

A Simple Guide to Modern C++

Beginner

Intermediate

Advanced

Luca Rengo

The Publisher

Indice

Indice	2
1 Basi del Linguaggio	11
1.1 Introduzione	11
1.2 Linguaggio	11
1.3 Preprocessor	12
1.4 Compilatore	12
1.5 Linker	13
1.6 Runtime Vs Compile time	13
1.7 C vs C++	14
1.8 Tipi di Dati	16
Cast	17
1.9 Costanti	19
const vs constexpr	19
static	19
Variabili statiche	19
Membri statici delle classi	19
static const	20
static constexpr	20
1.10 Arrays e Matrici	20
Arrays	20
Matrici	21
1.11 Operatori Aritmetici	22
1.12 Operatori Relazionali	22
1.13 Operatori Bitwise	23
1.14 Operatori di Assegnamento e Operatori Unari	24
Operatori di Assegnamento	24
Operatori Unari	25
1.15 Operatori Logici	27
1.16 Altri Operatori	27
1.17 Condizione If	29

If else if else	29
Operatore Ternario	29
1.18 Switch	30
Switch Cases Break	30
Default Case	31
1.19 Loops	32
While	32
Do-While	32
Continue	33
goto	33
For	34
Foreach	34
Cicli annidati	34
Cicli Infiniti	34
1.20 Enumeratori	35
1.21 Puntatori	37
Aritmetica dei puntatori	37
Puntatori a puntatori	38
1.22 References	39
References vs Puntatori	40
NULL vs nullptr	41
1.23 Stringhe	42
Char	42
C-string	42
char*	42
Tabella ASCII	43
std::string	43
char* vs std::string vs char[]	44
Usare char*	44
Usare std::string	45
Casi in cui preferire char* ad std::string	45
Usare char[]	45
Escape characters	46
1.24 Funzioni	47
return	47
void	48
main	48
Funzioni ricorsive	49
Argomenti passati per valore	50
Argomenti passati per referenza	50
Funzioni che ritornano puntatori	52

1.25 Variables Scope	53
Variabili Locali	53
Parametri formali	54
Variabili Globali	54
1.26 Header files	55
Only Once Headers pragma once ifndef	55
Cosa sono le librerie?	56
Header files libreria Standard	56
Librerie create dagli utenti	57
Differenza tra .h vs .hpp	57
1.27 Namespaces	58
std:: vs using namespace std	59
Mai mettere using namespace in un header file!	61
1.28 Strutture	62
typedef	63
Funzioni nelle strutture	64
Strutture nelle strutture	64
Puntatore ad una struttura	65
Array di Strutture	66
Strutture come parametri e come ritorno	66
Strutture in C vs in C++	69
1.29 Union	69
structure vs union	71
1.30 Classi	72
Costruttori e Distruttori	73
Costruttori	73
Initialization List	73
Distruttori	74
Proprietà del distruttore	74
Quando viene chiamato il distruttore?	75
Access modifiers	75
Incapsulamento	75
scope resolution operator ::	77
Getters & Setters	78
Ereditarietà	78
this pointer	81
Multi-Ereditarietà	81
Forward Declaration	83
Chiamata a funzione statica e a membro	84
Static nelle Classi	84
Funzioni e la keyword const	85

Class vs Struct	86
1.31 Convenzioni del linguaggio	87
Generale	87
Parentesi Graffe	87
Indentazione	87
Convenzioni sui nomi	88
Ordine Inclusione Header files	88
Files	88
Types	89
Variabili	89
Funzioni	89
Costanti	89
Namespaces	89
Header Guards	90
Spazi bianchi addizionali	90
Linee Guida Miste	92
2 Concetti Intermedi	93
2.1 Introduzione	93
2.2 STL Standard Template Library	93
Che cos'è #include <bits/stdc++.h>?	94
2.3 Templates	95
2.4 std::vector<>	97
2.5 Iteratori	99
2.6 Virtual	103
Virtual functions	103
Virtual Destructors	105
Virtual Inheritance	106
2.7 Polimorfismo	109
2.8 Overloading	110
Function Overloading	110
Operator Overloading	111
Overloading vs Overriding	112
2.9 Tipi di Casts	114
static_cast<>	114
const_cast<>	114
dynamic_cast<>	115
reinterpret_cast<>	116
C-style & function-style cast o Regular Cast	117
Ricapitolando	118
2.10 Lambdas	119

2.11 Memoria dinamica	121
Memoria Dinamica in C	121
new e delete	121
new	121
array normali vs array con la new	122
delete	122
Evitare di usare new	122
2.12 RAII Resource Acquisition is initialization	123
2.13 Constructor types Rules of	123
Rule of Zero	123
Copy Constructor	124
Copy Assign	125
=default Defaulted Functions	126
=delete Deleted Functions	127
Preventing Object Copy	128
Copy Constructor vs Copy Assignment Operator	129
Rule of Three	129
Move Constructor	130
lvalues references & rvalues references	132
Move Assignment Operator	133
Rule of Five	134
2.14 Move Semantics	135
Fallbacks of move semantics	136
syntax vs semantics	136
Ricapitolando le Move Semantics	138
2.15 Classi Astratte	139
Pure Virtual Functions	139
Pure Virtual Destructor	139
Abstract Class	140
Abstract class vs Interface	142
2.16 Eccezioni	143
try catch throw	143
try	143
catch	143
std::throw exception	144
Errori a compile time ed errori a runtime	145
noexcept	145
Assertions	147
2.17 Operazioni di Input/Output	149
Input-Output stream	149
std::endl vs newline	150

Manipolazione degli stream	151
Operazioni su file	154
2.18 std::chrono	157
Duration	157
Clock	158
Time point	158
Calcolare il tempo di esecuzione di un blocco di codice	158
2.19 Generatori di numeri pseudo-casuali	160
Numeri casuali come in C	160
Cosa vuol dire e perché pseudo-random?	162
I vari tipi di generatori	162
2.20 Map Set Pair Hash Table	172
Map	172
unordered_map	173
unordered_map function pointer	174
multimap	177
Set	178
multiset	179
Pair	179
Tuple	182
Pair vs Tuple	184
Hash Table	185
2.21 Type Traits	186
decltype	186
2.22 RTTI Run-Time Type Information	188
Runtime Casts	188
2.23 Typeinfo	189
typeid	190
typeinfo	191
2.24 Ellissi	192
2.25 Inline Functions	194
2.26 Copy Elision	197
2.27 Conversion Constructor & Explicit keyword	198
Explicit Keyword	198
Conversion Constructor	199
2.28 Private Destructors	200
2.29 Command Line Arguments	202
3 Concetti Avanzati	205
3.1 Introduzione	205
3.2 Friend Keyword	206

	Friend Class	206
	Friend Function	207
3.3	Smart Pointers	209
	Differenze con i puntatori normali	209
	unique pointers	211
	Differenza tra std::make_unique vs std::unique_ptr	213
	Ownership move	214
	Operazioni sugli unique_ptr	215
	Passare uno unique_ptr ad una funzione	216
	Restituire un unique_ptr	217
	Membri delle classi: unique pointer vs raw pointer vs reference	217
	share pointers	218
	Differenza tra std::shared_ptr vs std::make_shared	219
	Operazioni sui shared pointers	220
	Distruzione degli shared pointers	221
	Passare gli shared pointers ad una funzione	223
	Restituire gli shared pointers	224
	static_pointer_cast	225
	enable_shared_from_this	227
	Performance degli shared pointers	228
	Shared pointers, unique pointers or raw pointers .	228
	weak pointers	229
	Problema della dipendenza ciclica	229
	Quando usare i weak pointers?	230
	Operazioni sugli weak pointers	230
3.4	Uniform Real Distribution	233
3.5	Copy-And-Swap Idiom	234
3.6	Tecniche per il debugging	237
3.7	Puntatore a funzione	238
3.8	7 Concetti Avanzati	241
	RAII	241
	Return Type Resolver	244
	Type Erasure	245
	Type erasure usando void* (come nel C)	246
	Type erasure usando i templates	246
	Type erasure usando il polimorfismo	246
	Type erasure usando le union	247
	Type erasure usando i containers generici	248
	CRTP	250
	3-way comparison operator Spaceship Operator .	253

Virtual Constructor	254
SFINAE	256
Proxy	259
4 Le gemme degli Algoritmi	263
4.1 Introduzione	263

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, encapsulazione, ereditarietà il che significa che il C non supporta la programmazione ad oggetti.	Supporta polimorfismo, encapsulazione 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 encapsulati in forma di un oggetto.
Non supporta l'information hiding.	I dati sono encapsulati 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:

```
size_t      corrisponde ad unsigned int
uint8_t     unsigned char
uint16_t    unsigned int
uint32_t    unsigned long
```

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

```
string          (includere string)
wchar_t   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 <cstdint> 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

const	constexpr
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 rimuovo 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.
- **$\hat{=}$** : Combinazione di = e $\hat{=}$, 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_variabile** : 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 volendo lo 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++; // identico 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 verrano 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 { Lunedi , 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: " << var << std::
    endl; // Output: Valore di *ptr: 369
std::cout << "Valore di **pptr: " << var << 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

```

The screenshot shows a debugger interface with a code editor and several toolbars. The code editor displays the following C++ code:

```

29062 void Game::ComposeFrame()
29063 {
29064     int v = 69;
29065     int& r = v; /* int& significa reference, questa variabile r reference questa variabile v; possiamo pensarla come un alias o un remote-control, se facciamo qualcosa ad r, lo facciamo anche a v. Possiamo manipolare una variabile indirettamente. Referencing funziona solo con una variabile, se mettessimo per esempio un numero non andrebbe.*/
29066     r += 420; ≤ 1ms elapsed
29067     v -= 69;
29068
29069
29070
29071

```

The variable `r` is highlighted with a red circle, indicating it is the current variable being examined. The `Autos` toolbar shows the following variable values:

Name	Type	Value
<code>r</code>	int &	69
<code>this</code>	Game *	0x00d7a4 (wnd=(kbd=(autorepeatEnabled=false keystates=(size=256) keybuffer...)
<code>v</code>	int	69

The `Output` window shows the application has loaded several DLLs.

Figura 1.3: Reference

This screenshot is similar to Figure 1.3, but the variable `r` has been modified. The code editor now shows:

```

29062 void Game::ComposeFrame()
29063 {
29064     int v = 69;
29065     int& r = v; /* int& significa reference, questa variabile r reference questa variabile v; possiamo pensarla come un alias o un remote-control, se facciamo qualcosa ad r, lo facciamo anche a v. Possiamo manipolare una variabile indirettamente. Referencing funziona solo con una variabile, se mettessimo per esempio un numero non andrebbe.*/
29066     r += 420;
29067     v -= 69; ≤ 1ms elapsed
29068
29069
29070

```

The `Autos` toolbar shows the following variable values:

Name	Type	Value
<code>r</code>	int &	489
<code>this</code>	Game *	0x00d5f604 (wnd=(kbd=(autorepeatEnabled=false keystates=(size=256) keybuffer...)
<code>v</code>	int	489

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 NULL .	Possono avere un valore NULL .
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.

NULL vs nullptr

NULL portato dal C corrisponde semplicemente a 0 (è una macro) e non necessariamente ad un puntatore. Mentre **nullptr** (è un *pointer literal*), specifico del C++ è sempre un puntatore, di tipo `std::nullptr_t` (è un prvalue di tipo `nullptr_t`), un puntatore a tutti gli effetti. Se lo si cerca di adottare ad un'altra variabile, tipo ad un int darà errore.

`nullptr` è implicitamente convertibile a qualsiasi tipo di puntatore.

```
#include <iostream>

int main()
{
    int ptr = nullptr; // Errore.

    int *ptr = nullptr; // Va bene.
    return 0;
}

#include <iostream>

void func(int n);
void func(char *s);

func( NULL ); // Quale delle due funzioni verrà chiamata?
La prima, perché NULL è 0 e quindi un int.
```

`nullptr` è definito nell'header `<cstddef>`, ma non serve includerlo perché è una *built-in keyword*.

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 concisa 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	@	96	'	
01	SOH	(Start of Header)	33	!	A	97	a	
02	STX	(Start of Text)	34	"	B	98	b	
03	ETX	(End of Text)	35	#	C	99	c	
04	EOT	(End of Trans.)	36	\$	D	100	d	
05	ENQ	(Enquiry)	37	%	E	101	e	
06	ACK	(Acknowledgement)	38	&	F	102	f	
07	BEL	(Bell)	39	'	G	103	g	
08	BS	(Backspace)	40	(H	104	h	
09	HT	(Horizontal Tab)	41)	I	105	i	
10	LF	(Line feed)	42	*	J	106	j	
11	VT	(Vertical Tab)	43	+	K	107	k	
12	FF	(Form feed)	44	,	L	108	l	
13	CR	(Carriage return)	45	-	M	109	m	
14	SO	(Shift Out)	46	.	N	110	n	
15	SI	(Shift In)	47	/	O	111	o	
16	DLE	(Data link escape)	48	0	P	112	p	
17	DC1	(Device control 1)	49	1	Q	113	q	
18	DC2	(Device control 2)	50	2	R	114	r	
19	DC3	(Device control 3)	51	3	S	115	s	
20	DC4	(Device control 4)	52	4	T	116	t	
21	NAK	(Negative acknowl.)	53	5	U	117	u	
22	SYN	(Synchronous idle)	54	6	V	118	v	
23	ETB	(End of trans. block)	55	7	W	119	w	
24	CAN	(Cancel)	56	8	X	120	x	
25	EM	(End of medium)	57	9	Y	121	y	
26	SUB	(Substitute)	58	:	Z	122	z	
27	ESC	(Escape)	59	;	[123	{	
28	FS	(File separator)	60	<	\	124		
29	GS	(Group separator)	61	=]	125	}	
30	RS	(Record separator)	62	>	^	126	~	
31	US	(Unit separator)	63	?	95	_		
127	DEL	(Delete)						

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
<code>length()</code>	restituisce la lunghezza della stringa.
<code>capacity()</code>	restituisce la capacità allocata alla stringa che può essere più o meno la lunghezza.
<code>resize()</code>	cambia la grandezza della stringa che può essere aumentata o diminuita.
<code>shrink_to_fit()</code>	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.	

Escape characters

Definizione: Le sequenze di fuga sono usate per rappresentare certi caratteri speciali nelle stringhe e nei caratteri.

Caratteri di controllo:

Compatibili con l'encoding ASCII.

Escape sequence	Definizione
\a : \x07	alert (bell)
\b : \x08	backspace
\t : \x09	horizontal tab
\n : \x0A	newline (or line feed)
\v : \x0B	vertical tab
\f : \x0C	form feed
\r : \x0D	carriage return
\e : \x1B	escape (non-standard GCC extension)

Caratteri di punteggiatura:

Escape sequence	Definizione
\"	quotation mark
\'	apostrophe
\?	question mark (used to avoid trigraphs)
\\	backslash

Caratteri di referenze numeriche:

Escape sequence	Definizione
\	+ 3 cifre in ottale
\x	+ qualsiasi cifra in esadecimale
\u	+ 4 cifre esadecimali
\U	+ 8 cifre esadecimali
\0 = \00 = \000	octal escape for null character

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

int y = 15;
std::cout << "Fibonacci di " << y << " è\' "
    << 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, perché stiamo lavorando sulle copie delle
// variabili, non sulle variabili stesse.

// Parte 2: Usare argomenti passati per riferenza
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 ero 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/compilatore 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):

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

.h	.hpp
Sia per il C che per il C++ Dal punto di vista del C++, il codice C verrà definito come <code>extern "C"</code> 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
// Perchè 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 perchè 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 instanziare:

```

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 matriscia.

```
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

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

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.

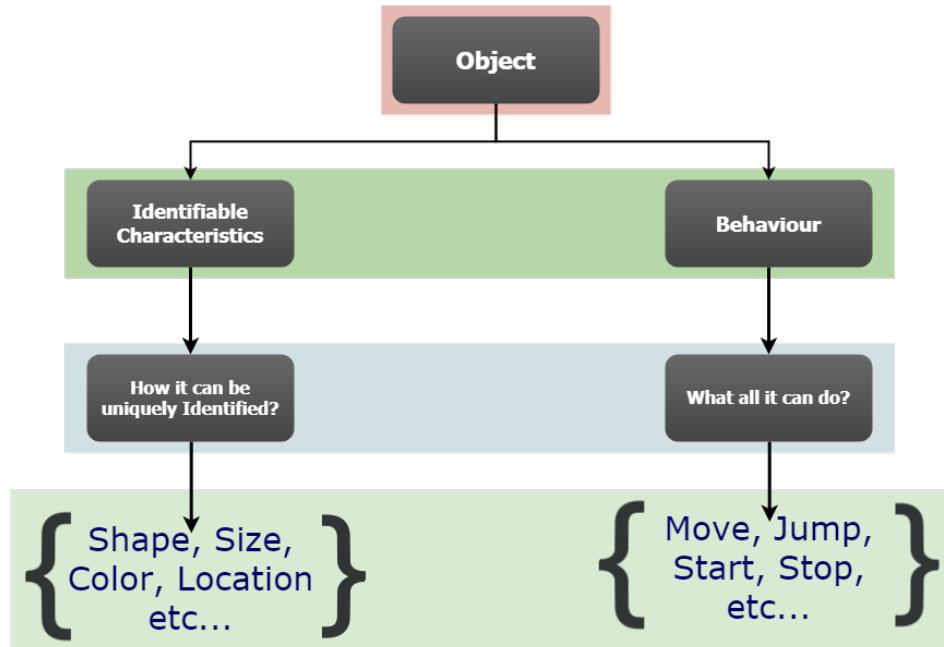


Figura 1.7: Dati e comportamenti di una classe

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
public	accessibile a tutti.
private	accessibile solo all'interno della classe stessa.
protected	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 getTotalle(){
            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.getTotalle() <<
        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 è: " << 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
    }
```

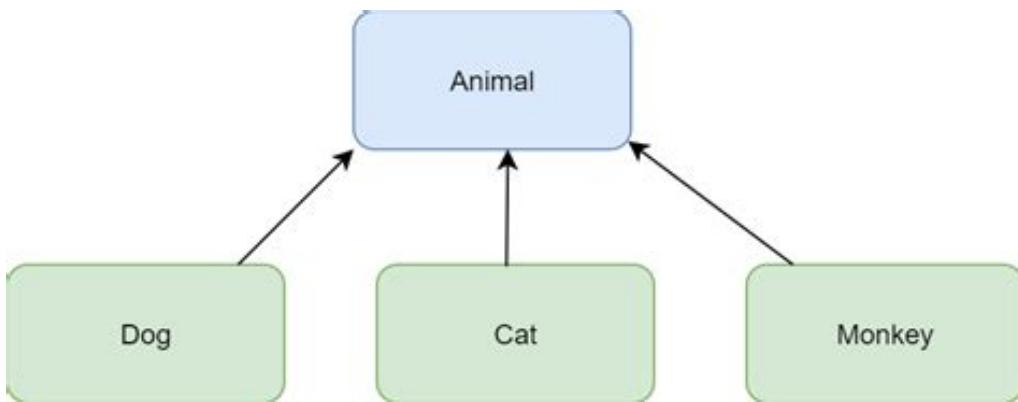


Figura 1.8: Concetto dell'ereditarietà

```
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.

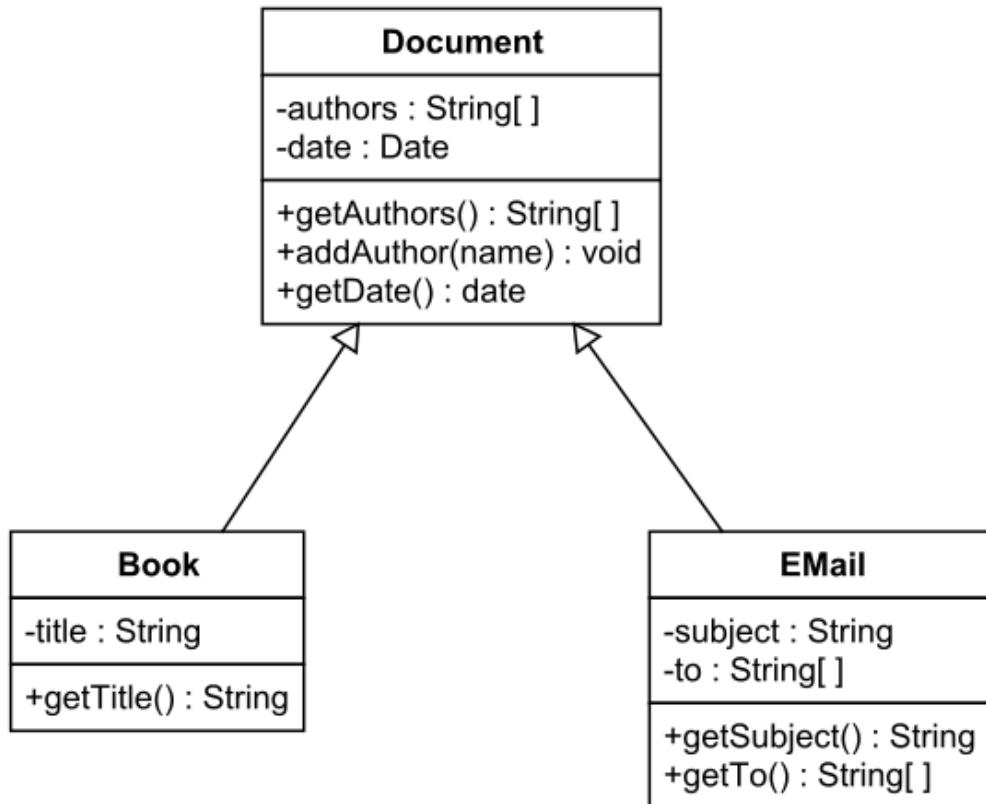


Figura 1.9: Ereditarietà in un diagramma UML

```

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.

```

Board.h  Board.cpp
Miscellaneous Files - No Configurations  Board

8  {
9  public:
10 enum class CellContents
11 {
12     Empty,
13     Obstacle,
14     Food,
15     Poison
16 };
17 public:
18     Board( Graphics& gfx );
19     void DrawCell( const Location& loc, Color c );
20     int GetGridWidth() const;
21     int GetGridHeight() const;
22     bool IsInsideBoard( const Location& loc ) const;
23     CellContents GetContents( const Location& loc ) const;
24     void ConsumeContents( const Location& loc );
25     void SpawnContents( std::mt19937& rng, const class Snake& snake, CellContents contents ); // Non possiamo avere header che
26     // includono altri header in modo circolare, non posso avere "Snake.h" in "Board.h" perché ho già "Board.h" in "Snake.h"
27     // E quindi scriviamo "const class Snake& snake" per dire al compilatore "devi prendere come parametro questa classe Snake
28     // non è necessario specificarla adesso, semplicemente sappi che devi prenderla come parametro."
29     // Poi in "Board.cpp" va tutto bene perché lì abbiamo incluso "Snake.h" e quindi sa cos'è la classe Snake.
30     // Questo è chiamato "Forward Declaration" ed è molto utile, ma se riesci a fare in modo che non ci siano alcune
31     // dipendenze circolari, di solito è meglio.

```

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 = screenWidth - 6; // Avrei potuto scrivere Graphics::ScreenWidth ; Graphics è il nome della classe e
                                // ScreenWidth è una costante intera statica, ed è proprio perché è statica che posso usare la notazione
                                // Graphics::ScreenWidth
    // Guarda sopra nell'if
}

if (top < 0)
{
    y_mobile = 5;
}
else if (bottom >= gfx.ScreenHeight)
{
    static constexpr int screenWidth = 800;
    static constexpr int screenHeight = 600;
};

```

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.

```

Poo::ProcessConsumption(const Dude& dude) // Meglio passarla per referenza, per evitare di copiarla 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 referenza.	È un tipo di dato per valore.
Si dichiara usando la keyword class .	Si dichiara usando la keyword struct .

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 someBMCVariable = 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 addizionali

- 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 ( ... )
{
} 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..

STL | Standard Template Library

Definizione: La Standard Template Library (STL) è un insieme di classi template che forniscono le classiche strutture dati e funzioni come: liste, stacks, arrays, eccetera..

È una libreria di contenitori (containers) di classi, algoritmi e iteratori. È una libreria generalizzata, i suoi componenti sono parametrizzati.

È stata sviluppata separatamente e poi sottoposta al C++ standard committee per essere considerata e per dare loro la possibilità di adottarla nel linguaggio.

L'STL ha quattro componenti:

- Algoritmi
- Contenitori
- Funzioni
- Iteratori

Che cos'è #include <bits/stdc++.h>?

Definizione: È un header file che include tutti gli header files della STL.

Non dovrebbe essere usato perché:

- è *lazy* (pigro).
- è un header del compilatore GCC e quindi **non** è portabile, ovvero potrebbe **non** andare in altri.
- non sai che cosa fa perché i suoi contenuti **non** sono standard.
- ogni header file dovrebbe essere *parsato* il che renderebbe tutto più lento.
- rende il codice **non** portabile.

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? Si e Si! 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. <T>
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> da1( 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> da1 ( 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 rimarebbe 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'ampiamento 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	\rightarrow	$= *i$		$++$	$==, !=$
Output			$*i =$	$++$	
Forward	\rightarrow	$= *i$	$*i =$	$++$	$==, !=$
Bidirectional		$= *i$	$*i =$	$++, --$	$==, !=,$
Random-Access	$\rightarrow, []$	$= *i$	$*i =$	$++, --, +=, -=, +, -$	$==, !=, <, >, <=, >=$

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};
```

```
// Dicho 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 v1 nel vettore v 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.

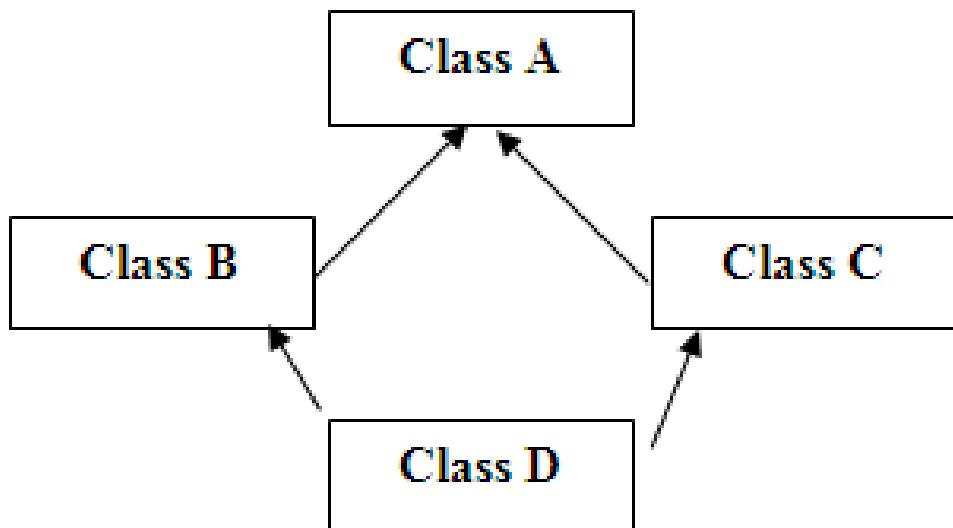


Figura 2.2: Problema del diamante

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.106).

