl; // Output:
a: 5
```

## *C-style & function-style cast o Regular Cast*

- Questo tipo di cast chiamato **Regular Cast** o **C-style cast** (derivando dal C ovviamente) è molto più potente degli altri tipi di cast, ma allo stesso tempo molto meno sicuro.
- Ignorano i controlli d'accesso quando si esegue uno `static_cast`.
- Permette di fare un cast sicuro ad una classe privata, mentre il suo "equivalente" `static_cast` darebbe un errore a tempo di compilazione (compile-time).

```
double d = 9.87;
int x;

x = (int)d;
std::cout << "x: " << x << std::endl; // Output: x: 9
```

## Ricapitolando

Cast	Definizione
<b>dynamic_cast</b>	per convertire puntatori/references in una gerarchia di ereditarietà.
<b>static_cast</b>	per le conversioni di tipi ordinari.
<b>reinterpret_cast</b>	per reinterpretare bit patterns di basso livello. Usare con cautela.
<b>const_cast</b>	per aggiungere/rimuovere <b>const</b> al cast.

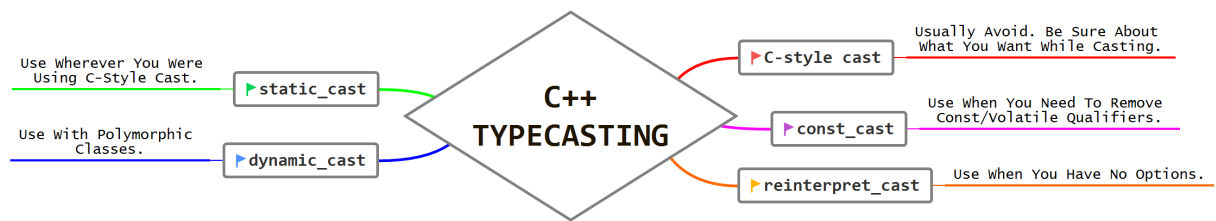


Figura 2.4: Typecasting

---

## Lambdas

---

**Definizione:** Dal C++11 sono presenti le **lambdas** che permettono di creare **funzioni anonime**.

Servono per creare delle funzioni, dei piccoli frammenti di codice che non hanno bisogno di un nome e non verranno riutilizzati.

Sono una parte centrale della **programmazione funzionale**.

Questa è la struttura di una tipica espressione **lambda** :

```
[ clausola di cattura ] ( lista di parametri che
    è opzionale ) -> tipoDiRitorno
{
    // Definizione della lambda.
}
```

Se nella clausola della cattura è presente un **=** (uguale), vuol dire che la lambda può accedere a qualsiasi variabile, se c'è un **&** vuol dire che stiamo accedendo alle variabili per reference, se la clausola **[]** è vuota allora può accedere soltanto alle variabili locali, altrimenti lì saranno presenti i nomi delle variabili che si vogliono utilizzare ("catturate" o per valore o per reference).

Cattura	Definizione
[]	accedere solo alla variabili locali
[=]	accedere a tutte le variabili per valore.
[&]	accedere a tutte le variabili per reference.
[nomeVariabile1, &nomeVariabile2]	"cattura" nomeVariabile per valore e nomeVariabile2 per referencia.

```

#include <iostream>
#include <vector>

std::vector<int> v1 = { 5, 8, 9, 1, 7};
std::vector<int> v2 = {12, 36, 27, 92};

// Lambda.
auto pushinto = [&](int m)
{
    v1.push_back(m);
    v2.push_back(m);
}; // Da notare il ; alla fine.

// Pusha in entrambi v1 e v2 il numero 24
pushinto(24);

// Lambda, accediamo a v1 per valore (quindi ne facciamo una
// copia).
[v1]()
{
    for(auto p = v1.begin(); p != v1.end(); p++)
    {
        std::cout << *p << std::endl;
    }
};

int n = 7;
// trova il primo numero maggiore di n.
// [n] significa che stiamo accedendo e possiamo soltanto accedere
// ad n (per valore, ovvero una copia di essa).
std::vector<int>::iterator p = std::find_if(v1.
begin(), v1.end(), [n](int i)
{
    return i > n;

```

```
});  
  
std::cout << "Il primo numero maggiore di n e\':  
" << *p << std::endl; // Output: Il primo numero  
maggiore di n e' 8  
  