```
#include <iostream>

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

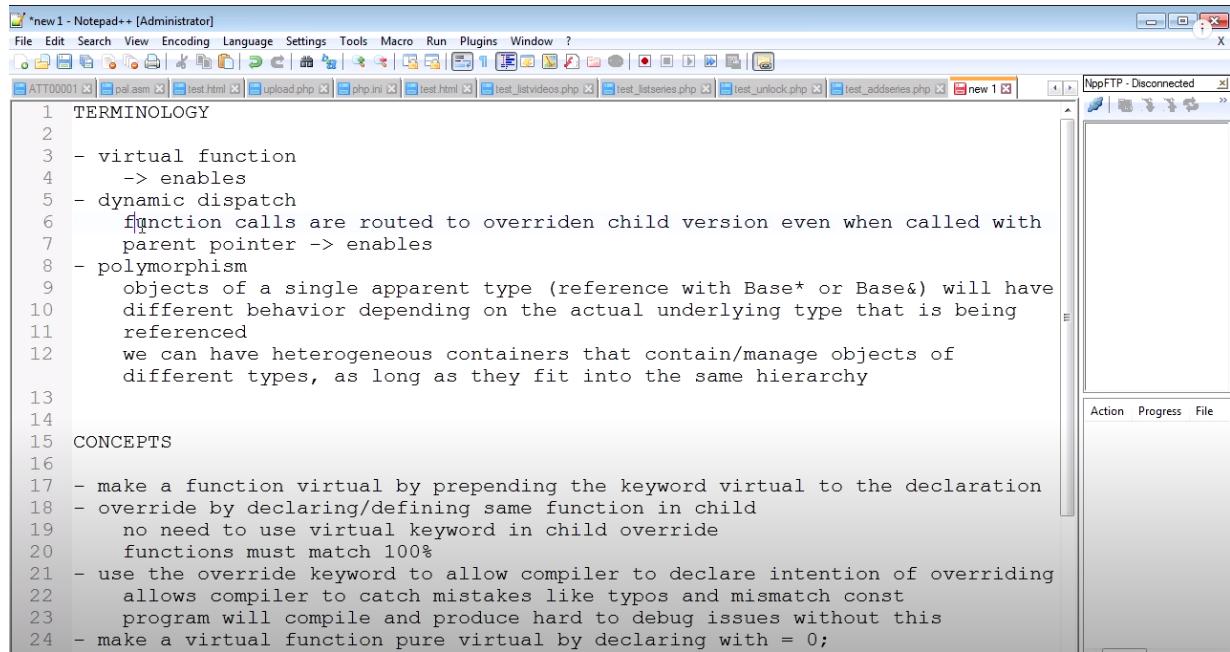
class B : public virtual A {
};

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**.

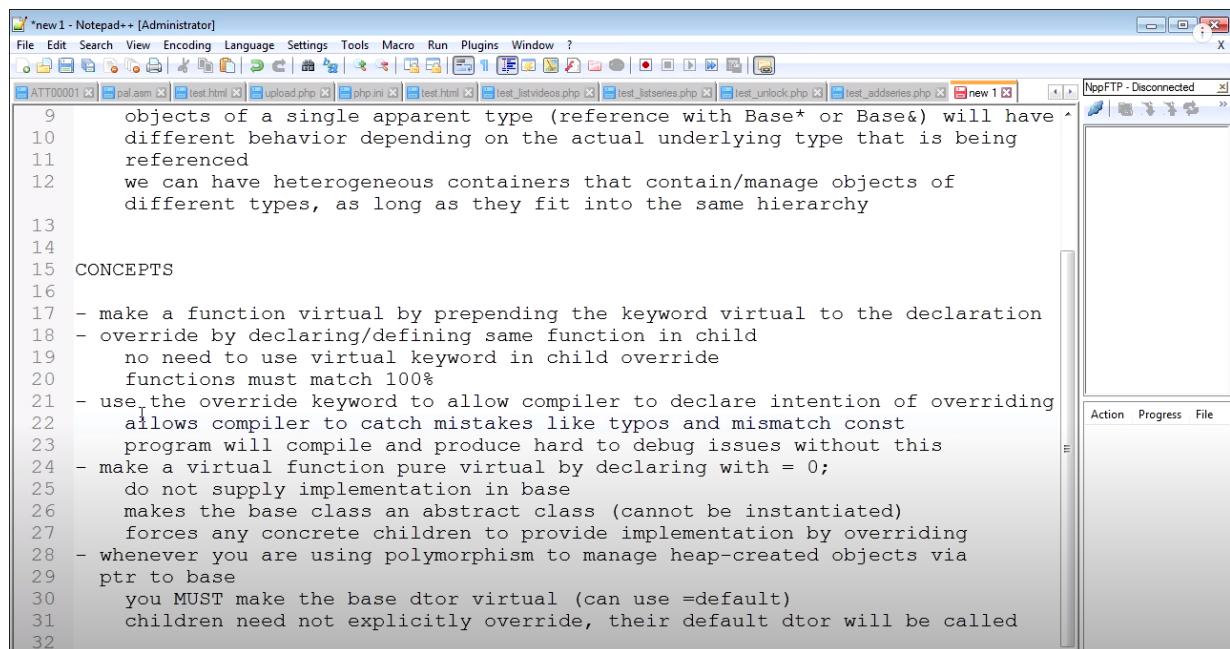


```

1 TERMINOLOGY
2
3 - virtual function
4     -> enables
5 - dynamic dispatch
6     function calls are routed to overridden child version even when called with
7     parent pointer -> enables
8 - polymorphism
9     objects of a single apparent type (reference with Base* or Base&) will have
10    different behavior depending on the actual underlying type that is being
11    referenced
12    we can have heterogeneous containers that contain/manage objects of
13    different types, as long as they fit into the same hierarchy
14
15 CONCEPTS
16
17 - make a function virtual by prepending the keyword virtual to the declaration
18 - override by declaring/defining same function in child
19     no need to use virtual keyword in child override
20     functions must match 100%
21 - use the override keyword to allow compiler to declare intention of overriding
22     allows compiler to catch mistakes like typos and mismatch const
23     program will compile and produce hard to debug issues without this
24 - make a virtual function pure virtual by declaring with = 0;

```

Figura 2.3: Recap Virtual Functions



```

9     objects of a single apparent type (reference with Base* or Base&) will have
10    different behavior depending on the actual underlying type that is being
11    referenced
12    we can have heterogeneous containers that contain/manage objects of
13    different types, as long as they fit into the same hierarchy
14
15 CONCEPTS
16
17 - make a function virtual by prepending the keyword virtual to the declaration
18 - override by declaring/defining same function in child
19     no need to use virtual keyword in child override
20     functions must match 100%
21 - use the override keyword to allow compiler to declare intention of overriding
22     allows compiler to catch mistakes like typos and mismatch const
23     program will compile and produce hard to debug issues without this
24 - make a virtual function pure virtual by declaring with = 0;
25     do not supply implementation in base
26     makes the base class an abstract class (cannot be instantiated)
27     forces any concrete children to provide implementation by overriding
28 - whenever you are using polymorphism to manage heap-created objects via
29     ptr to base
30     you MUST make the base dtor virtual (can use =default)
31     children need not explicitly override, their default dtor will be called
32

```

Figura 2.4: Recap Virtual Functions

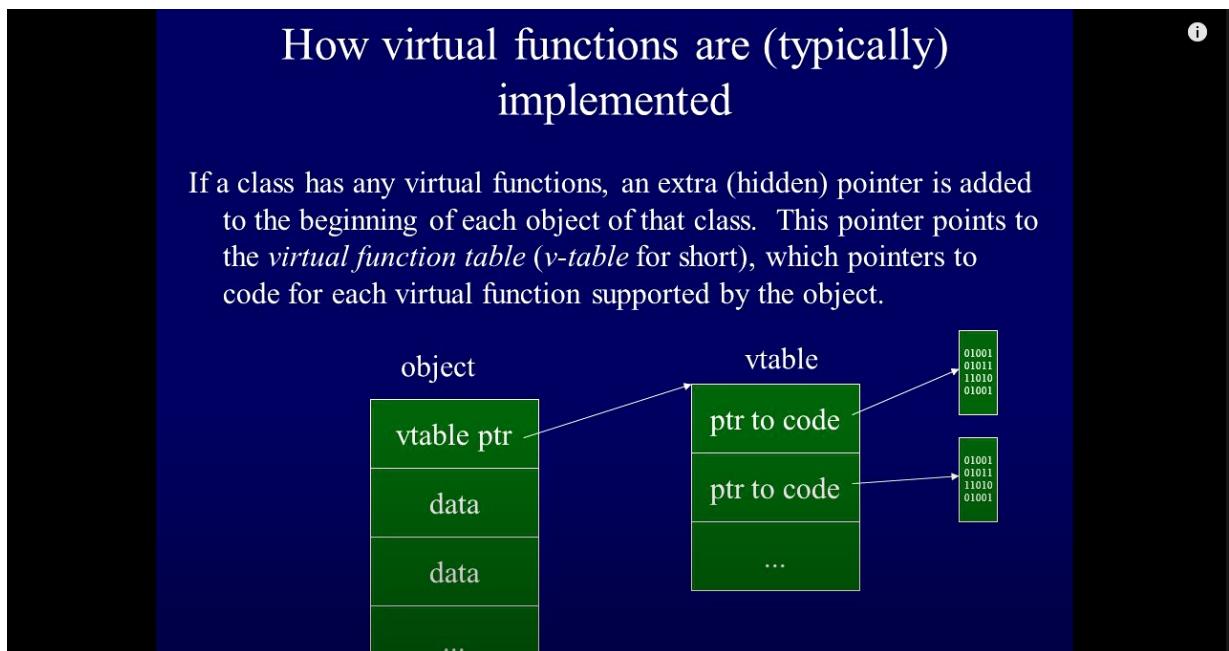


Figura 2.5: Virtual Functions Implementazione

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*).

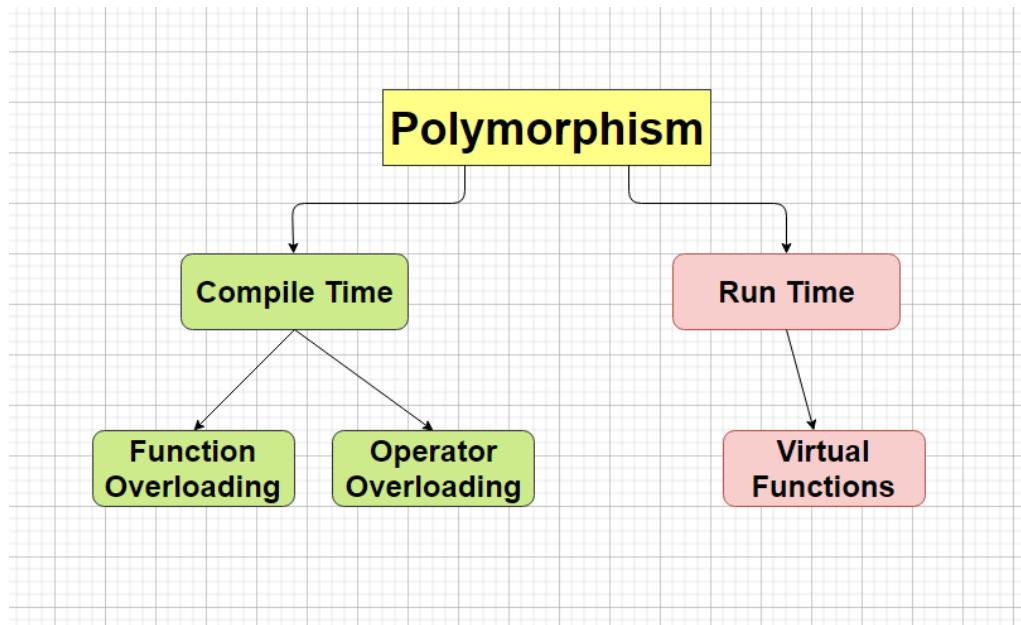


Figura 2.6: Polimorfismo

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 *overloaddirlo*.

```

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 *overloaddare*.

Gli operatori che non si possono *overloaddare* 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 override .
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 overriden è preceduta dalla keyword virtual 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 virtual 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 riporterai 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
dynamic_cast	per convertire puntatori/references in una gerarchia di ereditarietà.
static_cast	per le conversioni di tipi ordinari.
reinterpret_cast	per reinterpretare bit patterns di basso livello. Usare con cautela.
const_cast	per aggiungere/rimuovere const al cast.

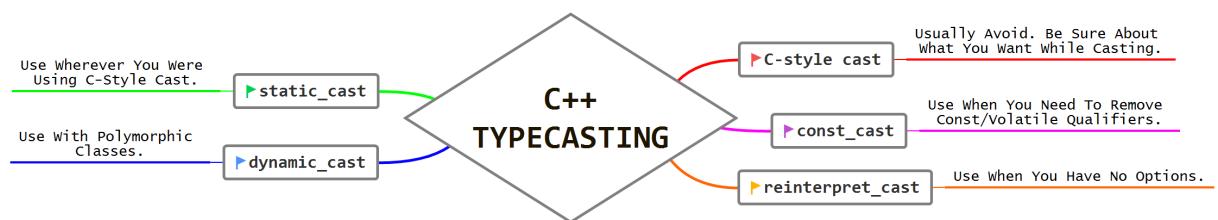


Figura 2.7: Typecasting

```

static_cast
  - converts types (int->float etc.)
  - casts pointers/references within an inheritance hierarchy (no checking!)
dynamic_cast
  - cast pointers/references within an inheritance hierarchy (dynamic check!)
  - requires that the types have at least one virtual function
  - if check fails, nullptr for ptr* cast, throw exception for ref& cast
reinterpret_cast
  - reinterpret pointer/references as pointing to some other type (no
    check, no limit!) (only really safe for char*)
  - reinterpret pointer as an integral value
const_cast
  - remove constness

|
c-style cast (int) etc.
  - can do all of the above, even if you don't mean to!
  const Base* pBase = &obj;
  ...
  Derived* pDerived = (Derived*)pBase;
  - casted away the constness by accident!

```

Figura 2.8: Recap Cast Types

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
<code>[]</code>	accedere solo alla variabili locali
<code>[=]</code>	accedere a tutte le variabili per valore.
<code>[&]</code>	accedere a tutte le variabili per reference.
<code>[nomeVariabile1, &nomeVariabile2]</code>	"cattura" nomeVariabile per valore e nomeVariabile2 per referenza.

```
#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
maggiori 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 usufruivamo di 4 diverse funzioni: **malloc()** (per allocare), **calloc()**, **realloc()** (per riallocare), **free()** (per liberare la memoria).

Tutte questi 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: RAII (*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.

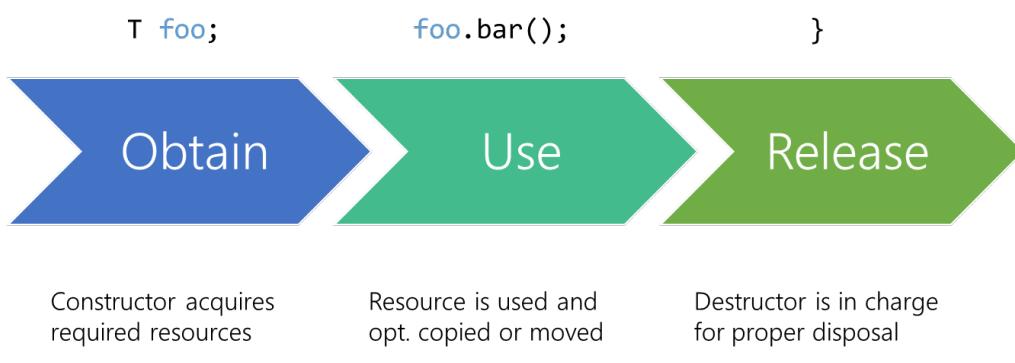


Figura 2.9: RAII

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.

Preventing Object Copy

Definizione: Serve per non permette ad un'istanza di una classe di essere copiata.

Per fare questo ci sono vari modi:

1. Porre il **copy constructor** ed il **copy assignment** privati
2. Ereditare una classe con un **copy constructor** e un **copy assignment operator** privati
3. Usare i **copy constructor = delete** e il **copy assignment operator = delete**

```

#include <iostream>

class Base {
    int x;
    public:
    Base() {}
    Base(int y) : x(y) {}
}

```

```

Base( const Base& temp_obj ) = delete;
Base& operator=( const Base& temp_obj ) =
    delete;
};

int main()
{
    Base b1(10);
    Base b2(b1); // Chiama il copy constructor, da errore
                  perché abbiamo messo = delete per non permettiamo la
                  copiatura di oggetti.
    b2 = b1; // Chiama il copy assignment operator, da errore
              perché abbiamo messo = delete per non permettiamo la
              copiatura di oggetti.
    return 0;
}

```

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 bitweise 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 sono allocati con memoria dinamica, in quel caso faranno riferenza 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 Constructors 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: " << rref << "\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: " << rref << "\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

Figura 2.10: 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 classi che implemetteranno 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 classi derivate.

Si dichiara una **pure virtual function** assegnando 0 alla dichiarazione.

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

Pure Virtual Destructor

Definizione: È possibile in C++ dichiarare un distruttore come **virtuale** e lasciare che le classi derivate si occupano della sua implementazione.

Solo i distruttori possono essere dichiarati **virtuali**, i costruttori no.

```
#include <iostream>

class Base {
public:
    virtual ~Base() = 0; // Pure virtual destructor
};

Base::~Base() {
    std::cout << "Chiamato il distruttore
    virtuale puro" << std::endl;
```

```

}

class Derived : public Base {
public:
    ~Derived() {
        std::cout << "Distruttore della
                     classe derivata" << std::endl;
    }
};

int main()
{
    Base *bPtr = new Derived();
    delete bPtr;
    //Output: Distruttore della classe derivata
    //Output: Chiamato il distruttore virtuale puro
    return 0;
}

```

Abstract Class

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

- Possono avere funzioni normali e allo stesso tempo **pure virtual functions**.
- Non può essere instanziata, 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.

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

noexcept

Definizione: La keyword **noexcept** è un operatore unario che valuta se l'operando può propagare un'eccezione. È un metodo per il programmatore di informare il compilatore se una funzione può o no lanciare eccezioni.

noexcept può produrre dei falsi negativi.

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

class Sigma {
public:
    Sigma()
    {
        std::cout << "Costruendo Sigma" <<
        std::endl;
    }

    Sigma( Sigma&& ) noexcept
    {
        std::cout << "Moving Sigma" << std::
        endl;
    }
}
```

```
    }

Sigma( const Sigma& )
{
    std::cout << "Copying Sigma" << std::
                    endl;
}
;

int main()
{
    std::vector<Sigma> sigmas;
    sigmas.emplace_back();
    sigmas.emplace_back();
    sigmas.emplace_back();

    //Output: Costruendo Sigma
    //Output: Costruendo Sigma
    //Output: Moving Sigma
    //Output: Costruendo Sigma
    //Output: Moving Sigma
    //Output: Moving Sigma

    return 0;
}
```

```

Exceptions!
=====

DO inherit from std::exception

DO use exceptions for errors that happen that are
    out of programmer control
DO NOT use exceptions for sanity checks (use assert)

DO throw by value
DO catch by const ref&

DO use RAII types like smart pointers ALL THE TIME

DO NOT throw exceptions from a dtor

DO NOT throw exceptions from a move ctor if you want
    the move optimization in a std::vector etc.

DO mark functions as noexcept if you are confident that
    they will NEVER want to throw an exception, either
    now, or later on in development

DO NOT mark functions noexcept just because they happen
    to not throw right now--they might in the future so
    think carefully before going down the noexcept path

```

Figura 2.11: Recap Eccezioni

Assertions

Definizione: Le **assertions** sono delle dichiarazioni usate per testare delle assunzioni fatte dal programmatore.

Se la condizione nell'**assert** fosse valutata falsa, allora il programma fermerebbe l'esecuzione.

```

#include <assert.h>

int length = 7;

// Un assert per controllare che la variabile length sia maggiore o
// uguale a 0.
assert(length >= 0);

```

```
// Se l'assert fosse valutato 'false' il programma si fermerebbe e
// verrebbe mostrato quel messaggio.
assert(length >= 0 && "La lunghezza non puo'
    essere negativa");

int x = 0;

// In questo caso verrebbe fermato il programma e verrebbe
// mostrato il messaggio "x deve essere maggiore di 0"
assert(x > 0 && "x deve essere maggiore di 0");
```

Operazioni di Input/Output

Definizione: La libreria standard del C++ mette a disposizioni diverse funzioni per le operazioni di input ed output di dati in streams (flussi di dati), in files, ecc...

Input-Output stream

Definizione: Le **stream** sono sequenze, flussi di dati. Quando i dati provengono da dispositivi come tastiere, hard disk, connessioni network sono delle operazioni di **input**. Quando, invece i dati provengono da dispositivi come schermi, stampanti, disco rigido o una connessione network sono chiamati operazioni di **output**.

Il file di intestazione **<iostream>** definisce le funzioni come **cin**, **cout**, **cerr** e **clog** che corrispondono allo *standard input*, *standard output*, *un-buffered standard error* e *buffered standard error stream*.

Per quanto riguarda lo *standard output* i dati vengono inseriti in esso attraverso l'operatore d'inserimento **«»**.

È possibile fare questo *chaining* di « perché l'operatore d'inserzione ritorna una referenza a cout (**cout&**).

```
#include <iostream>

int main()
{
    int x = 5;
    int y = 8;
    std::cout << "Valore di x: " << x << ","
           valore di y: " << y << std::endl; // Output:
           Valore di x: 5, valore di y: 8
    return 0;
}
```

Tramite **cin** chiediamo l'input dall'utente. In questa caso usiamo l'operatore di estrazione **»**.

```
#include <iostream>
```

```

int main()
{
    int age;

    std::cout << "Inserisci l'età: ";
    std::cin >> age;
    std::cout << "\n La tua età è: " << age <<
        std::endl;
    return 0;
}

```

`cerr` è usato per mostrare degli errori sullo schermo. Un-buffered vuol dire che il messaggio non può essere immagazzinato, viene immediatamente mostrato sullo schermo.

```

#include <iostream>

int main()
{
    std::cerr << "Si è verificato un errore!";
    return 0;
}

```

Anche `clog` come `cerr` mostra un errore, ma prima lo immagazzina all'interno di un buffer finché non è pieno o viene liberato (usando `flush()`) e poi verrà mostrato sullo schermo.

```

#include <iostream>

int main()
{
    std::clog << "Si è verificato un errore";
    return 0;
}

```

std::endl *vs* ***newline***

Qual è la differenza tra `std::endl` ed il carattere di escape newline `\n`?
`\n` va a capo, mentre `std::endl` va anch'esso a capo, ma in più ripulisce il buffer, di solito non hai sempre bisogno di farlo e questo ti costerà in termini di *performance*.

```
#include <iostream>

std::cout << std::endl;
// std::cout « std::endl è equivalente a:
std::cout << '\n' << std::flush;
// Mentre "\n" inserisce semplicemente una nuova riga.
std::cout << "\n";
```

std::endl	\n
È un manipolatore.	È un carattere.
Non occupa spazio in memoria.	Occupava un byte visto che è un carattere.
È una keyword e non ha significato se la si memorizza in una stringa.	Può essere memorizzata in una stringa e conserva il suo significato.
Non possiamo scrivere 'endl' in apici.	Possiamo scrivere sia '\n' sia "\n".
È supportata solo in C++.	È supportata sia in C che in C++.
Continua a ripulire il buffer.	Ripulisce il buffer solo alla fine del programma.

Quindi, in conclusione, usare \n sembra migliore in termini di prestazioni rispetto std::endl a meno che non sia necessario ripulire il buffer dello stream.

Manipolazione degli stream

Definizione: I manipolatori sono funzioni che permettono di manipolare l'input/output dello stream. Non vuol dire che modifichiamo la variabile, modifica soltanto lo stream.

Ci sono diversi tipi di manipolatori:

1. Manipolatori senza argomenti:

- **endl** : serve per andare a capo e dopo averlo fatto ripulisce il buffer dello stream.
- **ws** : definito nell'**istream** e usato per ignorare gli spazi bianchi nella stringa dello stream.
- **ends** : inserisce un *null character* nello stream.
- **flush** : definito nell'**ostream** e serve per ripulire l'output stream.

```
#include <iostream>
#include <iostream>
#include <sstream>
#include <string>
```

```

int main()
{
    // istringstream da <sstream>
    std::istringstream str("    Hello World! ");

    std::string line;

    // Ignora tutti gli spazi bianchi (con std::ws) prima della
    // prima parola.
    std::getline( str >> std::ws, line );

    std::cout << line << std::endl;

    std::cout << "Test" << std::flush;

    std::cout << "A" << "\n";
    std::cout << "B" << std::ends;
    std::cout << "C" << std::endl;
    return 0;
}

```

1. Manipolatori con argomenti:

a) Alcuni importanti manipolatori in <iomanip>:

- **setw(val)** : usato per impostare la larghezza dello spazio negli operatori di output.
- **setfill(c)** : usato per riempire lo stream di output con il carattere 'c'.
- **setprecision(val)** : Imposta val come nuovo valore di precisione dei valori floating-point.
- **setbase(val)** : Per impostare la base dei valori numerici. (es: base 2 (binario), base 16, esadecimale, base 8, ottale, ecc...)
- **setioflags(flag)** : Usato per impostare i flag di formato specificati dal parametro di maschera.
- **resetiosflags(m)** : Usato per resettare i flags di formato specificati dal parametro di maschera.

b) Alcuni manipolatori in <ios> :

- **showpos** : Mostra il segno + nei numeri positivi.
- **noshowpos** : Non mostra il segno + nei numeri positivi.
- **showbase** : Indica la base numerica usati per i valori numerici.
- **uppercase** : Mostra le lettere in maiuscolo.
- **nouppercase** : Mostra le lettere in minuscolo.
- **fixed** : Usa la notazione decimale per i valori floating-point.
- **scientific** : Usa la notazione scientifica per i numeri in virgola mobile (floating-point).
- **hex** : Legge e scrive valori in esadecimale per i numeri interi e funziona come `setbase(16)`.
- **dec** : Legge e scrive valori in decimale per i numeri interi, funziona come `setbase(10)`.
- **oct** : Legge e scrive i valori in ottale per i numeri interi, funziona come `setbase(8)`.
- **left** : Aggiusta l'output verso sinistra.
- **right** : Aggiusta l'output verso destra.

```
#include <iostream>
#include <iomanip>

int main()
{
    int a = 100;
    // Imposto la formattazione.
    std::cout << std::hex << std::left << std::
        showbase << std::nouppercase;

    std::cout << a << std::endl; // Output: 0x64
    // (esadecimale per 100)

    double b = 2001.5251;

    // Imposto la formattazione dello stream.
    std::cout << std::setbase(10) << std::right
        << std::setw(15)
        << std::setfill('_') << std::showpos
        << std::fixed << std::setprecision(2);
```

```

// Mostro la variabile b
std::cout << b << std::endl; // Output:
                               +2001.53

// Formattazione.
std::cout << std::scientific << std::
           uppercase
<< std::noshowpos << std::setprecision(9);

double c = 201455.2646;

// Stampo la variabile c.
std::cout << c << std::endl; // Output:
                               2.014552646E+05

return 0;
}

// Per questi serve includere <iomanip>
std::cout << std::setw(7) << 23 << std::setw(7) << 123 << std::setw(7) << 3 << std::setw(7) << 223 << std::endl;
std::cout << std::setw(7) << 3 << std::setw(7) << 123 << std::setw(7) << 3 << std::setw(7) << 223 << std::endl;
std::cout << std::setw(7) << 234 << std::setw(7) << 12333 << std::setw(7) << 23 << std::setw(7) << 223 << std::endl;
// setw sta per set width, quindi settiamo una larghezza fissa per l'output, se metto qualcosa che è minore di quello
// che c'è fra le parentesi di setw allora aggiungerà del padding

```

Figura 2.12: Stream iomanip

Operazioni su file

Definizione: Per le operazioni di Input/Output sui files avremo bisogno di includere **fstream**.

Ci sono diverse modalità di apertura di un file:

Costante membra	Significato	Accesso
in*	input	File aperto per la lettura: il buffer interno supporta le operazioni di input.
out	output	File aperto per la scrittura.
binary	binary	Le operazioni vengono eseguite
ate	at end	La posizione di output parte dalla fine.
app	append	Parte dalla fine e aggiunge contenuti a quelli già presenti.
trunc	truncate	Tutti i contenuti che esistevano prima di essere aperto vengono scartati.

Modalità di apertura di default:

Modalità di apertura di default	
ifstream	ios::in
ofstream	ios::out
fstream	ios::in ios::out

```
// Esempio 1
#include <iostream>
#include <fstream>

int main()
{
    std :: ofstream myfile;
    myfile.open( "nomefile.txt" );
    myfile << "Scrivere questa stringa nel file.\n";
    myfile.close();
    return 0;
}
```

```
// Esempio 2
#include <iostream>
#include <fstream>

int main()
{
    std :: string line;
    std :: ifstream myfile( "nomefile.txt" );
```

```

if (myfile.is_open())
{
    while (std::getline(myfile, line))
    {
        std::cout << line << '\n';
    }
    myfile.close();
} else {
    std::cout << "Impossibile aprire il file"
    << '\n';
}
return 0;
}

```

P.S. Ricordatevi di chiudere il file una volta che avete finito di utilizzarlo.

Uno dei modi per ottenere lo spazio occupato da un file:

```

// serve includere <iostream>
std::ifstream in("derp.txt");
if (!in)
{
    printf("Failed to open file!");
    while (!_kbhit());
    return -1;
}

in.seekg(0, std::ios_base::end); // in.seekg(0, std::ios_base::cur);
const int length = in.tellg(); // get absolute position of the get pointer in the file, get number of bytes in the file

printf("\n File Size: ");
char buffer[256];
std::to_string(length);
printf( "%s \n",&buffer );

in.seekg(0, std::ios_base::beg); // beginning of the file

```

Figura 2.13: Ottenere il file size

```

char buffer[256];
// Così otteniamo lo spazio che occupa il file.
std::ifstream warp_file(buffer);
warp_file.seekg(0, std::ios_base::end); // seek to the end of the file
const int file_size = warp_file.tellg();
warp_file.seekg(0, std::ios_base::beg); // seek to the beginning of the file
char* warp_string = new char[file_size + 1]; // Dynamic array memory allocation; Quel + 1 è just in case ci serva
// più spazio
delete [] warp_string; // Dealloco la memoria
warp_string = nullptr; // Pongo il puntatore a null

```

Figura 2.14: Ottenere il file size

std::chrono

Definizione: La libreria **chrono** è usata per occuparsi delle date e del tempo. Questa libreria è progettata col fatto che i timers e gli orologi potrebbero essere diversi su sistemi operativi differenti.

Fornisce una *precisione neutrale*, attraverso la separazione le durate e il tempo. Tutti gli elementi di questo header non sono definiti sotto il namespace std, ma sotto il **std::chrono namespace**.

Si occupa del tempo attraverso principalmente 3 concetti:

- **Duration (Durata)** : Un oggetto durata esprime un intervallo di tempo come 3 minuti, 3 ore, 3 millisecondi, ecc.. Per esempio 33 secondi potrebbe essere rappresentato dalla durata di 33 ticks ciascuno di 1 secondo.
- **Clock (Orologio)** : Un orologio consiste in un punto di partenza (o epoch) e un tick rate.
- **Time point (Punto nel tempo)** : Un oggetto di tipo *time_point* esprime un punto nel tempo relativo ad un epoch di un clock. Internamente, l'oggetto memorizza un oggetto di tipo durata (*duration*) e usa il clock come riferimento per via del suo epoch.

Duration

```
#include <iostream>
#include <chrono>

int main ()
{
    using namespace std :: chrono ;

    // std::chrono::milliseconds è una
    // instanziazione di std::chrono::duration<- 1 secondo

    milliseconds mil(1000) ;
```

```

mil = mil*60;

std::cout << "durata (in periodi): ";
std::cout << mil.count() << " millisecondi.\n"
"; // Output: durata (in periodi): 60000 millisecondi.

std::cout << "durata (in secondi): ";
std::cout << (mil.count() * milliseconds::
    period::num /
milliseconds::period::den);
std::cout << " secondi.\n"; // Output: durata (in
secondi): 60 secondi.

return 0;
}

```

Clock

Ci sono 3 diversi tipi di clock (orologi):

- **system_clock** : È il tempo corrente in base al sistema operativo (quello che vediamo nella barra dei comandi). È scritto come `std::chrono::system_clock`.
- **steady_clock** : È un orologio monotonic (monotonico, uniforme) che non può essere aggiustato, regolato. Va ad un rate uniforme. È scritto come `std::chrono::steady_clock`.
- **high_resolution_clock** : Fornisce il periodo di tick più piccolo possibile. È scritto come `std::chrono::high_resolution_clock`.

Time point

Calcolare il tempo di esecuzione di un blocco di codice

```

// Esempio 1 Implementazione 1 per calcolare il tempo che un
// blocco di codice ci mette per essere eseguito.
#include <iostream>
#include <chrono>
#include <ctime>

```

```

long fibonacci(unsigned n)
{
    if (n < 2) return n;
    return fibonacci(n-1) + fibonacci(n-2);
}

int main()
{
    // time point e system_clock
    std :: chrono :: time_point<std :: chrono :: system_clock> start , end;

    start = std :: chrono :: system_clock :: now();
    std :: cout << "f(42) = " << fibonacci(42) << '\n';
    // Output: f(42) = 267914296
    end = std :: chrono :: system_clock :: now();

    std :: chrono :: duration<double> elapsed_seconds
        = end - start;
    std :: time_t end_time = std :: chrono :: system_clock :: to_time_t(end);

    std :: cout << "Finito di eseguire il " << std
        :: ctime(&end_time)
        << "tempo passato: " << elapsed_seconds . count
        () << "s \n"; // Output: Finito di eseguire il Sun Mar
        13 22:53:59 2022 elapsed time: 2.71703s
}

```

Esempio 2 Implementazione 2 per calcolare il tempo di esecuzione di un blocco di codice:

```

using std::chrono::steady_clock; // Questo using è locale a questa porzione di codice, non all'intero file.
// ovviamente devo includere chrono per poterlo utilizzare #include <chrono>
std::chrono::steady_clock::time_point start = std::chrono::steady_clock::now(); // lo steady clock è meglio per
// misurare le performance rispetto al system clock
for (int y = 0; y < Graphics::ScreenHeight; y++)
{
    for (int x = 0; x < Graphics::ScreenWidth; x++)
    {
        gfx.PutPixel(x, y, Colors::Green);
    }
}
steady_clock::time_point end = std::chrono::steady_clock::now(); // salviamo il tempo finita l'operazione
std::chrono::duration<float> runtime = end - start; // Calcoliamo il tempo totale per eseguire quell'operazione
float durationSecond = runtime.count();

```

Figura 2.15: Calcolo tempo trascorso per eseguire un blocco di codice

Generatori di numeri pseudo-casuali

Definizione: I generatori di numeri pseudo-casuali ci permettono di generare dei numeri che sembrano casuali.

Numeri casuali come in C

In C e di conseguenza in C++ si può utilizzare `rand()` e `srand()` importando `stdlib.h` in C e `<cstdlib>` in C++.

```

// Esempio 1
#include <iostream>
#include <cstdlib>

// Questo programma genererà sempre la stessa sequenza di
// numeri ad ogni esecuzione.
int main()
{
    int number;
    for( int i = 0; i < 3; i++)
    {
        number = rand();
        std::cout << number << std::endl;
    }
}

```

```

    return 0;
}

```

Per generare un numero compreso in un range (raggio: tra due numeri) usiamo l'operatore modulo e il numero a cui può arrivare.

```

// Esempio 2
#include <iostream>
#include <cstdlib>

// Questo programma genererà un numero nel range 25 e 50.
int main()
{
    int max = 50;
    int min = 25;
    int range = max - min + 1;

    int num = rand() % range + min;
    return 0;
}

```

Definizione: Il **seed** (seme) è il punto d'inizio della sequenza, può essere cambiato affinché la sequenza di numeri generati sia diversa.

Nel prossimo esempio usiamo il tempo corrente per generare un **seed** (un seme). Usiamo **srand** per impostare il seme.

```

// Esempio 3
#include <iostream>
#include <cstdlib>
#include <time.h>

// Questo programma genererà un numero nel range 25 e 50.
int main()
{
    constexpr int NUM_OF_NUMBERS = 3;

    int number;

    // Usiamo srand per settare il seed ed usiamo il tempo
    // corrente in questo caso come parametro del seed.
    std::srand(time(0));
}

```

```

for( int i = 0; i < NUM_OF_NUMBERS; i++)
{
    number = rand();
    std::cout << number << std::endl;
}
return 0;
}

```

Cosa vuol dire e perché pseudo-random?

Gli **pseudorandom number generator** (*PRNG*), anche chiamati **deterministic random bit generator** (*DRBG*, ovvero generatori di bit casuali deterministici) sono degli algoritmi per generare sequenze di numeri le quali proprietà sono simili a quelle delle sequenze di numeri casuali.

I *PRNG* non generano vere sequenze casuali perché è completamente determinato da un numero di partenza chiamato *seed* (*seme*).

Quindi è chiamato "pseudo" casuale perché l'algoritmo può ripetere la sequenza e quindi i numeri non sono propriamente casuali.

I vari tipi di generatori

Definizione: Nell'header `<random>` sono contenuti diversi generatori di numeri pseudo-casuali e diverse distribuzioni:

- **Generatori**: oggetti che generano numeri distribuiti uniformemente.
- **Distribuzioni**: oggetti che trasformano sequenze di numeri generati da un generatore in sequenze di numeri che seguono una specifica distribuzione variabili e casuale, come: uniforme, normale, binomiale.

I **Pseudo-random number engines**: usano un algoritmo per generare numeri casuali basati su un seme iniziale. Questi sono:

1. **linear_congruential_engine**: è il più semplice nella **STL** (*Standard Template Library*) che genera numeri interi senza segno.

linear_congruential_engine	Implements linear congruential algorithm
mersenne_twister_engine	Implements Mersenne twister algorithm
subtract_with_carry_engine	Implements subtract with a carry algorithm

Figura 2.16: Motori di numeri pseudo-casuali

```
#include <iostream>
#include <chrono>
#include <random>

int main()
{
    // Trova il tempo tra il clock del sistema (tempo corrente) e
    // l'epoch del clock (data e tempo relativi quando il
    // timestamp di un clock di un computer sono determinati).
    unsigned seed = std::chrono::system_clock::
        now().time_since_epoch().count();

    std::minstd_rand0 generator(seed);

    std::cout << generator() << "è un numero
casuale tra ";
    std::cout << generator.min() << " e " <<
    generator.max();
    return 0;
}
```

2. **mersenne_twister_engine** : È un motore di numeri casuali basati sull'algoritmo *Mersenne Twister*. Produce dei numeri senza segno di alta qualità nell'intervallo $[0, (2^w) - 1]$, dove $w = \text{size della parola (word)}$.

// Esempio 1

```

#include <iostream>
#include <chrono>
#include <random>

int main()
{
    unsigned seed = std::chrono::system_clock::
        now().time_since_epoch().count();

    std::mt19937 generator(seed);

    std::cout << generator() << " è un numero
casuale tra ";

    std::cout << generator.min() << " e " <<
        generator.max();
    return 0;
}

// Esempio 2
#include <iostream>
#include <chrono>
#include <random>

int randomNumberGenerator( int limit )
{
    // generatore di numeri interi casuali con distribuzione
    // uniforme che produce numeri casuali non deterministici.
    std::random_device rd;

    // Generatore pseudo-random con l'algoritmo Mersenne
    // Twister di numeri a 32 bit con uno spazio di stato di 19937
    // bits.
    std::mt19937 rng( rd() );

    // Distribuzione uniforme
    std::uniform_int_distribution<> dis(1, limit)
        ;
    return dis( rng );
}

```

```

int main()
{
    int limit = 100;
    std::cout << randomNumberGenerator(limit) <<
        std::endl;
    return 0;
}

```

3. **subtract_with_carry_engine** : È un motore di generazione di numeri pseudo-casuali che produce numeri interi senza segno. L'algoritmo usato è il *generatore di fibonacci* con una sequenza di stati di r elementi, più uno per il carry (il resto).

```

#include <iostream>
#include <chrono>
#include <random>

int main()
{
    unsigned seed = std::chrono::system_clock::
        now().time_since_epoch().count();

    std::subtract_with_carry_engine<unsigned, 24,
        10, 24> generator(seed);

    std::cout << generator() << " è un numero
casuale tra ";

    std::cout << generator.min() << " e " <<
        generator.max();
    return 0;
}

```

II **Random number generator** : È un generatore di numeri casuali che produce numeri casuali non deterministici.

```

#include <iostream>
#include <random>

int main ()

```

```

{
    std::random_device rd;

    std::cout << "default random_device
                  caratteristiche:" << std::endl;

    std::cout << "minimo: " << rd.min() << std::
                  endl;

    std::cout << "massimo: " << rd.max() << std::
                  endl;

    std::cout << "entropia: " << rd.entropy() <<
                  std::endl;

    std::cout << "un numero casuale: " << rd() <<
                  std::endl;

    return 0;
}

```

III Pseudo-random number engines (instantiations): Questi sono particolari instanze di motori di generatori e adattatori:

1. **default_random_engine**: Questo è una classe che genera numeri pseudo-casuali.

```

#include <iostream>
#include <chrono>
#include <random>

int main()
{
    unsigned seed = std::chrono::system_clock::
        now().time_since_epoch().count();

    // minstd_rand0 è un standard linear_congruential_engine
    std::minstd_rand0 generator(seed);

```

<code>default_random_engine</code>	Default random engine
<code>minstd_rand</code>	Minimal standard minstd_rand generator
<code>minstd_rand0</code>	Minimal standard minstd_rand0 generator
<code>mt19937</code>	Mersenne twister 19937 generator
<code>mt19937_64</code>	Mersenne twister 19937 generator(64-bit)
<code>ranlux24_base</code>	Ranlux 24 base generator
<code>ranlux48_base</code>	Ranlux 48 base generator
<code>ranlux24</code>	Ranlux 24 generator
<code>ranlux48</code>	Ranlux 48 generator
<code>knuth_b</code>	Knuth-B generator

Figura 2.17: Motori di numeri pseudo casuali

```

    std :: cout << generator() << " è un numero
casuale tra ";
    std :: cout << generator.min() << " e " <<
    generator.max();
    return 0;
}

```

2. `minstd_rand`: Genera numeri pseudo casuali. È simile al linear congruential generator.

```

#include <iostream>
#include <chrono>
#include <random>

int main()
{
    unsigned seed = chrono::system_clock::now() .
        time_since_epoch().count();

    // minstd_rand0 è un standard
    // linear_congruential_engine
    std::minstd_rand0 generator(seed);

    std::cout << generator() << " è un numero
casuale tra ";

    std::cout << generator.min() << " e " <<
    generator.max();

    return 0;
}

```

3. **mt19937**: È il generatore *Mersenne Twister* 19937. È un generatore di numeri a 32 bit con uno spazio di stato di 19937 bits.
4. **ranlux24_base**: È un generatore *Ranlux base 24*. È un generatore pseudo casuale che sottrae con il resto di numeri a 24-bit, generalmente usato come base per il generatore *ranlux24*.

```

#include <iostream>
#include <chrono>
#include <random>

int main ()
{
    unsigned seed = std::chrono::system_clock ::
        now().time_since_epoch().count();
    std::subtract_with_carry_engine<unsigned
        ,24,10,24> generator(seed);

```

```

    std::cout << generator() << " è un numero
casuale tra ";
    std::cout << generator.min() << " e " <<
generator.max();
    return 0;
}

```

IV Engine Adaptors

1. **discard_block_engine**: È una classe template che adatta un motore di generatore di numeri pseudo-casuali attraverso il solo utilizzo di 'r' elementi di ogni blocco e 'p' elementi dalla sequenza che produce, scartando il resto. L'adattatore tiene un contatore interno per tenere traccia di quanti elementi sono stati prodotti nel blocco corrente.

I generatori **ranlux24** e **ranlux48** adattano un **subtract_with_carry_engine** usando questo adattatore.

```

#include <iostream>
#include <chrono>
#include <random>

int main ()
{
    unsigned seed = std::chrono::system_clock::
now().time_since_epoch().count();

    // ranlux24 è un standard instantiation
    // del discard_block_engine:
    std::ranlux24 generator(seed);

    std::cout << generator() << " è un numero
casuale tra ";

    std::cout << generator.min() << " e " <<
generator.max();

    return 0;
}

```

2. **independent_bits_engine** : È un adattatore che adatta un motore di generatore di numeri casuali per produrre numeri con uno specifico numero di bits (w).

```
#include <iostream>
#include <chrono>
#include <cstdint>
#include <random>

int main ()
{
    unsigned seed = std::chrono::system_clock::
        now().time_since_epoch().count();

    // independent_bits_engine
    std::independent_bits_engine<mt19937,64,
        uint_fast64_t> generator(seed);

    std::cout << generator() << " è un numero
casuale tra ";

    std::cout << generator.min() << " e " <<
    generator.max();

    return 0;
}
```

3. **shuffle_order_engine** : È un adattatore che adatta un motore di generatore di numeri pseudo-casuali così che i numeri siano ottenuti in una sequenza differente.

L'oggetto mantiene un buffer di k numeri generati internamente e quando richiesti restituisce alcuni selezionati casualmente nel buffer, rimpiazzandolo con un valore ottenuto dalla base del motore.

```
#include <iostream>
#include <chrono>
#include <random>

int main ()
{
```

```
unsigned seed = std::chrono::system_clock::  
now().time_since_epoch().count();  
  
// ranlux24 è un standard instantiation  
// del discard_block_engine:  
ranlux24 generator (seed);  
  
std::cout << generator() << " è un numero  
casuale tra ";  
  
std::cout << generator.min() << " e " <<  
generator.max();  
  
return 0;  
}
```

Map | Set | Pair | Hash Table

Definizione: Ora tratteremo delle **map**, **set**, **pair**, **hash table** che sono contenitori molto utili per poter eseguire delle operazioni sui loro elementi.

Map

Definizione: Le **mappe** sono dei contenitori associativi, ovvero che ogni elemento ha una chiave e un valore mappato. Non ci possono essere due valori con la stessa chiave.

Per poter utilizzare le **mappe** includiamo l'header `<map>`.

Quindi ricapitolando in una mappa ci sono una sequenza di (chiave, valore)

`{}{chiave:valore}, {chiave:valore}, {chiave:valore}, {chiave:valore.}{}{}`

```
#include <iostream>
#include <map>

int main()
{
    std::map<char , int> map;

    map.insert( 'a' , 0);
    map.insert( 'b' , 1);
    map.insert( 'c' , 2);

    std::cout << "map size: " << map.size() <<
        std::endl; // Output: map size: 3

    map.clear();

    std::cout << "map size dopo clear: " << map.
        size() << std::endl; // Output: map size dopo
        clear: 0

    std::map<std::string , int> myMap = {
        {"one" , 1}, {"two" , 2}, {"three" , 3}
    };
```

```
for( auto& i : myMap)
{
    std::cout << i.first << ": " << i.second
        << std::endl;
    // Output: one: 1
    // Output: two: 2
    // Output: three: 3
}

return 0;
}
```

unordered_map

Definizione: Una **unordered_map** è una mappa non ordinata, quindi i suoi elementi sono memorizzati in nessun tipo di ordine particolare, ma sono piazzati a caso.

Una **unordered_map** è implementata usando una *Hash Table*, quindi le performance della struttura dati dipendono dalla funzione hash.

Per poter utilizzare l'**unordered_map** è necessario includere il file di intestazione: `<unordered_map>`.