// Qui [=] vuol dire che possiamo accedere a tutte le variabili.  
int countN = std::count_if(v1.begin(), v1.end(),  
    [=](int a)  
    {  
        return a >= n;  
    });  
  
std::cout << "Il numero di elementi piu' grandi o  
uguali ad n sono: " << countN << std::endl;  
// Output: Il numero di elementi più grandi o uguali ad n sono:  
4 (perchè abbiamo inserito anche il 24 nell'operazione  
precedente).
```



---

## *Memoria dinamica*

---

**Definizione:** Riguarda l'allocazione di memoria manualmente da parte del programmatore. La memoria allocata dinamicamente è allocata nell' **Heap** mentre le variabili locali e la memoria non statica viene allocata nello **Stack**.

- **Heap** : memoria dinamica.
- **Stack** : variabili locali e non-statiche.

### *Memoria Dinamica in C*

In C per l'allocazione dinamica della memoria usufruiamo di 4 diverse funzioni: **malloc()** (per allocare), **calloc()**, **realloc()** (per riallocare), **free()** (per liberare la memoria).

Tutte queste funzioni del C, esistono anche nel C++, ma questo ha un suo modo per l'allocazione dinamica della memoria.

### *new e delete*

#### *new*

**Definizione:** L'operatore **new** denota una richiesta di allocazione di memoria nello spazio libero. Se sufficiente memoria è disponibile, l'operatore inizializza la memoria e restituisce l'indirizzo della nuova memoria allocata ed inizializzata al puntatore.

// Esempio 1

```
int *ptr = nullptr;  
ptr = new int;
```

// Esempio 2

```
double *dPtr = new double;
```

// Esempio 3

```
int *p = new int(22);
```

```
// Esempio 4
int *pArray = new int[12];
```

### *array normali vs array con la new*

L'unica differenza è che gli array normali vengono deallocati dal compilatore, mentre quelli creati con la `new` devono essere deallocati dal programmatore.

### *delete*

**Definizione:** Utilizziamo la keyword `delete` per deallocare la memoria precedentemente allocata.

```
// Esempio 1
int *ptr = new int;

delete ptr;

// Esempio 2
int *p = new int[6];

delete[] p;
```

### *Evitare di usare new*

**Definizione:** Ci sono vari motivi per cui evitare o minimizzare gli utilizzi della keyword `new`:

- Il C++ non ha un garbage collector, quindi per ogni `new` ci deve essere una corrispondente `delete`.
- Se viene lanciata un'eccezione poi la memoria non viene mai liberata.
- Dovrebbe essere tutto nel distruttore, concetto del *RAII*.
- Se restituisci per esempio una stringa a qualcuno, ora sono loro a doverla cancellare (con la `delete`). E se a loro volta la passassero come argomento? Quando dovrebbe essere liberata? (con `delete`).
- Può essere un problema nel multi-threading.
- Potrebbe portare a dei *memory leaks*.

---

## ***RAII | Resource Acquisition is initialization***

---

**Definizione:** RAI (*Resource Acquisition is Initialization*) è un idioma comune della programmazione e della gestione delle risorse. Ogni allocazione della risorsa è fatta alla creazione dell'oggetto da parte del **costruttore** mentre la deallocazione (rilascio della memoria) viene fatto dal **distruttore**. Quindi se non ci sono leaks all'oggetto, non ci sono leaks nemmeno alla risorse.

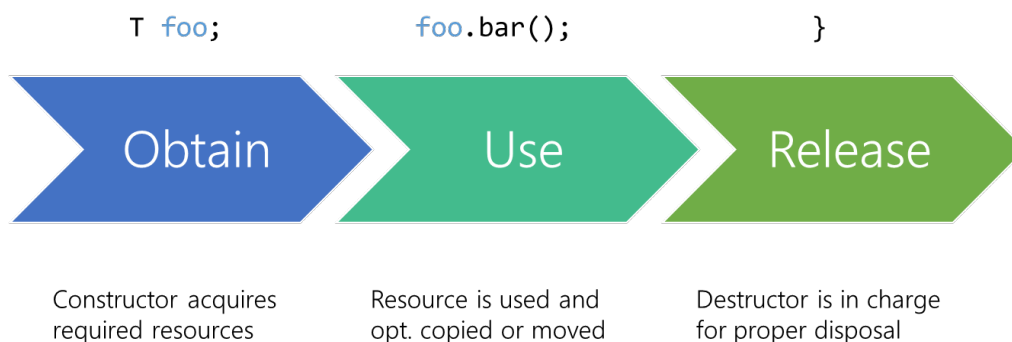


Figura 2.5: RAI

---

## ***Constructor types | Rules of***

---

### ***Rule of Zero***

**Definizione:** La **regola dello zero** (è una regola generale/un'indicazione) afferma che se non hai bisogno di nessuno di questi, allora non ne devi implementare nessuno:

- **Distruttore** : libera tutte le risorse precedentemente allocate.
- **Copy Constructor** : Fa una copia di un oggetto.
- **Copy assign** : overload dell'operatore di assegnamento.

## Copy Constructor

**Definizione:** Il Copy Constructor è un tipo di costruttore che inizializza un oggetto usando un altro oggetto della stessa classe.

Un costruttore di copia ha la seguente struttura:

```
// È un costruttore quindi si chiama con il nome della classe.  
NomeDellaClasse(const NomeDellaClasse &  
    vecchioOggetto);
```

Un copy constructor potrebbe essere chiamato per:

1. Quando un oggetto della classe è ritornato per valore.
2. Quando un oggetto della classe è passato come argomento ad una funzione per valore.
3. Quando un oggetto è costruito sulla base di un altro oggetto.
4. Quando il compilatore genera un oggetto temporaneo.

```
class Point {  
    public:  
        // Costruttore normale  
        Point(int x, int y)  
        {  
            this->x = x;  
            this->y = y;  
        }  
  
        // Copy Constructor  
        // Assiegnamo i valori di x ed y in base ai valori di  
        // un'istanza p della classe Point.  
        Point(const Point &p)  
        {  
            this->x = p.x;  
            this->y = p.y;  
        }  
  
        int getX() { return this->x };  
        int getY() { return this->y };  
  
    private:
```

```

        int x;
        int y;
    };

    int main()
    {
        Point p1(3, 5); // Il costruttore normale viene
                        chiamato qui.
        Point p2 = p1; // Il Copy Constructor viene chiamato
                        qui.

        std::cout << "p1.x: " << p1.getX() << ", p1.y
            : " << p1.getY() << "\n"; // Output: p1.x: 3,
            p1.y: 5
        std::cout << "p2.x: " << p2.getX() << ", p2.y
            : " << p2.getY() << "\n"; // Output: p2.x: 3,
            p2.y: 5

        return 0;
    }

```

## Copy Assign

**Definizione:** Il **copy assignment operator** è ciò che ti permette di assegnare, di copiare un'istanza e di portare i suoi dati in un'altra istanza.

È usato per rimpiazzare i dati di un oggetto precedentemente inizializzato con i dati di qualche altro oggetto.

Se non si dichiara un **assignment operator** allora il compilatore provvederà a fornirtene uno automaticamente.

In linea di massima, se hai bisogno di un **copy constructor** allora avrai bisogno anche di un **copy assignment operator**.

```

class Point {
public:
    // Normal Constructor
    Point(int x, int y)
    {
        this->x = x;
        this->y = y;
    }
}

```

```

// Copy Constructor
Point(const Point& p)
{
    this->x = p.x;
    this->y = p.y;
}

// Copy Assignment
Point& operator=(const Point& p)
{
    this->x = p.x;
    this->y = p.y;
    return *this;
}

int getX() { return this->x };
int getY() { return this->y };

private:
    int x;
    int y;
};

```

## ***=default | Defaulted Functions***

**Definizione:** Le **defaulted functions** in modo esplicito permette di aggiungere **=default** alla fine di una funzione per dichiararla una *funzione default esplicita*.

Questi sono più efficienti.

```

class A {
public:
    A(int a)
    {
        this->a = a;
        this->b = 0;
    }

    A(int a, int b) = default
    {
        this->a = a;
        this->b = b;
    }
};

```

```

};

// Non ci sarebbe bisogno di mettere =default al costruttore
// A(), perché questo è già il costruttore di default.
A();

int a;
};

int main()
{
    // Eseguito usando il default constructor
    A a(2);

    // Eseguito usando il costruttore parametrizzato.
    A b(5, 7);
    return 0;
}

```

### ***=delete | Deleted Functions***

**Definizione:** Apparte, deallocare la memoria, dal C++11 la **delete** ha un nuovo significato: *disabilitare l'utilizzo di una funzione membra*. Queste funzioni sono conosciute come **funzioni deleted esplicitamente**.