```
#include <iostream>
#include <string>
#include <conio.h>
#include <unordered_map>

int main()
{
    // create umap and initialize with initializer_list of std::pair
    std::unordered_map<std::string,int> map(
    {
        { "one",1 },
        { "seven",7 },
        { "five",5 },
        { "nine",9 },
        { "two",2 },
        { "eight",8 },
        { "three",3 },
        { "six",6 },
        { "four",4 },
        { "zero",0 }
    });

    // get user input
    std::string input;
    std::cin >> input;
    // lookup
    auto i = map.find( input );
}
```

Figura 2.18: unordered_map

unordered_map function pointer

The screenshot shows the Microsoft Visual Studio IDE interface. The top menu bar includes File, Edit, View, Project, Build, Debug, Team, Tools, Test, Analyze, Window, and Help. The toolbar below has icons for New, Open, Save, Print, and various debugging tools. The title bar indicates the current project is 'funky' and the build configuration is 'Debug - x86'. A status bar at the bottom shows 'Local Windows Debugger - Auto'. The main window contains a code editor with 'Source.cpp' open, displaying the following C++ code:

```
7
8     EVoid SixtyNine()
9     {
10        std::cout << "the sex number";
11    }
12
13     EVoid FourTwenty()
14    {
15        std::cout << "the weed number";
16    }
17
18
19     Eint main()
20    {
21        std::unordered_map<std::string,void(*)()> sw;
22
23        sw["sixty-nine"] = []()
24        {
25            std::cout << "the sex number";
26        };
27        sw["fourTwenty"] = FourTwenty;
28
29        sw["sixty-nine"]();
30
31        while( !_kbhit() );
32        return 0;
33    }
```

To the right of the code editor is the Solution Explorer, which lists the project 'funky' with its files: References, External Dependencies, Header Files, Resource Files, Source Files, and the specific file 'Source.cpp'.

Figura 2.20: unordered_map function pointer

The screenshot shows the Microsoft Visual Studio IDE interface. The main window displays the `Source.cpp` file under the project `funcy`. The code uses `std::unordered_map` to store function pointers:

```

4 #include <conio.h>
5 #include <memory>
6
7
8 void SixtyNine()
9 {
10     std::cout << "the sex number";
11 }
12
13 void FourTwenty()
14 {
15     std::cout << "the weed number";
16 }
17
18
19 int main()
20 {
21     std::unordered_map<std::string, void(*)()> sw;
22
23     sw["sixty-nine"] = SixtyNine;
24     sw["four-twenty"] = FourTwenty;
25
26     sw["sixty-nine"]();
27
28     while( !_kbHit() );
29     return 0;
30 }

```

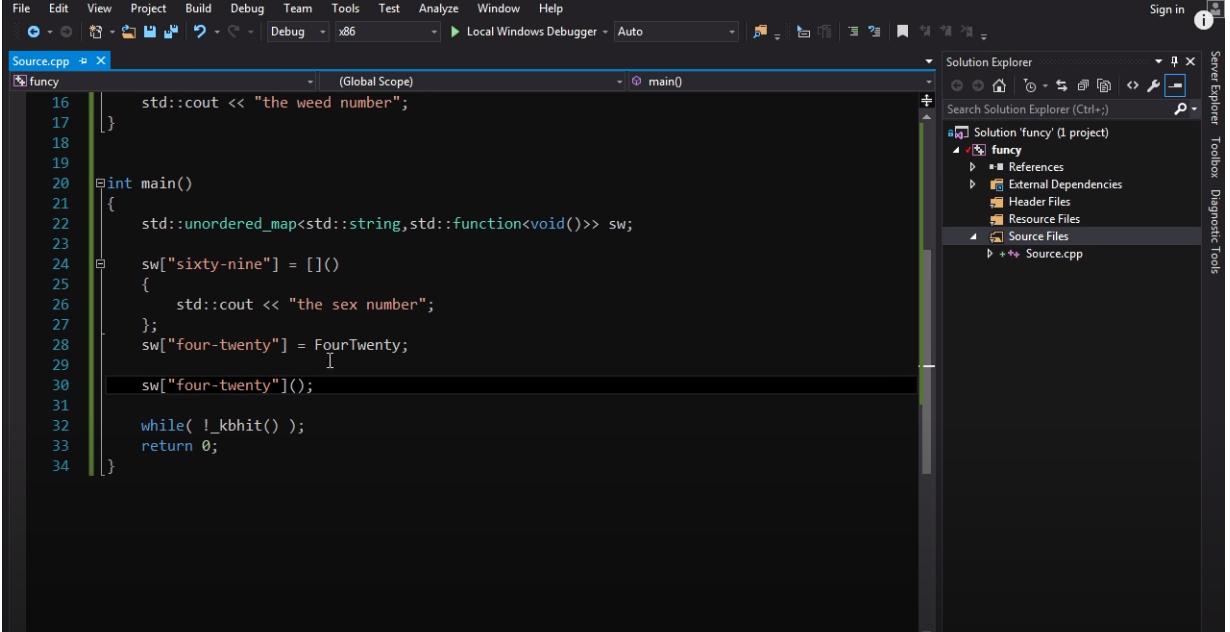
The Solution Explorer on the right shows the project structure with a single source file `Source.cpp`.

Figura 2.19: `unordered_map` function pointer

The screenshot shows the Microsoft Visual Studio IDE interface with handwritten annotations. Red arrows point from the `#include <functional>` directive and the `std::function<void()>` declaration in the `main()` function to the word **WRAPPER AROUND CALLABLE** written in red at the bottom right.

The annotated code is identical to Figura 2.19, but with the addition of the handwritten text and arrows.

Figura 2.21: `unordered_map` function pointer

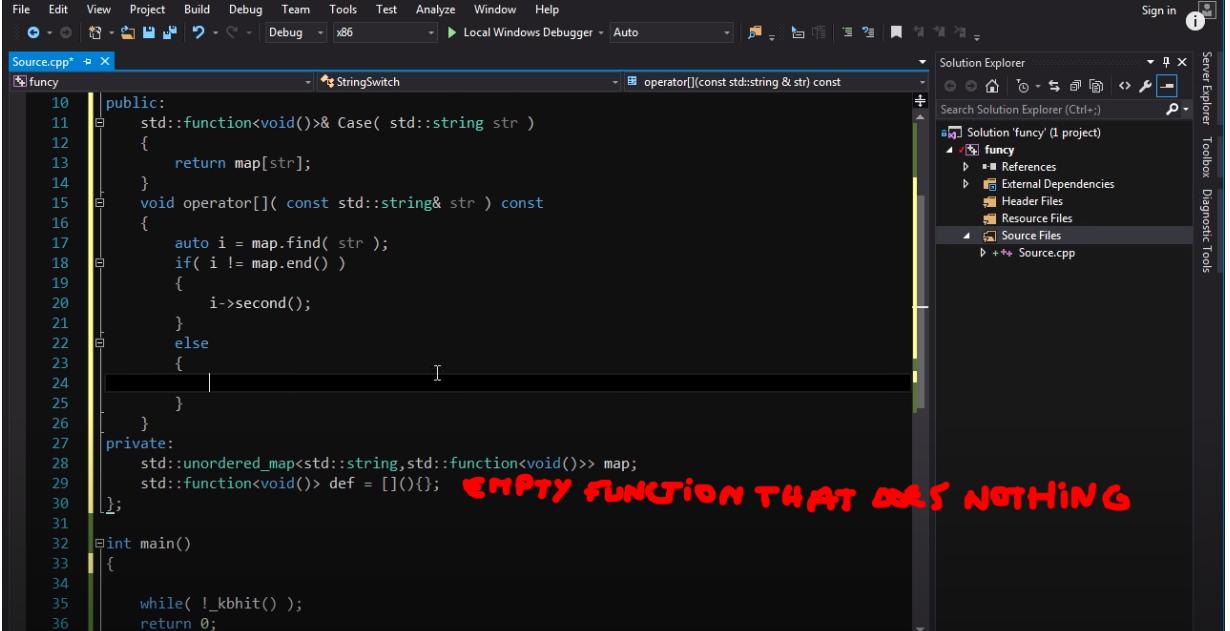


```

File Edit View Project Build Debug Team Tools Test Analyze Window Help
File Explorer Task List Solution Explorer Properties Server Explorer Toolbox Diagnostic Tools
Source.cpp (Global Scope) main()
16     std::cout << "the weed number";
17 }
18
19
20 int main()
21 {
22     std::unordered_map<std::string, std::function<void()>> sw;
23
24     sw["sixty-nine"] = []()
25     {
26         std::cout << "the sex number";
27     };
28     sw["four-twenty"] = FourTwenty;
29
30     sw["four-twenty"]();
31
32     while( !_kbhit() );
33     return 0;
34 }

```

Figura 2.22: unordered_map function pointer



```

File Edit View Project Build Debug Team Tools Test Analyze Window Help
File Explorer Task List Solution Explorer Properties Server Explorer Toolbox Diagnostic Tools
Source.cpp (Global Scope) operator[](const std::string & str) const
10 public:
11     std::function<void()>& Case( std::string str )
12     {
13         return map[str];
14     }
15     void operator[]( const std::string& str ) const
16     {
17         auto i = map.find( str );
18         if( i != map.end() )
19         {
20             i->second();
21         }
22         else
23         {
24             I
25         }
26     }
27 private:
28     std::unordered_map<std::string, std::function<void()>> map;
29     std::function<void()> def = [](){}; EMPTY FUNCTION THAT DOES NOTHING
30 };
31
32 int main()
33 {
34
35     while( !_kbhit() );
36     return 0;
37 }

```

Figura 2.23: unordered_map function pointer

multimap

Definizione: Una **multimap** è simile ad una **mappa**, ma con l'aggiunta che molteplici elementi possono avere le stesse chiavi. Inoltre, in questo caso, le coppie chiave-valore non devono essere per forza uniche. Inoltre, nelle **multimap** le chiavi sono sempre memorizzate in ordine.

```
#include <iostream>
#include <map>

int main()
{
    std::multimap<char, int> m1 = {
        {'a', 1},
        {'a', 2},
        {'b', 3},
        {'c', 4},
        {'c', 5},
    };

    std::multimap<char, int> m2(m1.begin(), m1.end());
}

std::cout << "La Multimap contiene i seguenti
elementi:" << std::endl;
for (auto it = m2.begin(); it != m2.end(); ++it)
    std::cout << it->first << " = " << it->second
    << std::endl;

//Output: La multimap contiene i seguenti elementi:
//Output: a = 1
//Output: a = 2
//Output: b = 3
//Output: c = 4
//Output: c = 5

return 0;
}
```

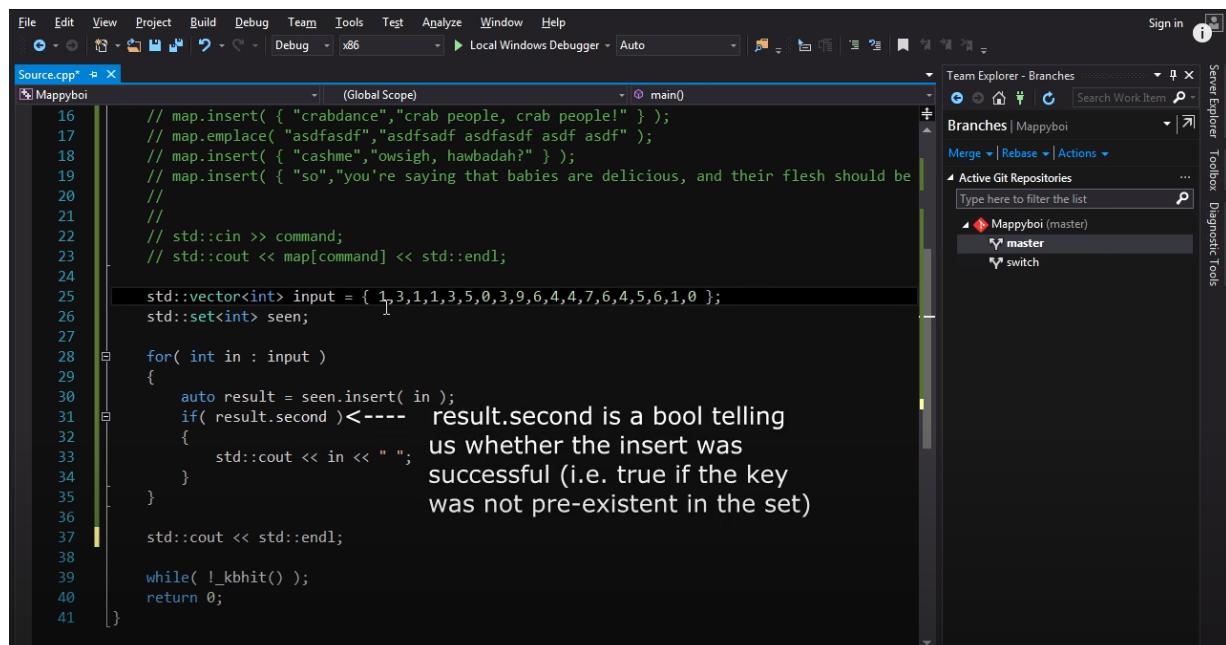
Set

Definizione: I **set** (insiemi in italiano) è un contenitore associativo nel quale gli elementi devono essere unici. Gli elementi dell'insieme non possono essere modificati una volta nel container, quindi sono sempre **const**. Per usare gli insieme è necessario includere l'header `<set>`.

Proprietà:

1. Il **set** memorizza gli elementi in ordine.
2. Tutti gli elementi in un **set** hanno valori unici.
3. I valori del **set** non possono essere modificati, quindi sono **immutable** (immutabili).
4. I **set** utilizzano la **Binary search tree**.
5. I valori in un **set** sono **non indicizzati**.

Per memorizzare gli elementi in maniera non ordinata c'è **unordered_set()**.



The screenshot shows the Microsoft Visual Studio IDE interface. On the left, the code editor displays a file named "Source.cpp" with the following content:

```

File Edit View Project Build Debug Team Tools Test Analyze Window Help
Source.cpp*  (Global Scope) main()
16 // map.insert( { "crabdance", "crab people, crab people!" } );
17 // map.emplace( "asdfsadf", "asdfsadf asdfsadf asdf asdf" );
18 // map.insert( { "cashme", "owsigh, hawbadah?" } );
19 // map.insert( { "so", "you're saying that babies are delicious, and their flesh should be
20 // "
21 // "
22 // std::cin >> command;
23 // std::cout << map[command] << std::endl;
24
25 std::vector<int> input = { 1,3,1,1,3,5,0,3,9,6,4,4,7,6,4,5,6,1,0 };
26 std::set<int> seen;
27
28 for( int in : input )
29 {
30     auto result = seen.insert( in );
31     if( result.second )---- result.second is a bool telling
32     {                               us whether the insert was
33         std::cout << in << " ";           successful (i.e. true if the key
34     }
35 }
36
37 std::cout << std::endl;
38
39 while( !_kbhit() );
40 return 0;
41 }

```

The right side of the interface shows the "Team Explorer" window, which includes a "Branches" section showing a local repository named "Mappyboi" with a single branch "master".

Figura 2.24: set

multiset

Definizione: I **multiset** sono dei container associativi simili ai **set**, ma con un'eccezione che molteplici elementi possono avere gli stessi valori.

```
#include <iostream>
#include <set>

int main()
{
    std::multiset<int> multiSet;

    multiSet.insert(9);
    multiSet.insert(9);
    multiSet.insert(9);

    std::cout << multiSet.count(9) << std::endl;
    //Output: 3

    multiSet.erase(multiSet.find(9)); // Rimuoverà
                                    // solo uno dei tre 9.

    std::cout << multiSet.count(9) << std::endl;
    //Output: 2

    multiSet.erase(9); // Rimuoverà tutte le occorrenze
                      // del valore 9.

    std::cout << multiSet.count(9) << std::endl;
    //Output: 0

    return 0;
}
```

Pair

Definizione: Il **pair** (paio, coppia) serve per combinare due valori anche di tipi diversi. È praticamente usato per memorizzare delle **tuple** (sequenza di n elementi).

I **pair** sono contenuti nell'header **<utility>**.

Alcune proprietà del **pair**:

- Il primo elemento viene referenziato come *first* ed il secondo come *second*.
- I **pair** possono essere assegnati, copiati, paragonati. Gli elementi in una mappa o hash_map sono di tipo **pair** per default, dove *first* è la chiave associata con il suo valore da *second*.
- Per accedere agli elementi, usiamo il nome della variabile seguita dall'operatore . (punto) seguito dalla keyword *first* o *second*.

```
#include <iostream>
#include <utility> // Per i pair

int main()
{
    std::pair<char, int> pair;
    pair.first = 'L';
    pair.second = 1;

    std::cout << pair.first << " " << pair.second
        << std::endl; //Output: L 1

    std::pair<std::string, double> pair2("Ciao a
        tutti", 2.22);

    std::cout << pair2.first << " " << pair2.
        second << std::endl; //Output: Ciao a tutti 2.22

    std::pair<int, int> coords = {3, 2}; //
        Un'altra sintassi valida per dichiarare un pair.

    std::cout << coords.first << " " << coords.
        second << std::endl; //Output: 3 2

    return 0;
}
```

Alcune funzioni membro di **pair**:

- **make_pair()**: permette di creare un **pair** anche senza passare i parametri esplicitamente.

- **swap()**: scambia i contenuti di un pair con un altro.
- **tie()**: crea una tupla di referenze lvalues come argomento, per disimballare, spacchettare il pair in variabili separate. Ci sono anche alcune varianti al tie, con o senza *ignore* che permette di ignorare un dato elemento dall'essere spacchettato.

```
#include <iostream>
#include <utility>

int main()
{
    // Esempio make_pair
    std::pair<int, int> pair = std::make_pair(8,
                                                2);
    std::cout << pair.first << " " << pair.second
    << std::endl; //Output: 8 2

    // Esempio swap
    std::pair<char, int> pair1 = std::make_pair(
        'A', 0);
    std::pair<char, int> pair2 = std::make_pair(
        'B', 1);

    std::cout << "Prima dello swap: " << pair1.
        first << " " << pair1.second << std::endl;
    //Output: Prima dello swap: A 0

    pair1.swap(pair2);

    std::cout << "Dopo lo swap: " << pair1.first
    << " " << pair1.second << std::endl; //
    Output: Dopo lo swap: B 1

    // Esempio tie
    std::pair<int, int> p1 = { 1, 2 };
    int a, b;
    std::tie(a, b) = p1;
    std::cout << a << " " << b << "\n"; // Output:
    1 2
```

```

std::pair<int, int> p2 = { 3, 4 };
std::tie(a, ignore) = p2;

std::cout << a << " " << b << "\n"; // Output:
      3 2 (b è rimasto al valore precedente perché abbiamo
            messo ignore)
return 0;
}

```

Possiamo usare gli operatori anche con i pair:

1. `=`: implementato in questo modo: `std::pair& operator=(const std::pair& pr)`.
2. `==` per comparare due pair.
3. `!=` per controllare se non sono uguali.
4. `>=, <=` per controllare se è maggiore uguale o minore uguale.

Tuple

Definizione: Una **tuple** (tupla) è un oggetto che può tenere un numero di elementi. Gli elementi possono essere di vari tipi.

Per poter usufruire delle tuple occorre importare `<tuple>`.

Varie operazioni che si possono fare sulle tuple:

- **get()** : è usato per accedere ai valori della tupla e per poterli modificare. Accetta sia l'indice sia il nome come argomento per accedere ad un particolare elemento.
- **make_tuple()** : usato per assegnare valori alla tupla.
- **tuple_size** : restituisce il numero di elementi presenti nella tupla.
- **swap()** : scambia gli elementi di due tuple.
- **tie()** : spacchetta le tuple in variabili separate. Ci sono due varianti con e senza **ignore** per ignorare un particolare elemento della tupla e per non farlo spacchettare.
- **tuple_cat()** : concatena due tuple e ne restituisce una nuova.

```

#include <iostream>
#include <tuple>

int main()
{
    std::tuple<char, int, float> tupla;
    tupla = std::make_tuple('S', 33, 52.6);

    std::cout << "Prima della modifica: " << std
        ::get<0>(tupla) << " " << std::get<1>(
        tupla) << std::get<2>(tupla) << std::endl;
        //Output: Prima della modifica: S 33 52.6

    // Usare get() per modificare i valori:
    std::get<0>(tupla) = 'T';
    std::get<2>(tupla) = 21.9;

    std::cout << "Dopo la modifica: " << std::get<
        <0>(tupla) << " " << std::get<1>(tupla) <<
        std::get<2>(tupla) << std::endl; //Output:
        Dopo la modifica: T 33 21.9

    // tuple_size
    std::cout << "Il size della tupla è: " << std
        ::tuple_size<decltype(tupla)>::value <<
        std::endl; //Output: Il size della tupla è: 3

    // swap()

    std::tuple<char, int, float> tupla2('R', 54,
        61.7);
    std::cout << "Elementi prima dello scambio: "
        << std::get<0>(tupla) << " " << std::get<
        <1>(tupla) << " " << std::get<2>(tupla) <<
        std::endl; //Output: Elementi prima dello scambio:
        T 33 21.9

    tupla.swap(tupla2);

    std::cout << "Elementi dopo lo scambio: " <<
        std::get<0>(tupla) << " " << std::get<1>(

```

```

tupla) << " " << std :: get<2>(tupla) << std
:: endl; //Output: Elementi dopo lo scambio: R 54
61.7

std :: get<0>(tupla) = 'T';
std :: get<1>(tupla) = 33;
std :: get<2>(tupla) = 21.9;

// tie()
char c;
int x;
float f;

std :: tie(c, x, f) = tupla;

std :: cout << "I valori spacchettati della
tupla sono: " << c << " " << x << " " << f
<< std :: endl; //Output: I valori spacchettati della
tupla sono: R 54 61.7

// tuple_cat()
auto tupla3 = std :: tuple_cat(tupla, tupla2);

std :: cout << "Gli elementi della nuova tupla
sono: " << std :: get<0>(tupla3) << " " <<
std :: get<1>(tupla3) << " " << std :: get<2>(
tupla3) << " " << std :: get<3>(tupla3) << "
" << std :: get<4>(tupla3) << " " << std ::
get<5>(tupla3) << " " << std :: endl; //
Output: Gli elementi della nuova tupla sono: T 33 21.9 R
54 61.7
return 0;
}

```

Pair vs Tuple

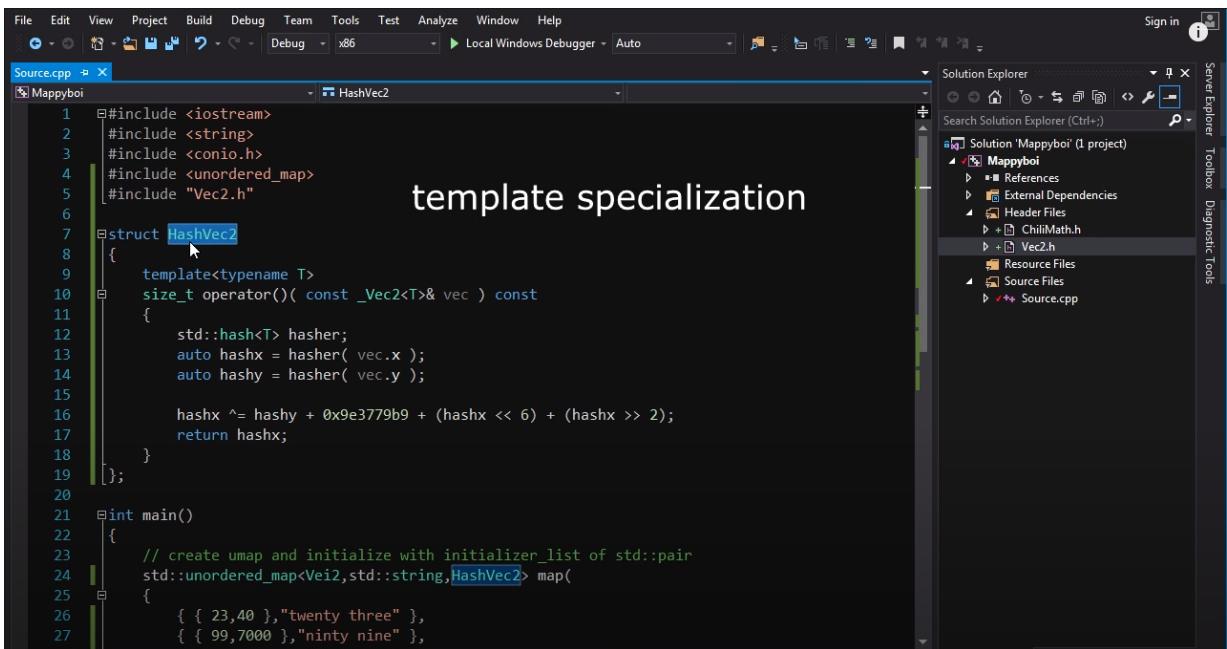
Quali sono le differenze tra **pair** e **tuple**?

Pair	Tuple
Può contenere soltanto una coppia di valori.	Può contenere molteplici valori.
Un Pair è una specifico caso di tuple a due elementi.	È una classe template con size fisso di valori eterogenei.
Non è difficile ottenere i suoi elementi.	È un po' più complicato ottenere i suoi elementi.
Dovrebbe essere un po' più veloce di tuple.	È un po' più lenta se non ottimizzata.

Hash Table

Definizione: L'**hashing** è una tecnica o processo di mappatura di chiavi e valori in una **hash table** attraverso una **hash function**.

Queste **hash function** potrebbero portare ad una collisione, ovvero quando due o più chiavi sono mappate allo stesso valore. Il **Chain Hashing** evita le collisioni: Ogni cella dell'**hash table** punta ad una *lista linkata* di records che hanno lo stesso valore di **hash function**.



The screenshot shows the Microsoft Visual Studio IDE interface. The left pane displays the 'Source.cpp' file containing C++ code for a hash table template specialization. The code defines a struct `HashVec2` with a template operator() that calculates a hash value for a 2D vector. The right pane shows the 'Solution Explorer' window, which lists the project 'Mappyboi' containing files like 'ChilliMath.h', 'Vec2.h', and 'Source.cpp'. The title bar indicates the project is 'Mappyboi' and the file is 'Source.cpp'.

```

File Edit View Project Build Debug Team Tools Test Analyze Window Help
Source.cpp Mappyboi HashVec2
1 #include <iostream>
2 #include <string>
3 #include <conio.h>
4 #include <unordered_map>
5 #include "Vec2.h"
6
7 struct HashVec2
8 {
9     template<typename T>
10     size_t operator()( const Vec2<T>& vec ) const
11     {
12         std::hash<T> hasher;
13         auto hashx = hasher( vec.x );
14         auto hashy = hasher( vec.y );
15
16         hashx ^= hashy + 0x9e3779b9 + (hashx << 6) + (hashx >> 2);
17         return hashx;
18     }
19 };
20
21 int main()
22 {
23     // create umap and initialize with initializer_list of std::pair
24     std::unordered_map<Vec2<string>, HashVec2> map(
25     {
26         { { 23,40 }, "twenty three" },
27         { { 99,7000 }, "ninety nine" },
28     }
29     );
30 }

```

template specialization

Figura 2.25: Hash Table Template Specialization

Type Traits

Definizione: L'header `<type_traits>` contiene un insieme di classi template per trasformare e controllare le proprietà dei tipi a compile-time.
 Sono di solito usati per controllare gli errori dell'utente, sviluppo della programmazione generale e permettere ottimizzazioni.
 La maggior parte dei **type traits** sono usati per controllare se un tipo soddisfa il criterio.

decltype

Definizione: **decltype** è un modo per specificare il tipo: gli dai un'espressione e **decltype** ti restituisce il tipo (la tipologia della variabile) dell'espressione che gli hai passato.

Per esempio: se passi a **decltype(a)**, dove a è il nome di una variabile (*id-expression*) di tipo **int** allora **decltype** restituirà la tipologia di quella variabile che in questo caso è **int**. Se, invece, a è un **lvalue** di tipo **T** allora restituirà **T &** e se a è un **rvalue** allora restituirà il tipo **T**.

```
int foo();
int n = 9;

// Usiamo decltype per ottenere la tipologia e la usiamo per
// assegnarla alle variabili.
decltype(n) x = 18; // la variabile x è di tipo int.
                     (id-expression)

decltype((n)) y = x; // y è un int& (n è un lvalue)

decltype(foo()) z = foo(); // z è un int (rvalue)

decltype(foo()) && w = foo(); // w è un int&&

decltype((n)) && k = n; // k è int& (& && collassa in &)
```

La forma speciale **decltype(auto)** deduce la tipologia della variabile dalla sua inizializzazione.

```
const int a = 99;
auto b = a; // b ha tipo int
decltype(auto) c = a; // c ha tipologia const int, ciò che è
                      stato ritornato da decltype.
```

Alcuni esempi su type_traits:

```
#include <iostream>
#include <type_traits>

// is_function
void f() {};
int x(int a){return a;};
int (*y)(int)=x;

// is_class
class MyClass {};

int main()
{
    // is_function
    std::cout << std::boolalpha << "is_function<x
        >: " << std::is_function<decltype(x)>::
        value << " is_function<y>: " << std::
        is_function<decltype(y)>::value << std::
        endl; //Output: is_function<x>: true is_function<y>:
        false

    // is_class
    std::cout << std::boolalpha << "is_class<
        MyClass>: " << std::is_class<MyClass>::
        value << std::endl; //Output:
        is_class<MyClass>: true
    return 0;
}

#include <iostream>
#include <type_traits>

template <typename T>
void miServeUnPuntatore(T t) {
```

```

    std::static_assert(std::is_pointer<T>::value,
                      "T deve essere di tipo puntatore");
}

//Overload per quando T non è un puntatore.
template<typename T>
typename std::enable_if<!std::is_pointer<T>::value>::type
faccioQualcosaColPuntatore(T t) {
    //Corpo della funzione
}

//Overload per quando T è un puntatore
template<typename T>
typename std::enable_if<std::is_pointer<T>::value>::type
faccioQualcosaColPuntatore (T t) {
    //Corpo della funzione
}

```

Oltre a queste sono presenti tante altre funzioni.

RTTI | Run-Time Type Information

Definizione: Il Run-Time Type Information (**RTTI**) è un meccanismo che espone le informazioni del tipo di un oggetto a runtime ed è disponibile solo per le classi che hanno almeno una **funzione virtuale**. Permette di determinare la tipologia di un oggetto durante l'esecuzione del programma.

Runtime Casts

Il cast a runtime che controlla che il cast sia valido è l'approccio più semplice per verificare il tipo dell'oggetto a runtime usando un puntatore o una referenza. È utile quando abbiamo bisogno di fare un cast da un puntatore ad una classe base a una classe derivata.

Quando abbiamo a che fare con una gerarchia di classi, il casting degli oggetti è richiesto, di solito.

Ci sono due tipi di casting:

- **Upcasting** : Quando un puntatore o referenza di una classe derivata è trattata come un puntatore a una classe base.
- **Downcasting** : Quando un puntatore o referenza a una classe base è convertito a puntatore di una classe derivata.

dynamic_cast : in una gerarchia di ereditarietà è usato per il *downcasting* di un puntatore di una classe base a una classe figlia.

Se riesce ritorna un puntatore al tipo convertito, altrimenti, fallisce se proviamo a fare un cast ad un tipo invalido come un puntatore che non è del tipo desiderato della classe derivata. Per esempio: **dynamic_cast** non riesce a fare un cast ad un' altra classe se questa non ha almeno una funzione virtuale.

```
#include <iostream>
class B {
};
class D : public B {
};

int main()
{
    B* b = new D;
    D* d = std::dynamic_cast<D*>(b); // Da errore
                                         perché B non ha nemmeno una funzione virtuale.
    return 0;
}
```

Typeinfo

Definizione: Il file di intestazione `<typeinfo>` contiene implementazioni riguardo l'ottenere informazioni di un tipo, incluso il nome del tipo e modi per compararlo con altri tipi.

typeid

È usato quando servono le informazioni del tipo dinamico o a runtime di un oggetto.

È un'espressione lvalue.

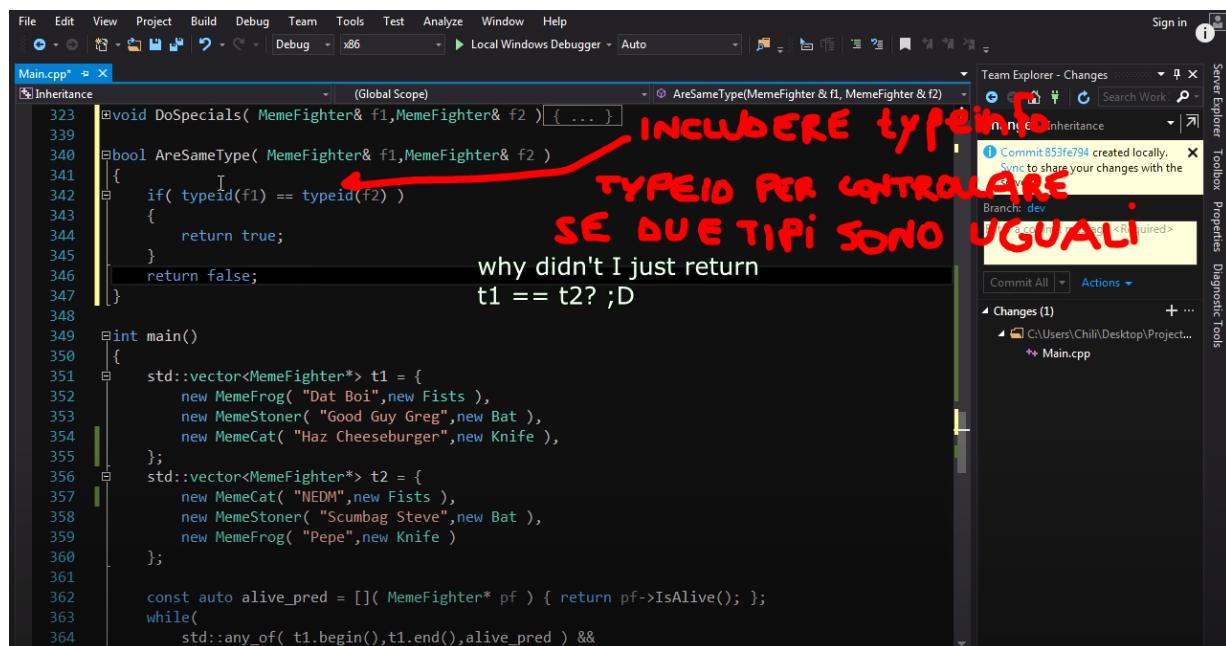


Figura 2.26: typeid

```

Main.cpp* ⓘ
Inheritance MemeStoner SpecialMove(MemeFighter & other)
255 {}
256 void SpecialMove( MemeFighter& other ) override
257 {
258     if( IsAlive() )
259     {
260         if( Roll() > 3 )
261         {
262             if( typeid(MemeFrog) == typeid(other) )
263             {
264                 std::cout << GetName() + " says: 'Oh sweet dude, it's a cool little froggie br";
265             }
266             else if( typeid(MemeStoner) == typeid(other) )
267             {
268                 std::cout << GetName() + " says: 'Duuuuude.\n";
269             }
270             else if( typeid(MemeCat) == typeid(other) )
271             {
272                 std::cout << GetName() + " says: 'Hey kitty bro, can I pet you?' \n";
273             }
274             std::cout << GetName() << " smokes the dank sticky icky, becoming " << "Super " <<
275             name = "Super " + name;
276             attr.speed += 3;
277             attr.power = (attr.power * 69) / 42;
278             attr.hp += 10;
279         }
280     }
281 }

```

Figura 2.27: typeid

In alcuni casi, **typeid** potrebbe essere più veloce di **dynamic cast**.

typeinfo

La funzione **typeinfo()** è un costrutto del compilatore GCC, non esiste (al momento) nel linguaggio e fa la stessa cosa di **decltype**, ovvero restituisce la tipologia di una variabile.

Ellissi

Definizione: Le **Ellissi**, ovvero i ... (tre puntini) permettono ad una funzione di accettare un numero variabile di argomenti. È anche conosciuta come *variable argument list*.

Di solito, le funzioni possono prendere solo un numero fisso di parametri, ma con le **ellissi** (*ellipsis* in inglese) si possono passare tanti argomenti quanti se ne vuole.

Le **ellissi** sono definite nel file di intestazione **cstdarg**.

```
#include <iostream>
#include <cstdarg>

double average(int count, ...)
{
    // va_list è usata per iterare sull'ellissi.
    va_list list;

    // Inizializza la posizione di va_list.
    va_start(list, count);

    double average = 0.0;

    // Itera su ogni argomento passato.
    for(int i = 0; i < count; i++)
    {
        average += static_cast<double>(va_arg(
            list, int)) / count;
    }

    // Finisce l'utilizzo di va_list.
    va_end(list);

    return average;
}

int main()
```

```
{  
    double avg = average(6, 3, 6, 9, 10, 7, 1);  
  
    std :: cout << "La media è: " << avg << std ::  
        endl; //Output: La media è: 6.0 ((3 + 6 + 9 + 10 + 7 +  
        1) / 6 = 36 / 6 = 6.0)  
    return 0;  
}
```

Come funziona?

- **va_list**: usato per accedere ai valori dell'ellissi.
- **va_start**: punta alla **va_list** all'inizio dell'ellissi.
- **va_arg**: restituisce il valore a cui **va_list** si sta riferendo e muove **va_list** al prossimo parametro.
- **va_end**: prende soltanto un argomento **va_list** stessa. È usato per pulire la **va_list**.

Non è molto usata in C++, perché al posto di usare l'ellissi possiamo passare come argomento un contenitore, come un **std::vector**, eccetera..

Inline Functions

Definizione: Le **inline functions** (*funzioni in linea*) sono un concetto molto potente. Se una funzione è **inline** il compilatore piazzerà una copia del codice di quella funzione ad ogni punto dove essa è chiamata.

Per le funzioni grandi, lunghe e/o che eseguono complesse tasks, l'*overhead* (sovraffante, sopra la testa, risorse in sovrappiù rispetto a quelle strettamente necessarie) è di solito insignificante comparato all'ammontare di tempo che la funzione ci mette per essere eseguita. Mentre, per quelle più piccole, le funzioni usate comunemente, il tempo necessario per fare la chiamata alla funzione è spesso molto di più rispetto a quello necessario per eseguirla.

Questo occorre per le piccole funzioni, perché il tempo di esecuzione è minore del tempo di cambio. (*switching time*).

Quindi, la keyword **inline** nelle funzioni serve per ridurre l'*overhead* nella chiamata alla funzione. È una funzione che viene espansa in linea quando viene chiamata. Quando la funzione **inline** viene chiamata, l'intero codice viene sostituito o inserito al momento della chiamata.

Il compilatore può ignorare il qualificato **inline** nel caso che la funzione definita **inline** sia più di una riga di codice.

È comune usarlo nelle funzioni delle classi.

```
#include <iostream>

inline int max(int x, int y)
{
    return x > y ? x : y;
}

int main()
{
    std::cout << "max(3,9): " << max(3,9) << std
        ::endl; //Output: 9
    std::cout << "max(21, 0): " << max(21, 0) <<
        std::endl; //Output: 21
    std::cout << "max(4, 12): " << max(4, 12) <<
        std::endl; //Output: 12
```

}

Inoltre il compilatore non la eseguirà **inline** se:

- Se una funzione contiene un loop.
- Se una funzione contiene variabili statiche.
- Se una funzione è ricorsiva.
- Se il tipo di ritorno non è void e il return non esiste nel corpo della funzione.
- Se la funzione contiene switch o goto.

Le funzioni **inline** hanno i seguenti vantaggi:

- Non occorre l'*overhead* della funzione.
- Risparmia l'*overhead* della push/pop di variabili nello stack quando è chiamata.
- Risparmia l'*overhead* di una chiamata **return** da una funzione.
- Il compilatore potrebbe eseguire delle ottimizzazioni sul corpo della funzione. Questo non è possibile per le chiamate alle funzioni normali. Inoltre potrebbe ottimizzare anche il flusso delle chiamate.
- Possono essere utili per gli *embedded systems* perché possono produrre meno codice.

Gli svantaggi delle funzioni **inline**:

- Le variabili aggiunte consumano più registri, il che potrebbe causare *overhead* sul *register variable resource utilization*.
- Se utilizzi troppe funzioni **inline** allora lo spazio del file binario eseguibile sarà molto grosso per via della duplicazione dello stesso codice.
- Troppe **inline** può anche ridurre le *cache hit* (quando il contenuto che ti serve si trova nella cache e quindi viene caricato direttamente da lì al posto di richiederla dalla memoria primaria), riducendo la velocità di fetch dalla cache alla memoria primaria.
- Potrebbero aumentare l'*overhead* del compilatore, perché se uno modifica una funzione **inline** allora il compilatore dovrebbe rimpiazzare tutto il codice un'altra volta, altrimenti continuerebbe ad essere eseguita con le vecchie funzionalità.

- Potrebbero non essere utili per gli *embedded systems*, perché in essi lo spazio del codice è più importante della velocità.
- Potrebbero causare *thrashing* (quando l'hard disk è in sovraccarico per via dello spostamento di informazioni tra memoria di sistema e memoria virtuale eccessivamente, portando a continui *page faults*), perché l'**inline** potrebbe incrementare lo spazio del file eseguibile.

```
#include <iostream>

// Bad practice
class A {
public:
    inline int square(int s) // è ridondante
    {
        // Questa funzione è automaticamente inline.
    }
};

// Good practice
class A {
public:
    int square(int s); // Dichiarazione prototipo
                       // della funzione
};

inline A::square(int s) // Definizione della funzione.
{
    // Corpo della funzione.
}
```

Altre curiosità sulle funzioni **inline**:

È raccomandato di usare sempre le funzioni **inline** al posto delle macro. Le macros (in C++) non sono quasi mai necessarie e portano ad errori. Ci sono alcuni problemi con le macro: non possono accedere ai membri privati di una classe, sembrano delle funzioni, ma non lo sono, ed altro..

Tendenzialmente le funzioni che fanno I/O (input/output) non dovrebbero essere definite come **inline** perché spenderebbero un considerevole ammontare di tempo. Il tempo del codice di I/O supera l'*overhead* della chiamata alla funzione.

Un buon utilizzo delle funzioni **inline** possono essere di ottimo valore, ma un uso inappropriate non porterà a risultati migliori. Non porre tutte le funzioni **inline**. È meglio avere il minor numero di funzioni **inline** possibili.

Copy Elision

Definizione: La **Copy Elision** o anche **Copy Omission** è una tecnica di ottimizzazione del compilatore che evita la copiatura non necessaria di oggetti.

```
#include <iostream>

class A {
public:
    A(const char* str = "\0")
    {
        std::cout << "Chiamato il costruttore
                    . " << std::endl;
    }

    A(const A &a)
    {
        std::cout << "Costruttore di copia
                    chiamato. " << std::endl;
    }
};

int main()
{
    A a = "copiami";
    // Output: Chiamato il costruttore.
    // In teoria quando l'oggetto a viene costruito, il costruttore a
    // un argomento è usato per convertire "copiami" all'oggetto
    // temporaneo e l'oggetto temporaneo è copiato nell'oggetto
    // a.
    // La maggior parte dei compilatori C++ evita questi overhead
    // di creare un oggetto temporaneo e di copiarlo.
```

```
// Ma se non vogliamo questo comportamento, possiamo
// compilare passando l'argomento "-fno-elide-constructors"
// g++ nomefile.cpp -fno-elide-constructors
return 0;
}
```

Conversion Constructor & Explicit keyword

Explicit Keyword

Definizione: La keyword **explicit** è usata per marcare il costruttore per non convertire implicitamente i tipi.

```
#include <iostream>

class Foo {
public:
    Foo(int n); // Alloca n bytes all'oggetto Foo.
    Foo(const char *p); // inizializza l'oggetto con char
                        *p.
};

class Foo2 {
public:
    explicit Foo(int n); // Alloca n bytes.
    Foo(const char* p); // inizializza con la stringa p.
};

int main()
{
    Foo mystring = 'x'; // In questo caso il carattere 'x'
                        verrà implicitamente convertito ad int e verrà chiamato il
                        costruttore Foo(int), ma questo non è ciò che intendevamo
                        fare. Per evitare questo mettiamo la keyword explicit nel
                        costruttore.
```

```

Foo2 mystring = 'x'; // In questo caso verrà chiamato
                     il costruttore con Foo(const char* p);
    return 0;
}

```

Conversion Constructor

Definizione: Se un costruttore può essere chiamato con un solo parametro, allora questo costruttore diventa un **conversion constructor**, perché permette la conversione automatica alla classe costruita.

```

#include <iostream>

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

class B {
public:
    int x, y;
    B(int x, int y)
    {
        this->x = x;
        this->y = y;
    }
};

int main()
{
    A obj(7);

    obj = 12; // Conversion constructor invocato.

    B obj2(3, 8);

```

```

obj2 = { 4, 0 }; // Conversion constructor a parametri
multipli invocato.

return 0;
}

Master n( 69 ); // Converting Constructor
Master doob = 420; // Noi passiamo un intero e lui crea dal Costruttore coll'intero un'istanza di Master
Func( 1337 ); // Qui la funzione prende come parametro un'istanza di Master, mentre noi gli passiamo un numero
// intero e lui crea un'istanza di Master chiamando il costruttore coll'intero.
// A volte però, magari, non si vuole che l'utente passi un intero alla funzione, ma che passi un'istanza e
// per poter far questo dobbiamo utilizzare la keyword "explicit" nel costruttore.

```

Figura 2.28: Conversion Constructor

Private Destructors

Definizione: Quando vogliamo prevenire la distruzione di un oggetto, possiamo porre il suo distruttore privato.

Questo non potrà mai essere allocato sullo stack, sempre sull'heap (memoria allocata dinamicamente). L'unico modo per poterli distruggere sarà in altri modi, attraverso un **friend** o un **membro**.

```

#include <iostream>

class MyClass {
private:
    ~MyClass()
    {
        // Distruttore privato.
    }
public:
    friend void destructMyClass(MyClass* );
};

// Dichiara all'esterno, ma che può ancora accedere all'interno
// della classe perché dichiarata friend.
void destructMyClass(MyClass* ptr)

```

```
{  
    delete ptr;  
}  
  
int main()  
{  
    MyClass object; // Questo darà errore fermando il  
                     programma.  
  
    MyClass *object; // Col puntatore non darà errore  
                     perché nessun oggetto è stato creato. (ovvio che bisogna  
                     rimuovere la riga sopra, altrimenti non va il programma)  
  
    MyClass object2 = new MyClass; // Anche questo  
                                   non darà errore quindi sarà il programmatore ad  
                                   occuparsi di rimuovere la memoria dell'oggetto allocata.  
  
    delete object2; // Però in questo caso se chiamiamo  
                    delete genererà un errore per via del distruttore privato.  
  
    destructMyClass(object); // In questo caso l'oggetto  
                            verrà effettivamente distrutto.  
    return 0;  
}
```

Command Line Arguments

Definizione: È possibile passare degli argomenti da linea di comando nei terminali dei sistemi operativi. Per fare questo definiamo il **main()** con due parametri: il primo corrisponde al numero di comandi da linea di comando e il secondo è la lista degli argomenti passati da linea di comando.

// Questi sono i due tipici modi di definire il main per passare argomenti da linea di comando:

```
int main(int argc, char *argv[])
// Oppure:
int main(int argc, char **argv)
```

- **argc (ARGument Count)** : è un intero e memorizza il numero di argomenti passati da linea di comando da parte dell'utente, incluso il nome del programma. Quindi se passiamo un valore al programma, il valore di `argc`, il numero di argomenti sarebbe 2: uno l'argomento passato e l'altro il nome del programma.
- **argv (ARGument Vector)** : è un array di puntatori di caratteri che lista tutti gli argomenti.
 - Se `argc` è maggiore di zero, l'array di elementi da `argv[0]` a `argv[argc - 1]` conterrà i puntatori alle stringhe.
 - `argv[0]` è il nome del programma. Dopo quello fino ad `argv[argc - 1]` ogni elemento è un comando passato da linea di comando al programma.

```
#include <iostream>

int main(int argc, char** argv)
{
    std::cout << "Hai inserito: " << argc << "
    argomenti: " << "\n";
    for(int i = 0; i < argc; i++)
```

```
{  
    std :: cout << argv[ i ] << "\n";  
}  
  
//Input: ./programma ciao a tutti  
//Output: Hai inserito 4 argomenti:  
//Output: ./programma  
//Output: ciao  
//Output: a  
//Output: tutti  
  
return 0;  
}
```

Proprietà degli argomenti da linea di comando:

- Sono passati alla funzione `main()`.
- Sono parametri/argomenti forniti al programma quando viene invocato.
- Sono usati per controllare il programma dall'esterno, piuttosto che metterli all'interno del codice.
- `argv[argc]` è un puntatore a `NULL`.
- `argv[0]` tiene il nome del programma.
- `argv[1]` punta al primo argomento e `argv[argc - 1]` all'ultimo.

Concetti Avanzati

3

Introduzione

In questo capitolo "finale" tratterò argomenti un po' più complessi o che almeno non mi verrebbe da mettere negli altri due capitoli precedenti.

Verranno trattati argomenti come gli smart pointers e quindi unique pointers, share pointers, weak pointers, le friend function, uniform real distribution e altri importanti concetti avanzati.

Friend Keyword

Friend Class

Definizione: La keyword **friend** è usata per accedere ai membri privati e protetti di una classe nella quale è dichiarata **friend**.