```

class A {
public:
    A(int a): x(a)
    {
        this->a = a;
    }

    // Disabilitare il copy constructor.
    A(const A& ) = delete;

    // Disabilitare il copy assignment operator.
    A& operator=(const A& ) = delete;

    int x;
};

```

```

int main()
{
    A a1(3), a2(6), a3(9);

    // Errore, l'utilizzo del copy assignment operator è
    // disabilitato.
    a1 = a2;

    // Errore, l'utilizzo del copy constructor è disabilitato.
    a3 = A(a2);
    return 0;
}

```

Ma qual è l'utilità di far ciò?

1. Previene il compilatore dal generare le **special member functions** (costruttori, distruttori, copy constructor, ecc..) che non vogliamo.
2. Il disabilitare le normali funzioni membro o non-membro previene problemi di promozioni di tipo dal causare una chiamata involontaria alla funzione.

### *Copy Constructor vs Copy Assignment Operator*

Copy Constructor	Copy Assignment Operator
È chiamato quando una nuova istanza viene creata da un oggetto già esistente, come copia di questo.	È chiamato quando ad un oggetto già inizializzato gli viene assegnato un nuovo valore da un oggetto già esistente.
Crea un nuovo blocco di memoria per il nuovo oggetto.	Non crea un nuovo blocco di memoria.
È un costruttore overloaded.	È un operatore bitwise.
Il compilatore fornisce implicitamente un copy constructor se uno non ne esiste già.	Una copia bitwise viene creata se l'assignment operator non viene overloaded.

### *Rule of Three*

**Definizione:** La **regola dei tre**, essenzialmente, afferma che se uno (o anche più) tra questi è definito, allora tutti e tre dovrebbero essere definiti:

- **Distruttore** : libera tutte le risorse precedentemente allocate.



- **Copy Constructor** : Fa una copia di un oggetto.
- **Copy assignment operator** : overload dell'operatore di assegnamento.

I costruttori e gli assignment operator generati implicitamente fanno una **shallow copy** (copia dei dati di tutte le variabili dell'oggetto originale. Ha problemi se i dati son allocati con memoria dinamica, in quel caso faranno referenza alla stessa locazione di memoria) dei dati membri. Noi abbiamo bisogno di una **deep copy** (copia dei dati di tutte le variabili e alloca simili risorse di memoria con lo stesso oggetto) quando la classe contiene puntatori che puntano a risorse di memoria allocate dinamicamente.

## Move Constructor

**Definizione:** I **Move Constructor** lavorano con le referenze **rvalue** e le **move semantics** (le move semantics riguardano il puntare ad un oggetto già esistente in memoria). Fa puntare il puntatore del nuovo oggetto ai dati dell'oggetto temporaneo e pone a null il puntatore degli oggetti temporanei.

I **Move Constructors** spostano le risorse nell'*heap* e l'assegnano al nuovo oggetto. A differenza dei **copy constructors** i **move constructors** prevengono dall'inutile copiatura di dati di memoria.

```
#include <iostream>
#include <vector>

class A {
public:
    // Costruttore
    A(int x)
    {
        data = new int;
        *data = x;
        std::cout << "Costruttore chiamato
per " << x << std::endl;
    }

    // Copy Constructor
    A(const A& source)
    : A { *source.data }
    {
```

```

        std::cout << "Copy constructor in
        deep copy chiamato per " << *
        source.data << std::endl;
    }

    // Move Constructor
    A(A&& source) : data {source.data}
    {
        std::cout << "Move Constructor
        chiamato per " << *source.data <<
        std::endl;
        source.data = nullptr;
    }

    // Distruttore
    ~A()
    {
        if( data != nullptr)
        {
            std::cout << "Distruttore
            chiamato per " << *data << std
            ::endl;
        } else {
            std::cout << "Distruttore
            chiamato per nullptr " << std
            ::endl;
        }

        // Liberiamo la memoria assegnata all'oggetto.
        delete data;
    }
};

int main()
{
    // Vector della classe A.
    std::vector<A> vec;

    // Inseriamo oggetti della classe A.
    vec.push_back(A {9} );
    vec.push_back(A {18} );

```