```
#include <iostream>

class A {
public:
    A() { a = 0 };
    friend class B; // Classe amica.

private:
    int a;
};

class B {
public:
    void showA(A& x)
    {
        // Visto che B è un'amica di A, può accedere ai
        // membri privati di A.
        std::cout << "A::a : " << x.a;
    }
};

int main()
{
    A a;
    B b;
    b.showA(); //Output: A::a : 0
    return 0;
}
```

Friend Function

Definizione: Come per le classi friend, una funzione friend ha accesso speciale ai membri privati e protetti.

Una **friend function** può essere:

- Un membro di un'altra classe.
- Una funzione globale.

Alcuni importanti punti riguardo alle **friend functions** e classes:

- Dovrebbero essere usate solo in maniera limitata. Troppe funzioni o classi friend diminuiscono l'encapsulazione.
- L'amicizia non è reciproca. Se la classe A è amica della classe B, allora B non è automaticamente amica di A.
- L'amicizia non è ereditata.

```
#include <iostream>

class A {
public:
    friend void printWidth( A a );
    void setWidth( double w );

private:
    double width;
};

// Definizione della funzione membro di A.
void A::setWidth( double w )
{
    width = w;
}

// printWidth non è una funzione membra di nessuna classe.
void printWidth( A a )
{
    // Visto che la funzione printWidth è amica di A, può
    accedere direttamente a qualsiasi membro di A.
```

```
    std::cout << "Width di A: " << a.width << std
                  ::endl;
}

int main()
{
    A a;

    a.setWidth(11.1);

    // Uso la funzione amica per stampare la width di a.
    printWidth( a ) ; //Output: Width di A: 11.1
    return 0;
}
```

Smart Pointers

Definizione: Gli **smart pointers** (puntatori intelligenti) sono dei puntatori che, in più rispetto ai normali puntatori, sono in grado deallocare la memoria automaticamente, senza che il programmatore debba occuparsene ed evitando *memory leak*.

Differenze con i puntatori normali

I puntatori servono per poter accedere a delle risorse che sono esterne al programma (alla memoria heap). Grazie ai puntatori saremo in grado di modificare direttamente la risorsa esterna, al posto di doverne fare una copia. Il problema di questi puntatori è che se non deallocati correttamente potrebbero portare ad uno spreco della memoria heap, il che è un *memory leak*.

```
#include <iostream>

class Rectangle {
private:
    int width;
    int height;
};

void fun()
{
    Rectangle* p = new Rectangle();
}

int main()
{
    while(1)
    {
        fun();
    }
    //Output: Il problema è che quando la funzione fun termina,
    //il puntatore p verrà distrutto come fosse una variabile
    //locale, ma la memoria allocata non verrà deallocated,
```

perché ci siamo scordati di usare `delete p`; alla fine della funzione.

```
// Ciò è un problema perché verrà sempre allocata altra memoria e mai deallocated, occupando spazio, sprecando memoria, il che è un memory leak.
// L'intera memoria heap potrebbe diventare inutile per questo motivo.
return 0;
}
```

Il problema è che quando la funzione fun termina, il puntatore `p` verrà distrutto come fosse una variabile locale, ma la memoria allocata non verrà deallocated, perché ci siamo scordati di usare `delete p`; alla fine della funzione.
Ciò è un problema perché verrà sempre allocata altra memoria e mai deallocated, occupando spazio, sprecando memoria, il che è un *memory leak*.
L'intera memoria heap potrebbe diventare inutile per questo motivo.

Uno **smart pointer** è un *wrapper* (un wrapper è un'entità che ne encapsula un'altra; è del codice che letteralmente avvolge, incarta, impacchetta, confeziona dell'altro codice) su un puntatore con un'operatore `*` e `->` overloaded. La memoria allocata dinamicamente verrebbe così automaticamente liberata.

```
// Una generica classe Smart Pointer
#include <iostream>

template <class T>
class SmartPointer {
    T *ptr;
public:
    SmartPointer(T *ptr = NULL)
    {
        p = ptr;
    }

    ~SmartPointer()
    {
        delete ptr;
    }

    T & operator * ()
```

```

    {
        return *ptr;
    }

T * operator ->()
{
    return ptr;
}
};

int main()
{
    SmartPointer<int> p(new int());
    *p = 22;
    std::cout << "Valore di *p: " << *p << std::endl; //Output: Valore di *p: 22
    return 0;
}

```

unique pointers

Definizione: Gli **unique pointers** sono un tipo di **smart pointers** che memorizzano un solo puntatore alla volta.

Prima degli **unique_ptr** c'erano gli **auto_ptr** (C++98), ma dal C++11 sono *depreciati* (ovvero che non è raccomandato utilizzare, è obsoleto o ha bisogno di ulteriore sviluppo), quindi ora si consiglia di utilizzare gli **unique_ptr**. Sarà necessario includere **<memory>** per poter usufruire degli **unique pointers**.

```

#include <iostream>
#include <memory> // Per gli unique pointers, ecc..

// Per dichiarare un unique pointer
std::unique_ptr<int> p(new int(3));

```

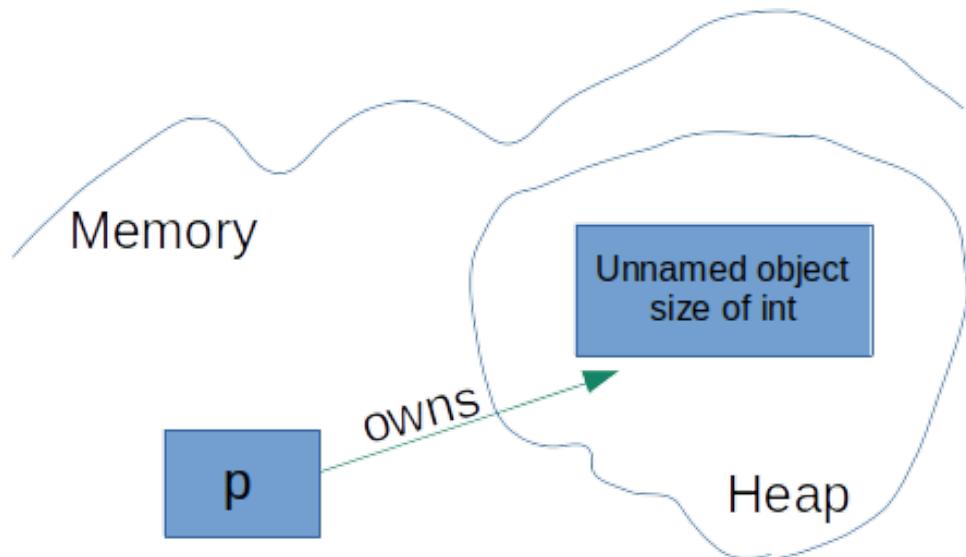


Figura 3.1: Unique ptr

Se lo `unique_ptr` viene distrutto, anche la memoria allocata nell'heap viene distrutta di conseguenza.

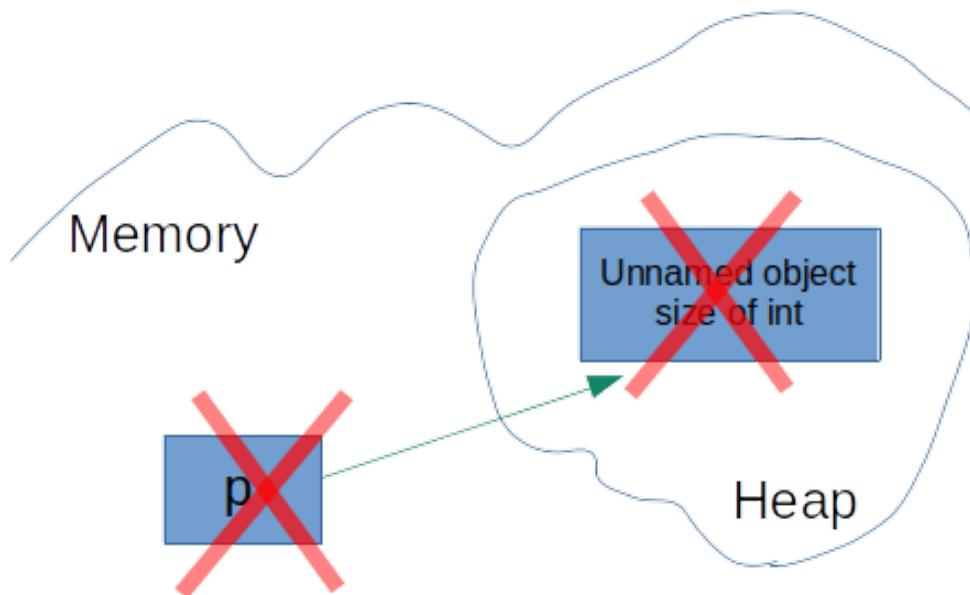


Figura 3.2: Unique ptr delete

Per creare uno `unique_ptr` si può anche utilizzare `std::make_unique`.

```

#include <iostream>

class Rectangle {
public:
    Rectangle(int w, int h)
    {
        this->width = w;
        this->height = h;
    }

    int area()
    {
        return width * height;
    }

private:
    int width;
    int height;
};

int main()
{
    auto pRect = std::make_unique<Rectangle>(3,
                                              4);
    std::cout << "Area del rettangolo: " << pRect
              ->area() << std::endl; //Output: Area del
              rettangolo: 12
    return 0;
}

```

Differenza tra std::make_unique vs std::unique_ptr

Ci sono varie ragioni per cui utilizzare **std::make_unique** al posto di **std::unique_ptr** con la new:

- È sicuro nel caso si vogliano creare dei temporanei, mentre con la new ti devi ricordare la regola: del non usare temporanei senza nome.
- Con l'utilizzo di **make_unique** si può finalmente evitare di usare la **new**, a differenza della vecchia regola: mai usare la **new** tranne per gli **unique_ptr**.

- Non richiede *type usage* ridondante: `unique_ptr<T>(new T()) -> make_unique<T>()`.
- Così da non dover esplicitare gli argomenti dei *template types*.
- Aggiunge sicurezza riguardo le eccezioni.
- Altrimenti non potresti accedere al costruttore della classe fuori dallo scope corrente.

Ownership | *move*

Un **unique pointer** è una relazione 1 a 1 con l'oggetto allocato. Non può essere copiato o passato per valore, però la **ownership** (proprietà) dell'oggetto può essere trasferita.

```
#include <iostream>

class Person {
public:
    Person(std::string s) : name(s) {};
    ~Person() { std::cout << "Libero spazio"
                << std::endl };

    std::string getName() { return this->name
                           };

private:
    std::string name;
};

int main()
{
    auto ptrPerson = std::make_unique<Person>("Luigi");

    std::cout << "Nome: " << ptrPerson->getName()
           << std::endl; //Output: Nome: Luigi

    std::unique_ptr<Person> ptrPerson2;

    ptrPerson2 = std::move(ptrPerson);
```

```

    std::cout << "Nome: " << ptrPerson2->getName
    () << std::endl; //Output: Nome: Luigi

    std::cout << "Nome dopo il trasferimento dell
    'ownership: " << ptrPerson->getName() <<
    std::endl; //Output: [non stampa niente]

    return 0;
}

```

```
auto ptrA = make_unique<Song>(L"Diana Krall", L"The Look of Love");
```



```
auto ptrB = std::move(ptrA);
```



Figura 3.3: Unique ptr move

Operazioni

sugli

unique_ptr

Varie operazioni sono supportate sugli `unique_ptr`:

- `*`: Dereferenza del puntatore.
- `->`: Accedere ai membri della classe.
- `.get()` : per ottenere il *raw pointer* del `unique_ptr` (non cancellarlo, perché è gestito dal unique pointer; è da usare solo per calcoli).
- `.reset(new int())` : cancella il vecchio oggetto e ne crea uno nuovo (al posto di `new int()` avremmo potuto passare qualsiasi altro oggetto, era per fare un esempio).
- `move` : trasferisce la proprietà del `unique_ptr`.
- `swap` : per scambiare due `unique pointers`.

- **if(unique_ptr)**: se passiamo uno **unique_ptr** all'if restituisce falso se non è associato a nessun oggetto.

Passare uno unique_ptr ad una funzione

Utilizziamo **std::move** per trasferire la proprietà del **unique_ptr**.

```
#include <iostream>
#include<memory>

struct A {
    int x;
    ~A() { std::cout << "Libero spazio" << std::endl; }
};

void passUniquePtr( std::unique_ptr<A> a )
{
    // Usciti dalla funzione lo unique_ptr e il suo oggetto
    // vengono cancellati, perché locali alla funzione.
    std::cout << "Puntatore ricevuto" << '\n';
    a->x = 5;
    std::cout << "a.x: " << a->x << std::endl;
}

int main()
{
    auto ptrA = std::make_unique<A>();
    passUniquePtr( std::move(ptrA) );

    // true = ptrA è vuoto.
    if (!ptrA)
    {
        std::cout << "ptrA è vuoto" << std::endl;
    }

    //Output: Puntatore ricevuto
    //Output: a.x: 5
    //Output: Libero spazio
    //Output: ptrA è vuoto
    return 0;
}
```

```
}
```

Restituire un unique_ptr

Si può restituire uno `unique_ptr` da una funzione.

```
#include <iostream>
#include <memory>

class A {};

std::unique_ptr<A> returnUniquePtr()
{
    auto a = std::make_unique<A>();
    return a;
}

int main()
{
    auto ptrA = returnUniquePtr();

    if (ptrA)
    {
        std::cout << "ptrA ha un oggetto. " <<
                    std::endl;
    }

    //Output: ptrA ha un oggetto.
    return 0;
}
```

Membri delle classi: unique pointer vs raw pointer vs reference

- **Unique pointer membro della classe** : la classe è la proprietaria dell'oggetto del puntatore.
- **Raw pointer membro della classe** : La classe è un osservatore e non è responsabile di rimuovere l'oggetto puntato dal puntatore. Viene rimosso da uno smart pointer fuori dalla classe.

- **Referenza membro della classe** : è garantito che la referenza contiene dati validi mentre la classe è "viva".

share pointers

Definizione: Gli **shared pointers** sono un tipo di **smart pointers** dove più di un puntatore può puntare allo stesso oggetto e un contatore (*Reference Counter*) verrà mantenuto di conseguenza.

Abbiamo sempre bisogno di includere l'header `<memory>` per poterlo utilizzare.

```
#include <iostream>
#include <memory>

class A {
public:
    int x;
    A(int x) : x(x) {};
};

int main()
{
    auto sharedPtr1 = std::make_shared<A>(7); // 
    oppure si può anche fare: std::shared_ptr<A>
    sharedPtr1(new A7);

    std::shared_ptr<A> sharedPtr2 = sharedPtr1;
    std::shared_ptr<A> sharedPtr3 = sharedPtr1;

    // Tutti e tre gli shared1_ptr puntano allo stesso oggetto.
    return 0;
}
```

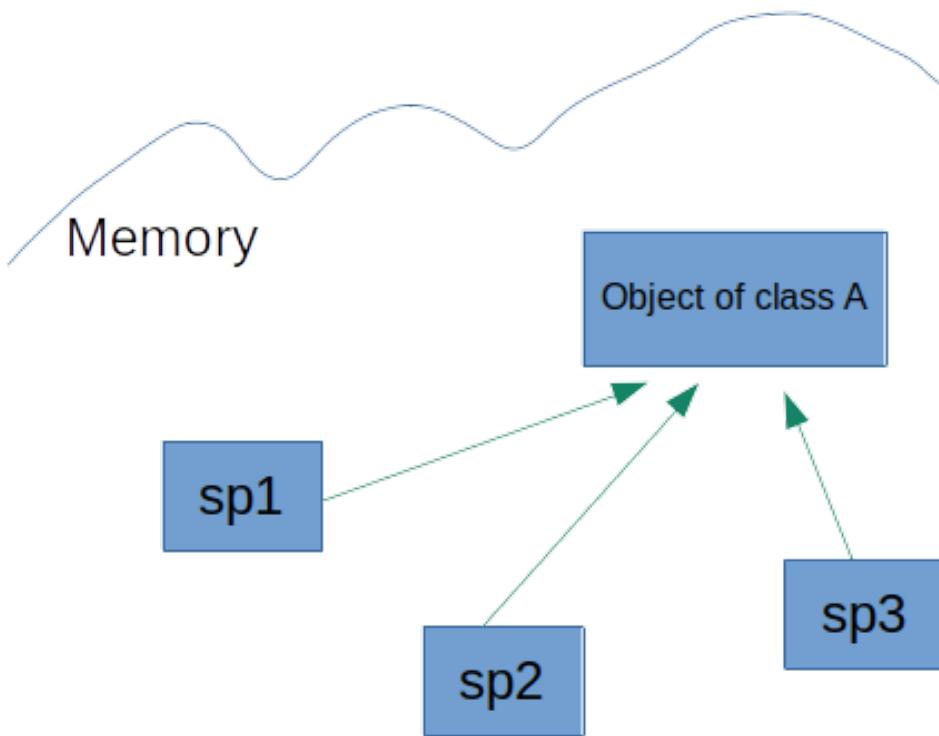


Figura 3.4: Shared ptr

Possiamo creare uno **shared pointer** sia con **shared_ptr** sia con **make_shared**.

Differenza tra std::shared_ptr vs std::make_shared

Una delle differenze tra questi due è che **make_shared** esegue una sola
allocazione nell'heap, mentre **shared_ptr** ne fa due.
shared_ptr si occupa di due entità:

- Il blocco di controllo (*control block*) che memorizza dei metadati come *ref-counts* (contatore delle referenze all'oggetto), *type-erased deleter*, ecc..
- L'oggetto stesso.

std::make_shared esegue una singola allocazione nell'heap per lo spazio necessario
sia per il *control block* sia per l'oggetto.

Inoltre **std::make_shared** è *exception-safe* (sicuro per quanto riguarda le
eccezioni).

Per di più, **make_shared** sfrutta dell'ottimizzazione conosciuta come *We know Where You Live* che permette al *control block* di essere un piccolo puntatore, quindi **make_shared** non solo evita un'ulteriore allocazione, ma alloca anche meno memoria totale.

Un problema che potrebbe esserci per quanto riguarda **std::make_shared** è che visto che fa una singola allocazione, non c'è modo di deallocare la memoria del *control block* e dell'oggetto in modo indipendente. Un altro svantaggio, di conseguenza è che essendoci una singola allocazione, la memoria non può essere deallocata finché il *control block* non è più usato.

Operazioni *sui* *shared* *pointers*

Sono possibili varie operazioni sugli **shared pointers**:

- `(*nomepuntatore).variabile` : dereferenza
- `nomepuntatore->variabile` : dereferenza come sopra
- `.get()` : per poter accedere al *raw pointer* chiamato **stored pointer**.
- `use_count()` : per ottenere il numero di **shared_ptr** che puntano allo stesso oggetto.
- `.reset()` : scollega e svuota il puntatore.

Lo **shared pointer** inoltre allo *stored pointer*, possiede un secondo puntatore che punta ad un **control block**. Il *control block* ha un *reference counter* (contatore delle referenze) che memorizza il numero di **shared pointers** che puntano allo stesso oggetto.

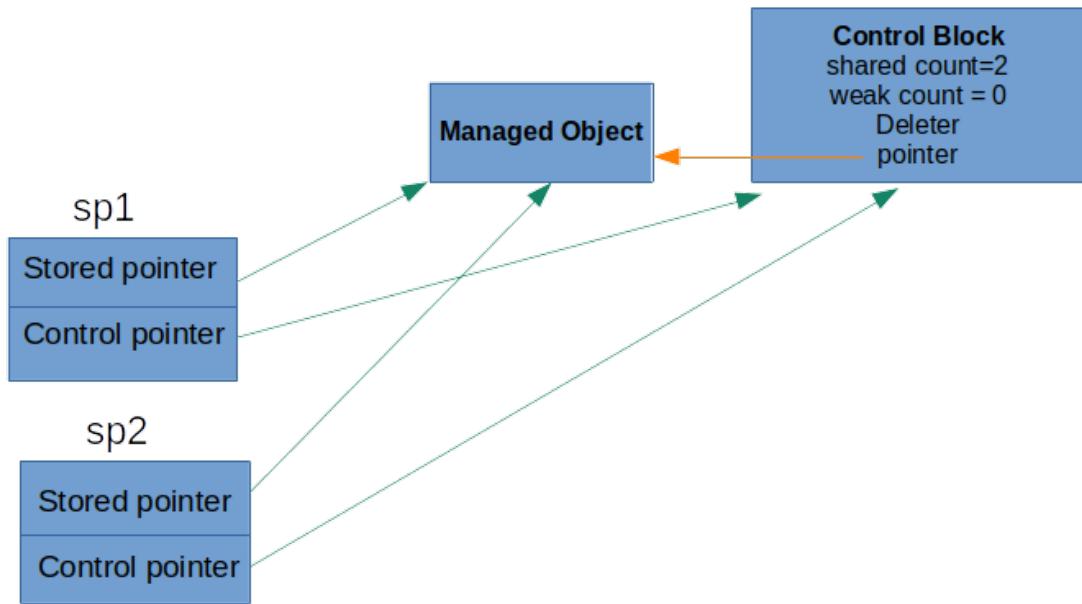


Figura 3.5: Shared ptr structure

Distruzione degli shared pointers

Quando verrà eliminato l'oggetto gestito dagli **shared_ptr**?

Quando uno **shared pointer** viene distrutto, allora il *control block* decrementerà il *reference counter*.

L'oggetto verrà eliminato quando l'ultimo **shared_ptr** verrà eliminato.

```
#include <iostream>
#include <memory>

class A {
public:
    int x;
    A(int x) : x(x) {}
};

int main()
{
    auto shrPtr1 = std::make_shared<A>(5);
    auto shrPtr2 = shrPtr1;
    auto shrPtr3 = shrPtr1;
```

```
{  
    auto shrPtr4 = shrPtr1;  
    std::cout << shrPtr1.use_count() << std:::  
        endl; //Output: 4  
}  
  
std::cout << shrPtr1.use_count() << std::endl  
; //Output: 3  
  
shrPtr3 = std::make_shared<A>(8); // shrPtr3  
    punta ad un altro oggetto.x  
  
shrPtr2.reset(); // shrPtr2 scollegato e svuotato.  
  
std::cout << shrPtr1.use_count() << std::endl  
; //Output: 1  
return 0;  
}
```

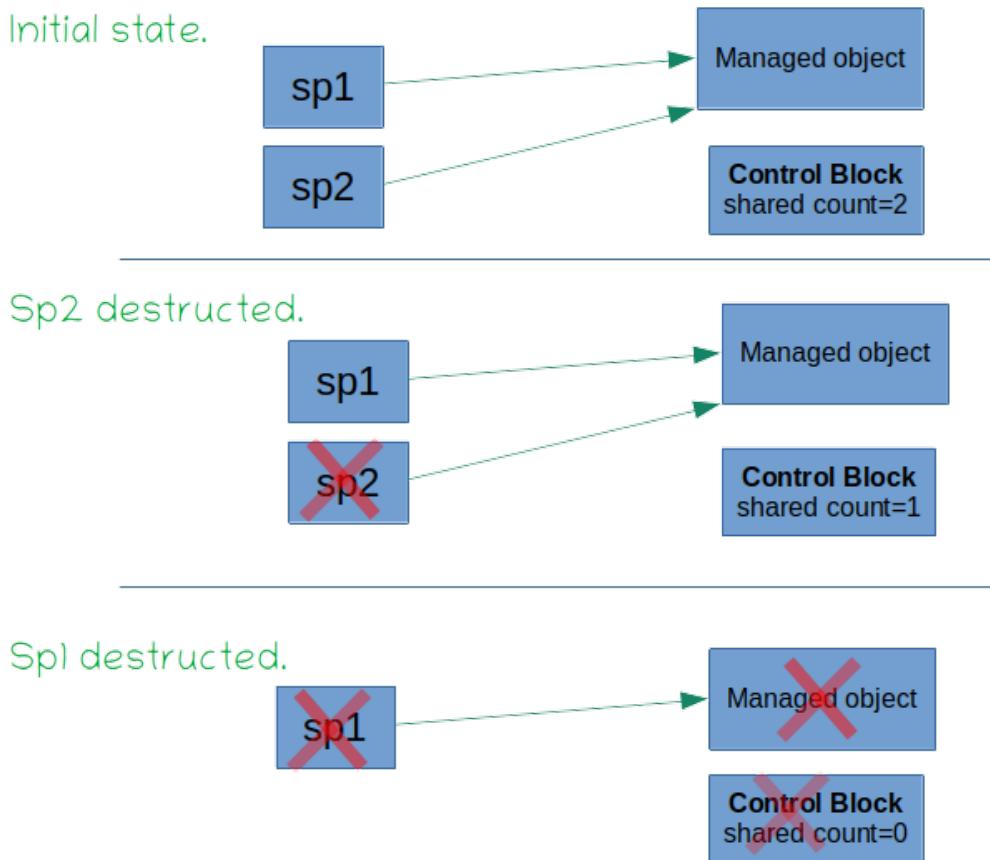


Figura 3.6: Shared ptr destruction

Passare gli shared pointers ad una funzione

Se una funzione vuole l'*ownership* (proprietà) su un `shared_ptr`, possiamo passarlo per valore:

```
#include <iostream>
#include <memory>

void function( std::shared_ptr<int> sp )
{
    std::cout << sp.use_count() << std::endl;
}

int main()
{
```

```

auto sp1 = std::make_shared<int>(6);
std::cout << sp1.use_count() << std::endl; // Output: 1

function(sp1); //Output: 2

std::cout << sp1.use_count() << std::endl; // Output: 1 (lo shared_ptr nella funzione "function" viene distrutto una volta usciti da essa, perché è locale alla funzione)
return 0;
}

```

Restituire gli shared pointers

Una funzione può restituire `shared_ptr` per valore:

```

#include <iostream>
#include <memory>

std::shared_ptr<int> function()
{
    auto sp = std::make_shared<int>(9);
    return sp;
}

int main()
{
    auto sp1 = function(); // Lo shared_ptr dentro alla funzione non esiste più, però viene ritornato e recuperato nella variabile sp1.

    std::cout << sp1.use_count() << std::endl; // Output: 1
    std::cout << *sp1 << std::endl; //Output: 9
    return 0;
}

```

Un problema col restituire uno `shared_ptr`, se lo devi restituire al "mondo esterno", meglio restituirlo come `reference`, perché altrimenti uno potrebbe chiamare la `.reset()` su quello `shared pointer`.

Un altro modo per restituire uno `shared_ptr` è attraverso `std::allocate_shared`.