```

// Output:
// Costruttore chiamato per 9
// Move Constructor chiamato per 9
// Distruttore chiamato per nullptr
// Costruttore chiamato per 18
// Move Constructor chiamato per 18
// Costruttore chiamato per 9
// Copy constructor in deep copy chiamato per 9
// Distruttore chiamato per 9
// Distruttore chiamato per nullptr
// Distruttore chiamato per 9
// Distruttore chiamato per 18

return 0;
}

```

### *lvalues*      *references*      &      *rvalues*      *references*

**Definizione:** Gli **l-values** fanno riferimento alla locazione di memoria che definisce l'oggetto. Gli **r-values** fanno riferimento al valore che è memorizzato in qualche indirizzo di memoria.

Proprietà degli **r-values**:

- Estendono la durata di vita degli oggetti temporanei a cui sono assegnati.
- Gli **r-values** non costanti permettono di modificare l'**rvalue**.

**Importante:** Le referenze **lvalues** possono essere assegnate con le **rvalues**, ma le referenze **rvalues** non possono essere assegnate con le **lvalues**.

```
#include <iostream>
```

```

int main()
{
    int a = 7;

    // Dichiariamo una lvalue reference
    int& lref = a;

    // Dichiariamo una rvalue reference
    int&& rref = 15;
}

```

```

std::cout << "lref: " << lref << "\n"; //
    Output: lref: 7
std::cout << "rref: " << lref << "\n"; //
    Output: rref: 15

// Sia il valore di a che della lref vengono cambiati così
lref = 18;

// Valore della rref cambia
rref = 24;

std::cout << "lref: " << lref << "\n"; //
    Output: lref: 18
std::cout << "rref: " << lref << "\n"; //
    Output: rref: 24

// Questa riga genererà un errore perché l-value non può
// essere assegnata al r-value
// int &&ref = a;
return 0;
}

```

Usi delle referenze **lvalues**:

- Possono essere usate come alias di un oggetto già esistente.
- Possono essere usati per implementare le semantiche *pass-by-reference*.

Usi delle referenze **rvalues**:

- Sono usati per lavorare con il **move constructor** ed il **move assignment**.
- Non possono congiungere delle referenze lvalue non costanti di tipo '**int&**' con un rvalue di tipo '**int**'.
- Non possono mettere assieme delle referenze rvalue di tipo '**int&&**' con dei lvalues di tipo '**int**'.

## *Move Assignment Operator*

**Definizione:** Similmente al copy assignment in cui possiamo copiare un lvalue, possiamo anche muovere valori da un oggetto ad un altro senza costruirne uno

nuovo. Chiamiamo questo: **Move Assignment**. Muoviamo i valori da un oggetto ad un altro oggetto esistente.

Per fare questo overloadiamo l'operatore `=`, non che prenda un **lvalue** come nei **copy constructors**, ma un **rvalue**.

```
#include <iostream>

class A {
public:
    int a;
    // Move Assignment
    A& operator=(A&& other)
    {
        this->a = other.a;
        other.a = 0;
        return *this;
    }
};

int main()
{
    A a;
    a.a = 1;

    A b;
    b = std::move(a); // Chiamiamo l'operatore
                       // overloadato.

    std::cout << a.a << std::endl; // Output: 0
    std::cout << b.a << std::endl; // Output: 1
    return 0;
}
```

## *Rule of Five*

**Definizione:** La **regola dei cinque** è applicata per la gestione delle risorse. Se uno (o più) fra questi 5 viene implementato e le **move semantics** sono desiderate allora vanno implementate tutte e 5.

- **Distruttore** : libera tutte le risorse precedentemente allocate.
- **Copy Constructor** : Fa una copia di un oggetto.

- **Copy assignment operator** : overload dell'operatore di assegnamento.
- **Move Constructor** : Al posto di copiare come il copy constructor, trasferisce le risorse e pone a null i puntatori degli oggetti temporanei.
- **Move Assignment Operator** : Si può usare un move assignment operator per trasferire la proprietà da un oggetto ad un altro.

## Special Members

compiler implicitly declares

	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

user declares

Figura 2.6: Rule of Five

---

## *Move Semantics*

---

**Definizione:** Le move semantics permettono, sotto certe condizioni, di trasferire la proprietà delle risorse esterne di qualche oggetto. Questo è importante per due motivi:

1. Trasformare delle costose copie in delle move economiche.
2. Implementare dei tipi di "move-only" sicure; ovvero tipi che per le copie non hanno senso, ma hanno senso per le move.

Per queste è molto importante il concetto di **lvalue** e di **rvalue**:

- **lvalue T&** : riguardano il copiare.
- **rvalue T&&** : riguardano il trasferire.

Per muovere un oggetto useremo **std::move(oggetto)**. Questa funzione restituisce un **rvalue** all'oggetto, così che possiamo rubare i dati dall'oggetto in quello nuovo.

**std::move(obj)** non cambia il contenuto dell'oggetto, ma **auto obj2 = std::move(obj)** possibilmente sì.

### *Fallbacks of move semantics*

1. Chiamare la **std::move()** su un oggetto costante, di solito, non produce effetti.
  - a) Non ha senso rimuovere o muovere/trasferire le risorse di un oggetto costante.
2. La semantica della copiatura è un ripiego soltanto se la semantica della copiatura è supportata.
3. Se non c'è implementazione che prenda l'**rvalue** come argomento allora ci sarà l'ordinario e costante **lvalue** che verrà usato.
4. Se una funzione non prende un **rvalue** e **lvalue** allora un errore a compile-time verrà generato.
- 5.

### *syntax vs semantics*

Syntax (Sintassi):

- Riguarda le regole per scrivere qualsiasi cosa in un linguaggio di programmazione.
- Non ha niente a che fare con il significato di ciò che si è scritto.

- Una dichiarazione è sintatticamente valida se segue tutte le regole.
- È legata alla grammatica e alla struttura della lingua.
- Gli errori di sintassi si trovano dopo che il programma è stato eseguito.
- Alcuni esempi: un punto e virgola mancante. Sono errori semplice da trovare.

#### Semantics (Semantica):

- Riguarda il significato associato alla dichiarazione in una lingua di programmazione.
- È tutto riguardante il significato della dichiarazione che interpreta il programma.
- Gli errori sono gestiti a runtime.
- Si riferisce al significato delle linee di codice associate al linguaggio.
- Anche se un pezzo di codice ha la sintassi corretta, potrebbe comunque non fare ciò che voleva che facesse. Sono errori un po' più complicati da trovare.

// Codice per dimostrare un errore semantico

```
#include <iostream>
```

```
int main()
```

```
{
```

```
    // Dichiarazione di ritorno prima del cout
```

```
    return 0;
```

```
    std::cout << "Hello World!" << std::endl; //
```

```
    Output: nessun output perché è dopo il return statement.
```

```
}
```

Sintassi	Semantica
Si riferisce alle regole di qualsiasi riga di codice.	Si riferisce al significato associato a qualsiasi riga di codice.
Errori di sintassi. Occorrono a compile time Alcuni esempi: mancanza di un punto e virgola.	Si riferisce ad un errore semantico. Occorre quando delle righe di codice sono valide sintatticamente, ma non fanno quello che il programmatore volesse che facessero.



## *Ricapitolando le Move Semantics*

Ricapitolando:

- Le **Move Semantics** ci permettono di ottimizzare la copiatura di oggetti. È usato spesso implicitamente (per gli oggetti temporanei) o esplicitamente con `std::move()`.
- `std::move()` significa *non ho più bisogno di questo valore*.
- Un oggetto segnato con `std::move()` non è mai parzialmente distrutto. Il distruttore verrà chiamato per distruggere l'oggetto appropriatamente.

---

## *Classi Astratte*

---

**Definizione:** Una classe **astratta** è una classe generica dove non siamo in grado di implementare tutte le funzioni e che il suo unico scopo è quello di essere ereditata da altre classe che implementeranno le sue funzioni. Una classe **astratta** per essere ritenuta tale deve, almeno, implementare una **pure virtual function**.

### *Pure Virtual Functions*

**Definizione:** Una **pure virtual function** o **abstract function** è una funzione virtuale per cui non abbiamo bisogno di scrivere l'implementazione. Le implementazioni verranno poi implementate nelle classe derivate. Si dichiara una **pure virtual function** assegnando 0 alla dichiarazione.

```
class Test {  
    public:  
        // Pure Virtual Function  
        virtual void show() = 0;  
};
```

### *Abstract Class*

**Definizione:** Proprietà di una classe astratta:

- Possono avere funzioni normali e allo stesso tempo **pure virtual functions**.
- Non può essere istanziata, ma si possono creare puntatori e referenze della classe astratta.
- Le classi astratte sono usate principalmente per **Upcasting** così che la classe derivata possa usare la sua interfaccia.
- Se una classe **astratta** ha una classe derivata, essa deve implementare tutte le **pure virtual functions** altrimenti diventeranno anch'essi una classe **astratta**.
-