```
#include <iostream>
#include <memory>

int main () {
    std :: allocator<int> alloc;      // allocatore di
    default per gli int.
    std :: default_delete<int> del; // deleter di default
    per gli int.

    std :: shared_ptr<int> foo = std :: 
        allocate_shared<int> (alloc ,12);

    auto bar = std :: allocate_shared<int> (alloc
        ,24);

    auto baz = std :: allocate_shared<std :: pair<int
        ,int>> (alloc ,33 ,44);

    std :: cout << "*foo: " << *foo << '\n'; //
    Output: *foo: 12
    std :: cout << "*bar: " << *bar << '\n'; //
    Output: *bar: 24
    std :: cout << "*baz: " << baz->first << ' ' <<
        baz->second << '\n'; //Output: *baz: 33 44

    return 0;
}
```

static_pointer_cast

Definizione: `static_pointer_cast` è una funzione, non una keyword e restituisce uno `shared_ptr` che possiede e contiene un puntatore all'oggetto costruito.

Se il parametro passato non è vuoto, ciò che viene restituito condivide la proprietà con il parametro passato, quindi il contatore viene incrementato di 1.

Se il parametro è vuoto (non possiede nulla), allora l'oggetto ritornato è uno `shared_ptr` vuoto.

```
#include <iostream>
#include <memory>
```

```
class Base {};

class Derived : public Base {
public:
    void print()
    {
        std::cout << "Hello World!" << std::
            endl;
    }
};

int main()
{
    std::shared_ptr<Base> spBase(std::make_shared
        <Derived>());

    std::static_pointer_cast<Derived>(spBase)->
        print();

    static_cast<Derived*>(spBase.get())->print();

    //Output: Hello World!
    //Output: Hello World!
    return 0;
}
```

```

class BarHolder
{
public:
    BarHolder( std::shared_ptr<Bar> pb )
        :
        pBar( std::move( pb ) )
    {}
private:
    std::shared_ptr<Bar> pBar;
};

std::shared_ptr<Foo> factory()
{
    return std::make_shared<Bar>();
}

int main()
{
{
    BarHolder fh{ std::static_pointer_cast<Bar>( factory() ) };

    while( !_kbhit() );
}

```

Figura 3.7: Static pointer cast

enable_shared_from_this

Definizione: `std::enable_shared_from_this` permette un oggetto *t* che è gestito da uno `shared_ptr` *sp* di generare, in modo sicuro, degli `shared_ptr` addizionali, istanze *sp1*, *sp2* che condividono tutti la proprietà dell'oggetto *t* con *sp*.

Ereditare da `std::enable_shared_from_this` fornisce il tipo *T* con una funzione membro `shared_from_this()`. Se un oggetto *t* di tipo *T* è gestito da uno `shared_ptr` *sp*, allora chiamare `shared_from_this()` restituirà un nuovo `shared_ptr` che condivide la proprietà di *t* con *sp*.

```

#include <iostream>
#include <memory>

struct B : std::enable_shared_from_this<B> {};

int main()

```

```

{
    std::shared_ptr<B> foo, bar;

    foo = std::make_shared<B>();

    bar = foo->shared_from_this();

    if (!foo.owner_before(bar) && !bar.
        owner_before(foo))
    {
        std::cout << "foo e bar condividono la
                     proprietà" << std::endl;
    }

    //Output: foo e bar condividono la proprietà

    return 0;
}

```

Inoltre, c'è anche una funzione `weak_from_this` che restituisce un `weak_ptr` che tiene traccia della proprietà di `*this` da parte di tutte le istanze di `shared_ptr`.

Performance degli shared pointers

Uno **shared pointer** ha bisogno di due **raw pointers**. Un insieme di **shared pointers** hanno bisogno di essere gestiti da una **control unit** (*control block*). Quindi la memoria che uno **shared pointer** occupa è maggiore dei **raw** e degli **unique pointers**.

Shared pointers, unique pointers or raw pointers

Se un oggetto ha bisogno di un singolo proprietario per tutta la durata del programma e possiamo immaginare il puntatore come un'entità singola, allora usiamo un **unique pointer**. Per delle performance più buone gli **unique pointers** sono migliori rispetto agli **shared pointers**.

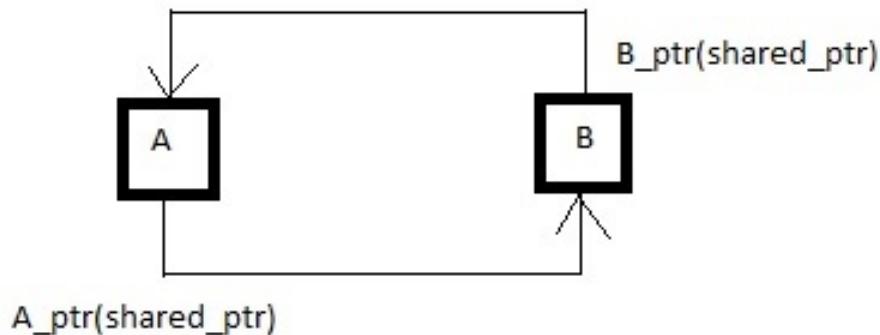
Gli **shared pointers** sono utili dove non abbiamo bisogno di pensare alle *performance, ownership e lifetime* degli oggetti.

weak pointers

Definizione: I **weak pointers** sono un tipo di **smart pointers** che non prende la proprietà dell'oggetto, ma agisce come un osservatore. Non partecipa al *reference counter* (non viene contato) e non estende la *lifetime* dell'oggetto. Sono usati, principalmente, per rompere la dipendenza circolare degli **shared pointers**.

Problema della dipendenza ciclica

Definizione: Mettiamo di avere due classi: A e B, se entrambe hanno un puntatore che punta all'altra, avremo un ciclo e il *use_count()* non arriverà mai a 0, il che crea un problema nella rimozione di questi due puntatori.



Circular Reference

Figura 3.8: Cyclic dependency

Per questo motivo usiamo gli **weak pointers**, perché non vengono conteggiati nel *reference counter*. Possono, comunque avere accesso all'oggetto.

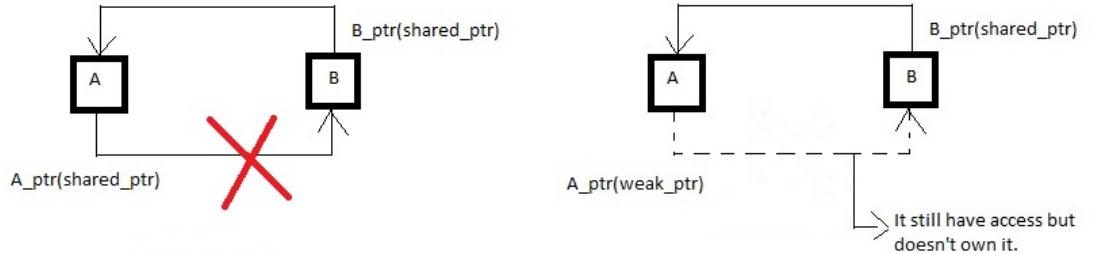


Figura 3.9: Cyclic dependency

Quindi, il problema della *dipendenza ciclica* si risolve con l'utilizzo degli **weak pointers**.

Quando usare i weak pointers?

Quando usare i weak pointers?

Quando vuoi riferire al tuo oggetto da molteplici posti, per quelle referenze dove non è okay ignorarle e dealloccarle.

Operazioni sugli weak pointers

Ci sono varie operazioni sui **weak pointers**:

- * : dereferenza
- -> : dereferenza, accedere ai membri della classe/struct, ecc..
- .lock() : restituisce uno **shared_ptr** con le informazioni preservate nel **weak_ptr** se non è *expired*. Se il **weak_ptr** è *expired* (scaduto), la funzione restituisce un **shared_ptr** vuoto.
- .reset() : cancella il vecchio oggetto e ne crea uno nuovo.
- .swap() : scambia due **weak pointers**.
- .use_count() : restituisce il numero di **shared pointers** che puntano allo stesso oggetto.
- .expired() : restituisce se il **weak_ptr** è vuoto o non ci sono più **shared_ptr** nel *owner group*. I puntatori "scaduti" (*expired*) sono come **weak pointers** vuoti quando *locked*, e non possono quindi essere più usati.

- `.owner-before()` : restituisce se l'oggetto deve andare prima del parametro. Se l'oggetto appartiene allo stesso *owner group* del parametro, allora restituisce *false*, anche se il valore memorizzato dai puntatori è diverso.

```
#include <iostream>
#include <memory>

class Person {
public:
    std::string name;
    Person(std::string name) : name(name) {};
};

int main()
{
    std::weak_ptr<Person> wp;

    auto teacher = std::make_shared<Person>("Giorgio");

    wp = teacher;

    // Per controllare se l'oggetto è ancora lì o no.
    // lock() restituisce un shared_ptr temporaneo.
    if (auto temp = wp.lock())
    {
        std::cout << temp->name << std::endl;
    } else {
        std::cout << "L'oggetto non c'è più" << std::endl;
    }
    return 0;
}
```

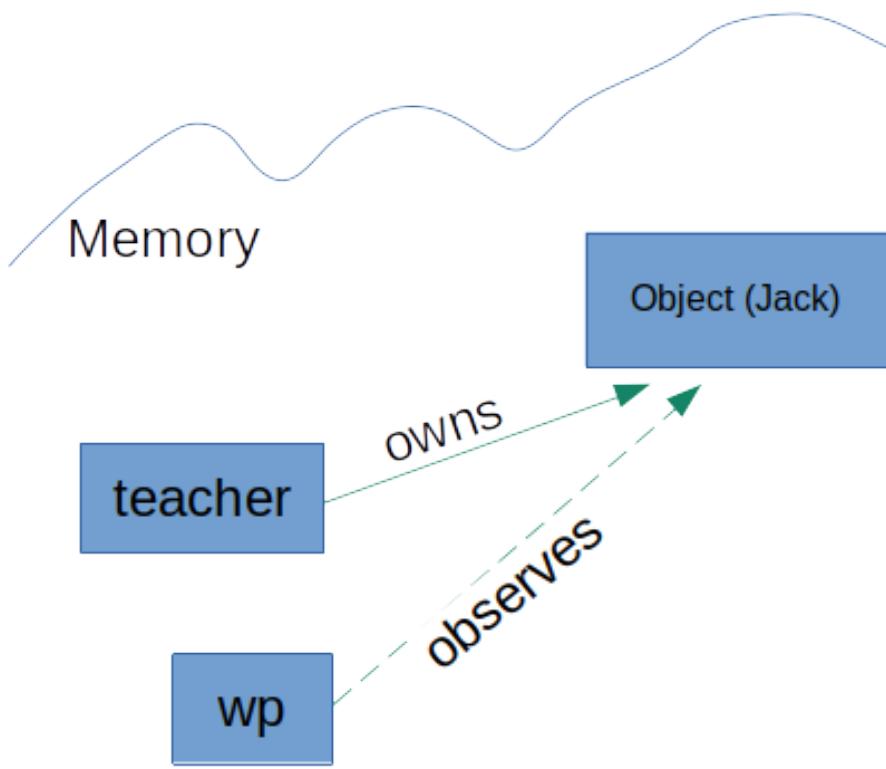


Figura 3.10: Weak pointers

Uno **shared pointer** ha un **control block** che conta il numero di **shared pointers** e di **weak pointers**. Quando il contatore degli **shared pointers** arriva a 0, l'oggetto viene eliminato, ma il **control block** resta vivo finché il contatore dei **weak pointers** non raggiunge 0.

Uniform Real Distribution

Definizione: La **uniform real distribution**, in probabilità, si riferisce alla distribuzione nella quale le probabilità sono definite da una variabile casuale continua, una che può prendere qualsiasi valore tra due numeri, così la distribuzione è detta essere una distribuzione probabilistica continua.

Per poter usufruire della **uniform real distribution** sarà necessario includere `<random>`.

Questa ha diverse funzioni:

- **a()** : restituisce il limite inferiore.
- **b()** : restituisce il limite superiore.
- **min()** : restituisce il valore minimo che la funzione può restituire.
- **max()** : restituisce il valore massimo che la funzione può restituire.
- **reset()** : si resetta la distribuzione affinché la prossima non sia basata sui valori precedenti.

```
#include <iostream>
#include <random>

int main()
{
    double a = 0, b = 2.0;

    std::uniform_real_distribution<double>
        distribution(a, b);

    std::cout << "limite inferiore: " <<
        distribution.a() << std::endl; //Output:
        limite inferiore: 0
    std::cout << "limite superiore: " <<
        distribution.b() << std::endl; //Output:
        limite superiore 2.0
```

```

    std::cout << "distribuzione min: " <<
        distribution.min() << std::endl; //
    distribuzione min: 0
    std::cout << "distribuzione max: " <<
        distribution.max() << std::endl; //
    distribuzione max: 2.0

    distribution.reset();
    return 0;
}

```

Come detto, in precedenza, può essere utile nella generazione di numeri casuali.

Copy-And-Swap Idiom

Cos'è il copy-and-swap idiom e a cosa serve?

Qualsiasi classe che gestisce una risorsa (*un wrapper*, come uno **smart pointer**) ha bisogno di implementare la *Rule of Three* (destructor, copy constructor, copy assignment). La *copy-and-swap* assiste il *copy assignment* in due cose: evitare la duplicazione di codice e fornisce una forte sicurezza a livello di eccezioni (*exception safety*).

Concettualmente funziona utilizzando la funzionalità del *copy constructor* di creare una copia locale dei dati, prendere questa copia con una funzione *swap*, scambiando i vecchi dati con quelli nuovi.

La copia temporanea viene distrutta, portandosi i vecchi dati con essa.

Restiamo con una copia dei nuovi dati.

Per poter far questo quindi avremo bisogno di:

- copy-constructor
- destructor
- swap function

```
#include <iostream>
#include <cstring>
```

```

class anyArrayClass
{
    int size;
    int *ptr;
    public:
        anyArrayClass(int s=0): size(s),
        ptr(size? new int[size]: nullptr) {}

    // Copy constructor
    anyArrayClass(const anyArrayClass& obj): size(
        obj.size),
        ptr(size? new int[size]: nullptr)
    {
        memmove(ptr, obj.ptr, size*sizeof(int));
    }

    friend void swap(anyArrayClass& obj1,
        anyArrayClass& obj2)
    {
        std::swap(obj1.size, obj2.size);
        std::swap(obj1.ptr, obj2.ptr);
    }

    // overloaded assignment operator
    // argomento passato per valore. chiama il copy constructor
    anyArrayClass& operator=(anyArrayClass obj)
    {
        // calling friend function
        swap(*this, obj);
        return *this;
    }

    ~anyArrayClass()
    {
        delete[] ptr;
    }
};

```

I vantaggi sono:

- Non c'è più bisogno di controlli per l'assegnamento a se stesso visto che il

parametro è passato per valore. Inoltre l'allocazione a se stesso è molto rara quindi l'*overhead* della copiatura non dovrebbe essere un problema.

- Ora il *copy constructor* è usato per creare un oggetto temporaneo, lo swapping (scambio) avverrà soltanto se l'oggetto temporaneo è stato creato affatto. Praticamente, quello che stiamo facendo manualmente, il compilatore lo sta facendo per noi.
- Riusabilità del codice: l'operatore *operator=()* non ha molto codice nel suo corpo, anzi stiamo usando il *copy constructor* e la funzione *swap* per fare il lavoro.

Tecniche per il debugging

Definizione: Debugging si riferisce alle tecniche impiegate dai programmatori per cercare di correggere gli errori, i bugs di un programma.

Una metodologia di base è questa:

- Osserva il problema.
- Identifica la natura del problema.
- Identifica la condizione (o condizioni) che innesca il problema.
- Identifica il meccanismo di guasto, fallimento.
- Fixa il problema.

Per fare questo potrebbe tornare utile:

- Commentare il codice.
- Validare il flusso del codice, programma (*code flow*).
- Controllare valori stampandoli a schermo.

Puntatore a funzione

Definizione: I puntatori a funzione o function pointers sono dei puntatori che puntano a delle funzioni.

```
#include <iostream>

void fun(int a)
{
    std::cout << "Valore di a: " << a << std::endl;
}

int main()
{
    // fun_ptr è un puntatore alla funzione fun() (quella sopra)
    void (*fun_ptr)(int) = &fun;

    /* La riga sopra è equivalente alle seguenti
       due sotto:
    void (*fun_ptr)(int);
    fun_ptr = &fun;
    */

    // Chiamando fun() attraverso fun_ptr
    (*fun_ptr)(10);

    return 0;
}
```

Si può anche scrivere così:

```
#include <iostream>

void fun(int a)
{
    std::cout << "Il valore di a: " << a << std::endl;
```

```

}

int main()
{
    void (*fun_ptr)(int) = fun; // & rimosso

    fun_ptr(10); /* rimosso

    return 0;
}

```

Alcuni fatti riguardo i **function pointers** dal C:

- Questi puntano all'inizio del codice eseguibile.
- A differenza dei normali puntatori, non allochiamo e deallochiamo memoria usando i **function pointers**.
- Il nome della funzione può anche essere usato per ottenere l'indirizzo della funzione.
- Proprio come i puntatori normali, possiamo avere un array di **function pointers**.
- Possono essere usati al posto degli switch.
- Come i puntatori normali anche questi possono essere passati come argomenti e restituiti dalle funzioni.
- Alcune funzionalità del C++ sono implementate usando i **function pointers**, come ad esempio le **funzioni virtuali**.

```

#include <iostream>

void add(int a, int b)
{
    std::cout << "Addizione: " << a + b << std::
        endl;
}
void subtract(int a, int b)
{
    std::cout << "Sottrazione: " << a - b << std
        ::endl;
}

```

```

}

void multiply(int a, int b)
{
    std::cout << "Moltiplicazione: " << a * b <<
        std::endl;
}

int main()
{
    // fun_ptr_arr è un array di function pointers (puntatori a
    // funzioni).
    void (*fun_ptr_arr[]) (int, int) = {add,
        subtract, multiply};
    unsigned int ch, a = 7, b = 5;

    printf("Enter Choice: 0 for add, 1 for
        subtract and 2 "
    "for multiply\n");
    std::cout << "0. Aggiungi" << "\n";
    std::cout << "1. Riduzione" << "\n";
    std::cout << "2. Moltiplicazione" << "\n";
    std::cout << "Scelta: ";

    std::cin >> ch;

    if (ch > 2) return 0;

    (*fun_ptr_arr[ch])(a, b);

    return 0;
}

```

7 Concetti Avanzati

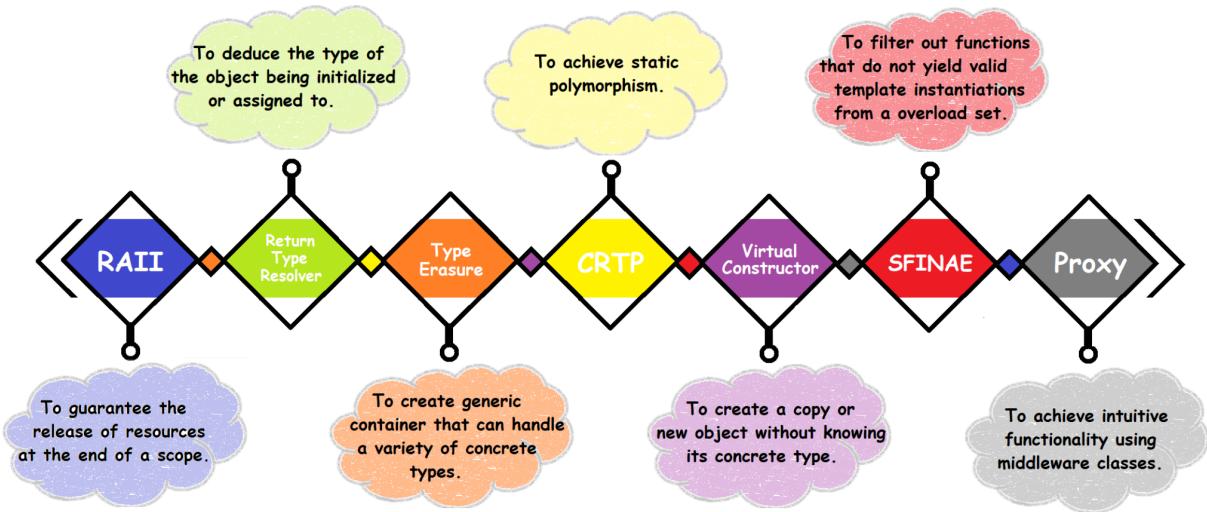


Figura 3.11: 7 Concetti Avanzati

RAII

Abbiamo già trattato questo argomento nel precedente capitolo *Concetti Intermedi*, ma lo riprenderemo qui visto che è uno dei 7 Concetti Avanzati del linguaggio.

Definizione: RAII (*Resource Acquisition is Initialization*), o anche conosciuta come *Execute-around object*, *Resource release is finalization*, *Scope-bound resource management* è una tecnica di programmazione il cui intento è quello di rilasciare le risorse precedentemente allocate alla fine dello scope.

Implementiamo questo attraverso: il *wrapping* delle risorse nella classe; acquisizione della risorse nel costruttore immediatamente dopo l'allocazione e automaticamente rilasciata nel distruttore; risorse usate attraverso un'interfaccia della classe.

```
#include <iostream>
```

```
class Resource {
```

```

public:
    Resource(int x, int y) { std::cout << "risorsa acquisita\n"; }
    ~Resource() { std::cout << "risorsa distrutta\n"; }
};

void func()
{
    Resource *ptr = new Resource(1, 2);
    int x;

    std::cout << "Inserisci un numero: ";
    std::cin >> x;

    if(x == 0)
    {
        // La funzione restituisce prima del return e quindi ptr
        // non verrà deallocated, cancellato!
        throw 0;
    } else if(x < 0)
        // La funzione restituisce prima della deallocazione,
        // quindi ptr non verrà deallocated!
        return;
    }
    delete ptr;
}

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

```

Nel codice sopra, il *return* o il *throw* avvengono prima della *delete* quindi ptr non viene deallocated e la sua memoria è *leaked* ogni volta che la funzione viene chiamata.

```

#include <iostream>

template<class T>
class smart_ptr

```

```

{
    T* m_ptr;
public:
    template<typename ... Args>
    smart_ptr(Args&&... args) : m_ptr(new T(
        std::forward<Args>(args)...)) {}
    ~smart_ptr() { delete m_ptr; }

    smart_ptr(const smart_ptr& rhs) = delete;
    smart_ptr& operator=(const smart_ptr& rhs
        ) = delete;

    smart_ptr(smart_ptr&& rhs) : m_ptr(
        exchange(rhs.m_ptr, nullptr)) {}
    smart_ptr& operator=(smart_ptr&& rhs){
        if (&rhs == this) return *this;
        delete m_ptr;
        m_ptr = exchange(rhs.m_ptr, nullptr);
        return *this;
    }

    T& operator*() const { return *m_ptr; }
    T* operator->() const { return m_ptr; }
};

void func()
{
    auto ptr = smart_ptr<Resource>(1, 2); // ora
        ptr garantisce il rilascio della risorsa (resource).
    // ...
}

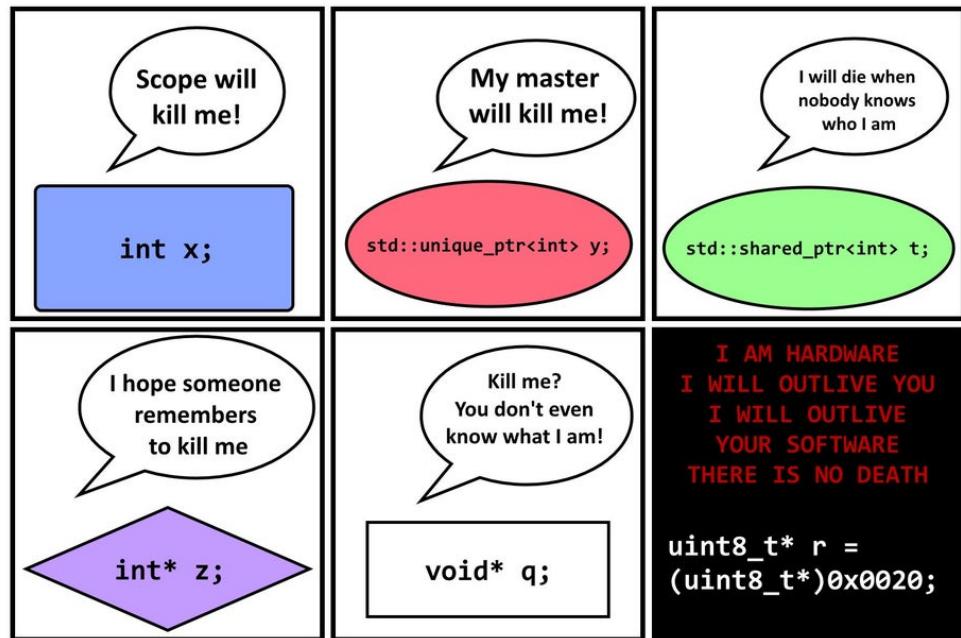
int main()
{
    func();
    return 0;
}

```

Ora non importa cosa succede dopo la dichiarazione di *ptr*, verrà comunque distrutto quando la funzione termina (a prescindere da come termina). Visto che *ptr* è un oggetto locale, il distruttore verrà chiamato mentre la

funzione riavvolge indietro lo stack (*function stack frame*). Perciò siamo sicuri che la risorsa verrà appropriatamente liberata.

Death and Memory (C++ Stories)



2017 Ólafur Waage (@olafurw)
with thanks to Frank A. Krueger (@praeclarum)

Figura 3.12: La morte della memoria e RAI

Return Type Resolver

Definizione: Il **Return Type Resolver**, o anche conosciuto come *Return type overloading* è un idioma il cui intento è quello di dedurre il tipo della variabile inizializzata o da assegnare.

Il problema:

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

int fromString(const char *str) { return std::
    stoi(str); }
float fromString(const char* str) { return std::
    stoi(str); } // Errore
```

Una funzione non può essere *overloadata* solo dal suo tipo di ritorno.

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

class fromString {
    const std::string str;
public:
    fromString(const char* str) : str(str) {}
    template<typename type>
    operator type() {
        if constexpr(std::is_same_v<type, float>) return stof(str);
        else if(std::is_same_v<type, int>) return stoi(str);
    }
};

int main()
{
    // Funzionerà con C++17 e oltre perché ha is_same_v
    int iStr = fromString("321");
    float fStr = fromString("654.987");
    return 0;
}
```

- Quando usi `nullptr` serve per dedurre il corretto tipo dipendendo dal puntatore a cui sta assegnando.
- Si può anche superare questa limitazione nel modo in cui abbiamo visto sopra.
- **Return Type Resolver** può anche essere usato per fornire un'interfaccia generale per l'assegnamento, indipendente dall'oggetto a cui è stato assegnato.

Type Erasure

Definizione: **Type Erasure**, o anche conosciuto come *Duck-typing* serve per generare dei container che possono gestire una varietà di tipi.

Questo può essere implementato attraverso: *void**, *templates*, *polimorfismo*, *union*, *proxy class*, eccetera..

- Il C++ è un linguaggio *statically typed* con uno *typing* forte. In questi tipi di linguaggio, la tipologia dell'oggetto è conosciuta a compile-time.
- Quindi il tipo di un oggetto, dopo la compilazione, non cambia.
- Per ovviare a questo esistono vari containers come: *std::any* (C++17), *std::variant* (C++17), *std::function* (C++11), eccetera..

Quindi esistono varie tecniche per implementare questo:

Type erasure usando void (come nel C)*

```
void qsort(void* base, size_t num, size_t size,
           int (*comapare)(const void*, const void*));
```

Lo svantaggio è che non è sicuro e separa la funzione compare necessaria per ogni tipo.

Type erasure usando i templates

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first,
          RandomAccessIterator last);
```

Lo svantaggio: potrebbe portare all'istanziamento di molte funzioni template e quindi di tempi di compilazione più lunghi.

Type erasure usando il polimorfismo

```
class Base {
public:
    virtual void method() = 0;
};

class C : Base {
public:
    void method() { cout << "classe C\n"; }
```

```

};

class D : Base {
public:
    void method() { cout << "classe D\n"; }
};

// Non vediamo un tipo concreto (viene cancellato) anche se lo si
// può dynamic_cast.
void call(Base* ptr)
{
    ptr->method();
};

```

Lo svantaggio è che costa a livello di runtime (dynamic dispatch, indirection, vtable, eccetera..)

<i>Type</i>	<i>erasure</i>	<i>usando</i>	<i>le</i>	<i>union</i>
-------------	----------------	---------------	-----------	--------------

```

class Data {
public:
    int x;
};

union U {
    Data d;           // occupa 1 byte
    std::int32_t n;  // occupa 4 bytes
    char c;          // occupa 1 byte

    ~U() {}          // ha bisogno di sapere il tipo
                      // correntemente attivo.
}; // un'istanza di U occupa in totale 4 bytes.

int main()
{
    U u; // creo un'istanza chiamata u della union U.

    u.c = 's';
    u.d.x = 3;
}

```

```

    std::cout << "u.c: " << u.c << std::endl; //  

        Output: u.c: s  

    std::cout << "u.d.x: " << u.d.x << std::endl;  

        //Output: u.d.x: 3  

    return 0;  

}

```

Lo svantaggio è che non è sicuro a livello di tipo (*type-safe*).

Type erasure usando i containers generici

```

#include <iostream>
#include <memory>

struct any
{
    struct base {};

    template<typename T>
    struct inner: base{
        inner(T t): m_t{std::move(t)} {}
        T m_t;
        static void type() {}
    };

    any(): m_ptr{nullptr}, typePtr{nullptr} {}

    template<typename T>
    any(T && t): m_ptr{std::make_unique<inner<T>>(t)}, typePtr{&inner<T>::type} {}

    template<typename T>
    any& operator=(T&& t) {
        m_ptr = std::make_unique<inner<T>>(t);
        typePtr = &inner<T>::type;
        return *this;
    }

    private:
    template<typename T>
    friend T& any_cast(const any& var);

```

```

    std :: unique_ptr<base> m_ptr = nullptr;
    void (* typePtr)() = nullptr;
};

template<typename T>
T& any_cast(const any& var)
{
    if(var.typePtr == any::inner<T>::type)
        return static_cast<any::inner<T>*>(var.m_ptr.
            get())->m_t;
    throw std::logic_error{ "Bad cast!" };
}

int main()
{
    any var(10);
    std::cout << any_cast<int>(var) << std::endl;

    var = std::string{ "some text" };
    std::cout << any_cast<std::string>(var) <<
        std::endl;

    return 0;
}

```

Da notare come stiamo usando la funzione statica vuota `inner<T>::type` per determinare l'istanza del template in `any_cast<T>`.

Questo è usato per gestire molteplici tipi di valori di ritorno dalle funzioni
(Anche se non è raccomandato).

Implementation	Arbitrary Datatype	Type Safe	Easy to Implement	Common Base Class
<code>void*</code>	Yes	No	Yes	No
<code>OO</code>	No	Yes	Yes	Yes
<code>Templates</code>	Yes	Yes	No	No

Figura 3.13: Type Erasure

CRT_P

Definizione: Il CRT_P (*Curiously Recurring Template Pattern*), o anche conosciuto come *Upside-down inheritance* o *Static polymorphism* è utilizzato per ottenere il polimorfismo statico.

Per ottenere questo fa uso della specializzazione delle classi template (*base class template specialization*).

Problema:

```
#include <iostream>

struct obj_type_1
{
    bool operator<(const value &rhs) const {
        return m_x < rhs.m_x;}
    // bool operator==(const value &rhs) const;
    // bool operator!=(const value &rhs) const;
    // e la lista va avanti..
    private:
        // dati membri da comparare
};

struct obj_type_2
{
    bool operator<(const value &rhs) const {
        return m_x < rhs.m_x;}
    // bool operator==(const value &rhs) const;
    // bool operator!=(const value &rhs) const;
    // e la lista va avanti..
    private:
        // dati membri da comparare.
};

struct obj_type_3 { }
struct obj_type_4 { }
// e la lista va avanti..
```

- Per ogni oggetto comparabile, c'è bisogno di un rispettivo operatore di comparazione, il che è ridondante perché se abbiamo l'operatore <, possiamo overloadare gli altri operatori basandoci su quello.

- Quindi, l'operatore < è l'unico operatore che possiede l'informazione del tipo, altri operatori possono essere fatti indipendenti per questioni di riusabilità.

La soluzione:

L'implementazione è semplice: *separare le funzionalità dei tipi dipendenti e indipendenti e legare le funzionalità del tipo indipendente con la classe base usando la self-referencing template (template a se stesso).*

```
#include <iostream>

template <class derived>
struct compare {};
```

```
struct value : compare<value> {
    int m_x;
    value(int x) : m_x(x) {}
    bool operator<(const value &rhs) const {
        return m_x < rhs.m_x; }
```

```
};
```

```
template <class derived>
bool operator > (const compare<derived> &lhs,
                  const compare<derived> &rhs) {
    // static_assert(std::is_base_of_v<compare<derived>,
    //               derived>); // Compile time safety measures
    return (static_cast<const derived&>(rhs) <
            static_cast<const derived&>(lhs));}
```

```
}
```

```
/* Same goes with other operators
== :: returns !(lhs < rhs) and !(rhs < lhs)
!= :: returns !(lhs == rhs)
>= :: returns (rhs < lhs) or (rhs == lhs)
<= :: returns (lhs < rhs) or (rhs == lhs)
*/
```

```
// Ora non c'è più bisogno di scrivere gli operatori di
// comparazione per ogni classe.
// Scriviamo solo il tipo dipendente 'operator <' & usiamo il CRTP.
int main() {
    value v1{5}, v2{10};
```

```

    std::cout << std::boolalpha << "v1 > v2: " <<
        (v1 > v2) << '\n'; //Output: v1 > v2: false
    return 0;
}

```

CRTPO è ampiamente usata per il polimorfismo statico senza sostenere il costo del meccanismo di *virtual dispatch*. Consideriamo il seguente codice, senza l'uso di **virtual** e continuando ad ottenere le funzionalità del polimorfismo statico.

```

#include <iostream>

template<typename specific_animal>
class animal {
public:
    void who() { implementation().who(); }
private:
    specific_animal& implementation() { return *
        static_cast<specific_animal*>(this); }
};

class dog : public animal<dog> {
public:
    void who() { std::cout << "dog" << std::endl; }
};

class cat : public animal<cat> {
public:
    void who() { std::cout << "cat" << std::endl; }
};

template<typename specific_animal>
void who_am_i(animal<specific_animal> & animal) {
    animal.who();
}

int main()
{
    cat c;
    c.who(); //Output: cat
}

```

```
    return 0;
}
```

CRTPO può anche essere usato per l'ottimizzazione e permette anche la riusabilità del codice.

Con C++20 risolviamo anche con lo **spaceship operator** `<=>` (default comparisons)/ *three-way-comparison operator*.

3-way comparison operator | Spaceship Operator

Con il C++20 è stato aggiunto l'operatore *three-way-comparator* `<=>` chiamato lo **spaceship operator**.

Questo operatore determina per due oggetti A e B se $A < B$, $A = B$ o $A > B$.

Inoltre, una funzione *three-way-comparison* è una funzione che darà l'intera relazione in una query. Tradizionalmente, una funzione del genere è `strcmp()`, date due stringhe restituisce un intero dove:

- < 0 significa che la prima stringa è minore.
- $= 0$ significa che sono uguali.
- > 0 significa che la prima stringa è maggiore.

Può dare uno di questi tre risultati e quindi per questo è chiamata *three-way-comparison*.

Equality	Ordering
Primario: <code>==</code>	<code><=></code>
Secondario: <code>!=</code>	<code><, >, <=, >=</code>

$(A \text{ } <=> \text{ } B) \text{ } < \text{ } 0 \text{ } \text{ // è vera se } A < B.$
 $(A \text{ } <=> \text{ } B) \text{ } > \text{ } 0 \text{ } \text{ // è vera se } A > B.$
 $(A \text{ } <=> \text{ } B) \text{ } == \text{ } 0 \text{ } \text{ // è vera se } A \text{ e } B \text{ sono uguali.}$

Esempio:

```
#include <iostream>

int main()
{
    float a = -0.0;
    float b = 0.0;

    auto compare = a <=> b;
```

```

if (comapre < 0)
{
    std::cout << "-0.0 è minore di 0.0" <<
        std::endl;
} else if (compare == 0){
    std::cout << "-0.0 e 0.0 sono uguali." <<
        std::endl;
} else {
    std::cout << "-0.0 è maggiore di 0.0" <<
        std::endl;
}

//Output: -0.0 e 0.0 sono uguali.
return 0;
}

```

Virtual Constructor

Definizione: Il Costruttore Virtuale, o anche conosciuto come *Factory method/design-pattern*, ha lo scopo di creare una copia o un nuovo oggetto senza sapere di concreto il suo tipo.

Per fare questo sfruttiamo le funzioni *overloaded* con l'assegnamento polimorfico.

Il problema:

- C++ supporta la distruzione di oggetti polimorfici usando il **distruttore virtuale** della classe base, ma l'equivalente supporto per la creazione e la copiatura di oggetti manca visto che non supporta i **costruttori virtuali**, ma ci sono i **copy constructors**.
- Inoltre, non puoi creare un oggetto a meno che sai il suo tipo, perché il compilatore deve sapere l'ammontare di spazio da allocare. Per lo stesso motivo, la copiatura di un oggetto richiede anch'esso di sapere il suo tipo a compile-time.

```

#include <iostream>

class Animal {
public:

```

```

    virtual ~Animal() { std::cout << "~animal\n"
        ;
    };

class Dog : public Animal {
public:
    ~Dog() { std::cout << "~dog\n"; }
};

class Cat : public Animal {
public:
    ~Cat() { std::cout << "~cat\n"; }
};

void who_am_i(Animal *who) {
    // Come 'creare' l'oggetto dello stesso tipo, ovvero puntato da
    // who?
    // Come 'copiare' l'oggetto dello stesso tipo, ovvero puntato
    // da who?

    delete who; // si può cancellare gli oggetti puntati da
    // who.
}

```

La soluzione:

- La tecnica del **Costruttore Virtuale** permette la creazione polimorfica e la copiatura di oggetti delegando la creazione e la copiatura dell'oggetto alla classe derivata attraverso l'uso di funzioni virtuali.
- Il seguente codice non solo implementa il **costruttore virtuale**, ma anche il **virtual copy constructor**.

```

#include <iostream>
#include <memory>

class Animal {
public:
    virtual ~Animal() = default;
    virtual std::unique_ptr<Animal> create()
        = 0;

```

```

    virtual std::unique_ptr<Animal> clone() =
        0;
};

class Dog : public Animal {
public:
    std::unique_ptr<Animal> create() { return
        std::make_unique<Dog>(); }
    std::unique_ptr<Animal> clone() { return
        std::make_unique<Dog>(*this); }
};

class Cat : public Animal {
public:
    std::unique_ptr<Animal> create() { return
        std::make_unique<Cat>(); }
    std::unique_ptr<Animal> clone() { return
        std::make_unique<Cat>(*this); }
};

void who_am_i(Animal *who) {
    auto new_who = who->create(); // 'create' l'oggetto
                                   // dello stesso tipo, ovvero puntato da who ?

    auto duplicate_who = who->clone(); // 'copy'
                                       // l'oggetto dello stesso tipo, ovvero puntato da who ?

    delete who; // si può cancellare l'oggetto puntatore da
                // who.
}

```

Fornisce un'interfaccia generica per produrre/copiare una varietà di classi usando solo una classe.

SFINAE

Definizione: SFINAE (*Substitution Failure Is Not An Error*) ha lo scopo di filtrare le funzioni che non producono delle istanze template valide da un insieme di funzioni *overloaded*.

Otteniamo questo in maniera automatica dal compilatore oppure sfruttando `std::enable_if`.

Motivazioni:

- È una feature del linguaggio (non un idioma).
- Durante l'*overload* delle funzioni template, quando la sostituzione del tipo specificato o dedotto dei parametri del template fallisce, la specializzazione viene scartata invece che causare un errore di compilazione.
- Il fallimento della sostituzione (*Substitution Failure*) accade quando il tipo o l'espressione sono mal formati.

```
#include <iostream>

template<class T>
void func(T* t){ // Single overload set
    if constexpr(std::is_class_v<T>){ std::cout
        << "T è un tipo definito dall'utente\n"; }
    else { std::cout << "T è un tipo primitivo\n"
        ; }
}

int primitive_t = 6;
struct {char var = '4';} class_t;

int main()
{
    func(&class_t);
    func(&primitive_t);

    //Output: T è un tipo definito dall'utente
    //Output: T è un tipo primitivo
    return 0;
}
```

Immagina di volere creare due insiemi (basati sui tipi primitivi e sui tipi definiti dall'utente separatamente) di una funzione che ha la stessa *signature*? (tipo di ritorno, nome, parametri della funzione).

Soluzione:

```
#include <iostream>
```

```

template<class T, typename = std::enable_if_t<std::
         ::is_class_v<T>>>
void func(T* t){
    std::cout << "T è un tipo definito dall'utente.\n";
}

template<class T, std::enable_if_t<std::
         ::is_integral_v<T>, T> = 0>
void func(T* t){ // la signature della funzione non è modificata.
    std::cout << "T è un tipo primitivo.\n";
}

```

Il code snippet sopra è un esempio di come sfruttare **SFINAE** attraverso l'uso di `std::enable_if`, nella quale l'istanze template diventeranno uguali a `void func<(anonymous), void>((anonymous) * t)` e `void func(int *t)`

Assieme con `std::enable_if`, **SFINAE** è ampiamente usata nel *template metaprogramming*.

La libreria standard ha anche influenzato **SFINAE** nel *utilities <type_traits>*.

```

#include <iostream>

template<typename T>
class is_class_type {
    template<typename C> static char test(int C
        ::*) ;
    template<typename C> static double test(...);
public:
    enum { value = sizeof(is_class_type<T>::test<T>(0)) == sizeof(char) };
};

struct class_t {};

int main()
{
    std::cout << is_class_type<class_t>::value <<
        std::endl; //Output: 1
    std::cout << is_class_type<int>::value << std::
        endl; //Output: 0
}

```

```

    return 0;
}

```

Senza SFNAE avresti un errore dal compilatore, qualcosa del tipo: «*0 cannot be converted to member pointer for a non-class type int*» visto che sia l’overload di test differisce solo del tipo di ritorno.

Visto che int non è una classe non può avere membro puntatore di tipo *int int::**.

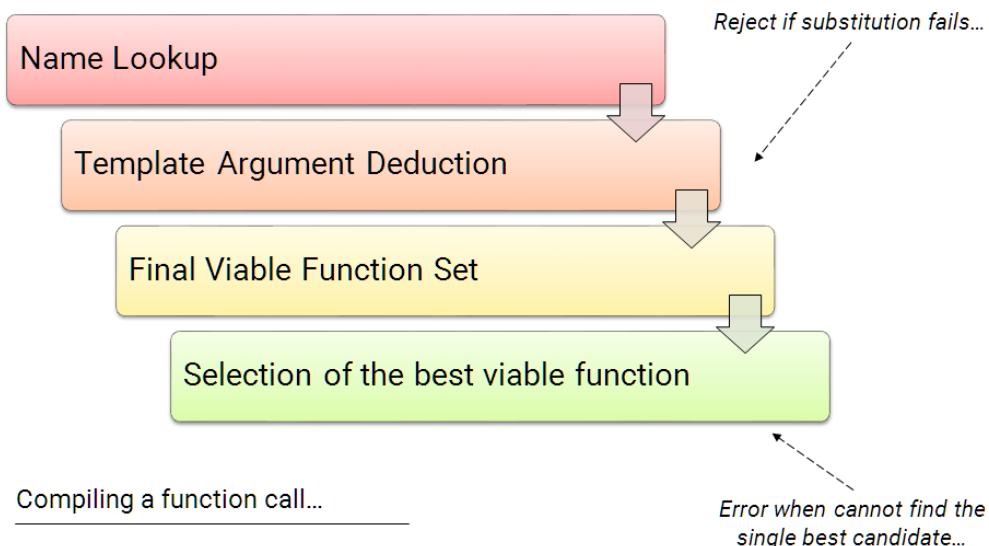


Figura 3.14: SFNAE

Proxy

Definizione: Il **Proxy**, o anche conosciuto come *operator[]* (*subscript*) *proxy*, *double/twice operator overloading* è un *design pattern* il cui intento è quello di ottenere funzionalità pratiche attraverso una classe *middleware*.

Quindi, il **Proxy** è una classe che fornisce un’interfaccia modificata ad un’altra classe ed è usata per implementare il **Proxy Patern**: nel quale un oggetto è un intermediatore per qualche altro oggetto.

Implementiamo questo attraverso una classe *temporanea/proxy*.

- È per via del *subscript operator* (*operator[]*), ma è possibile che anche il tipo/classe che si frappone nello scambio di dati sia proxy.
- Abbiamo già visto un esempio di questo nel **type-erasure** (classe *any::inner<>*).
- Vedremo un ulteriore esempio ora con l’*operator[]*

```

#include <iostream>

template <typename T = int>
struct arr2D{
    private:
        struct proxy_class{
            proxy_class(T *arr) : m_arr_ptr(arr) {}
            // operator[]
            T &operator[](uint32_t idx) { return
                m_arr_ptr[idx]; }

            private:
                T *m_arr_ptr;
        };

    T m_arr[10][10];

    public:
        arr2D::proxy_class operator[](uint32_t idx) {
            return arr2D::proxy_class(m_arr[idx]); }
};

int main()
{
    arr2D<> arr;
    arr[0][0] = 1;
    std::cout << arr[0][0]; //Output: 1
    return 0;
}

```

Per creare delle funzionalità pratiche come *double operator overloading*,
std::any, eccetera..

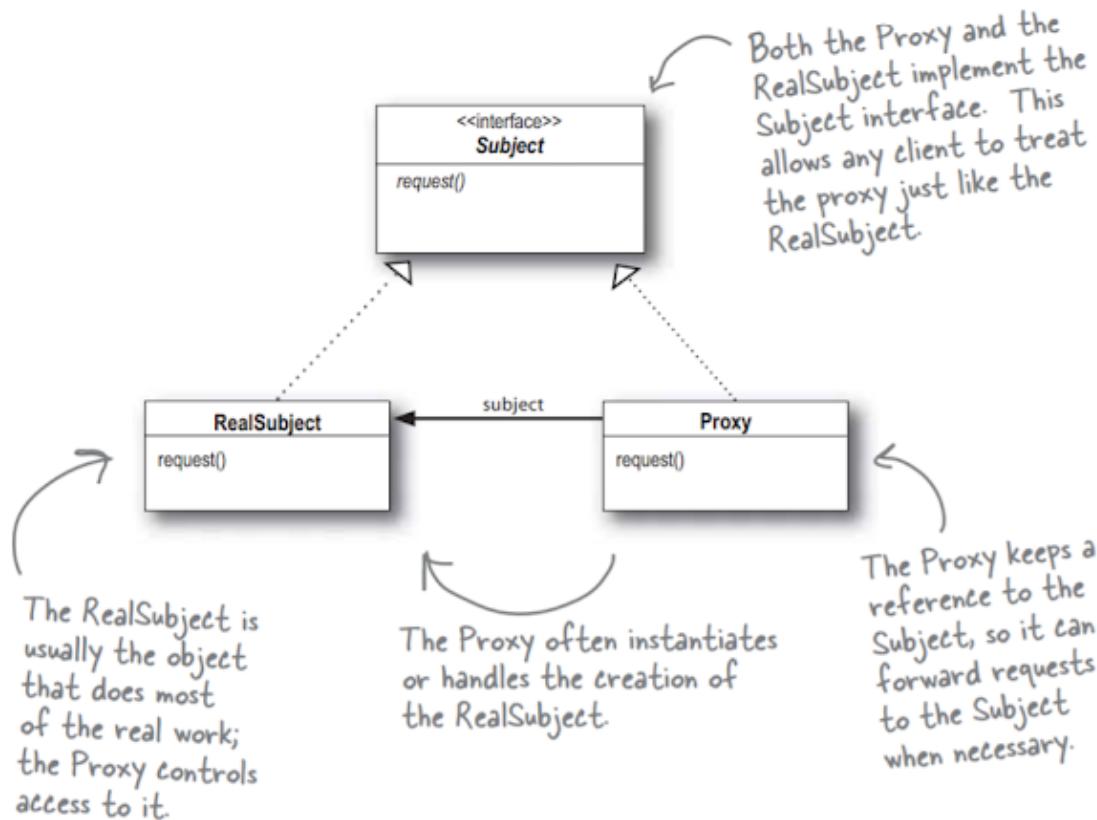


Figura 3.15: Proxy Pattern

Le gemme degli Algoritmi



Introduzione