- 

```
#include <iostream>

class Shape {
public:
    virtual int getArea() = 0;
    void setWidth(int w)
    {
        width = w;
    }

    void setHeight(int h)
    {
        height = h;
    }

protected:
    int width;
    int height;
};

class Rectangle : public Shape {
public:
    int getArea()
    {
        return width * height;
    }
};

class Triangle : public Shape {
public:
    int getArea()
    {
        return (width * height) / 2;
    }
};

int main()
{
```

```
Rectangle rect;
Triangle tri;

rect.setWidth(5);
rect.setHeight(4);

std::cout << "Area rettangolo: " << rect.
    getArea() << std::endl; // Output: Area
    Rettangolo: 20

tri.setWidth(5);
tri.setHeight(4);

std::cout << "Area triangolo: " << tri.
    getArea() << std::endl; // Output: Area
    Triangolo: 10
return 0;
}
```

### *Abstract class vs Interface*

In un certo senso, possono quasi essere definite come, quelle che in altri linguaggi, si chiamano **Interfacce**, ovvero un'interfaccia mette semplicemente a disposizione le funzioni, ovvero le loro dichiarazioni e lascia a chi **implementa** l'interfaccia implementare queste funzioni. Il C++ non ha **Interfacce**, ma possiamo quasi dire che le **classi astratte** sono le **interfacce** del C++.

Quindi si può simulare un'interfaccia in C++ ponendo tutte le funzioni della classe **astratta** come **pure virtual functions**.

---

## *Eccezioni*

---

**Definizione:** Cosa sono le **eccezioni** (in inglese *exceptions*)? Le **eccezioni** sono delle anomalie o condizioni anormali incontrate durante l'esecuzione del programma.

Il C++ a differenza del C mette a disposizione delle keywords per occuparsi del codice che potrebbe lanciare un'eccezione.

Durante l'esecuzione del codice diversi errori potrebbero capitare:

- errori nel codice.
- errori nell'input.
- altri tipi di errori.

Quando un errore occorre, C++, di solito, si fermerà e genererà un errore (lancerà un'eccezione).

### *try|catch|throw*

**try, catch, throw:** Queste sono le keywords per *Exception Handling* (Gestione delle eccezioni):

#### *try*

Il **try** ti permette di definire una porzione di codice che verrà testata durante la sua esecuzione. Ovvero se il blocco di codice nel **try** non lancerà alcuna eccezione allora verrà eseguito il **try** e poi il codice seguente ad esso, altrimenti verrà eseguito il codice nella clausola **catch**.

#### *catch*

Il **catch** ti permette di definire un blocco di codice che verrà eseguito se un errore sarà accorso nel **try**.

```

try {
    // Blocco di codice del try.
} catch (NomeEccezione e) {
    // Blocco di codice del catch.
}

```

C'è uno special catch chiamato *catch all* che permette di "catturare" tutte le eccezioni. Si fa in questo modo: `catch(...)` (quindi un catch con l'ellissi, ovvero i tre puntini).

### ***std::throw***

### ***exception***

Il `throw` ti permette di lanciare un'eccezione quando un problema è stato rilevato.

```

double division(int a, int b)
{
    if (b == 0)
    {
        std::throw "Divisione per zero!";
    }
    return a/b;
}

int main()
{
    int x = 22;
    int y = 0;
    double z = 0;

    try {
        z = division(x, y);
    } catch (const char* msg) {
        //std::cerr è lo standard error, lo si può usare per
        //mostrare errori sullo schermo.
        std::cerr << msg << std::endl; //Output:
        //Divisione per zero!
    }

    return 0;
}

```

### *Errori a compile time ed errori a runtime*

Gli errori a **compile time** sono degli errori che occorrono quando si violano le regole di sintassi del linguaggio, come per esempio:

- Mancanza di una parentesi.
- Stampare il valore di una variabile senza dichiararla.
- Mancanza di un punto e virgola.

Gli errori a **runtime** sono quegli errori che occorrono durante l'esecuzione del programma dopo che la compilazione sia avvenuta con successo.

<b>Compile-Time Errors</b>	<b>Runtime-Errors</b>
Errori di sintassi. Prevengono l'esecuzione del codice. Vengono rilevati dal compilatore e possono essere corretti nel momento della programmazione.	Non sono rilevati dal compilatore. Prevengono il codice dalla completa esecuzione. Vengono fixati (sistemati) solo dopo che il codice viene eseguito.

---

*Operazioni su file*

---

Definizione: