

# A Simple Guide to Modern C++

*Beginner*

*Intermediate*

*Advanced*

Luca Rengo

Novembre 2023 v1.1.1

# Indice

<b>Indice</b>	<b>ii</b>
<b>1 Basi del Linguaggio</b>	<b>1</b>
1.1 Introduzione . . . . .	1
1.2 Linguaggio . . . . .	1
1.3 Versioni . . . . .	2
1.4 Preprocessor . . . . .	5
1.5 Compilatore . . . . .	5
1.6 Linker . . . . .	5
1.7 Runtime Vs Compile time . . . . .	6
1.8 C vs C++ . . . . .	7
1.9 Keywords . . . . .	8
1.10 Tipi di Dati . . . . .	10
Cast . . . . .	11
Importanza di scrivere la f nei float . . . . .	12
1.11 Costanti . . . . .	14
const vs constexpr . . . . .	14
static . . . . .	14
Variabili statiche . . . . .	14
Membri statici delle classi . . . . .	14
static const . . . . .	15
static constexpr . . . . .	15
1.12 Arrays e Matrici . . . . .	15
Arrays . . . . .	15
Matrici . . . . .	16
1.13 Operatori Aritmetici . . . . .	17
1.14 Operatori Relazionali . . . . .	17
1.15 Operatori Bitwise . . . . .	18
1.16 Operatori di Assegnamento e Operatori Unari . . . . .	19
Operatori di Assegnamento . . . . .	19
Operatori Unari . . . . .	21

1.17 Operatori Logici . . . . .	22
1.18 Altri Operatori . . . . .	23
1.19 Condizione If . . . . .	25
If else if else . . . . .	25
Operatore Ternario . . . . .	25
if constexpr . . . . .	26
if vs if constexpr . . . . .	27
1.20 Switch . . . . .	27
Switch Cases Break . . . . .	27
Default Case . . . . .	28
1.21 Loops . . . . .	29
While . . . . .	29
Do-While . . . . .	29
Continue . . . . .	30
goto . . . . .	30
For . . . . .	31
Range-based for loop   Foreach . . . . .	31
Cicli annidati . . . . .	31
Cicli Infiniti . . . . .	32
1.22 Enumeratori . . . . .	32
1.23 Puntatori . . . . .	34
Aritmetica dei puntatori . . . . .	34
Puntatori a puntatori . . . . .	35
1.24 References . . . . .	36
References vs Puntatori . . . . .	38
NULL vs nullptr . . . . .	38
1.25 Stringhe . . . . .	40
Char . . . . .	40
C-string . . . . .	40
char* . . . . .	40
Tabella ASCII . . . . .	41
std::string . . . . .	41
char* vs std::string vs char[] . . . . .	42
Usare char* . . . . .	42
Usare std::string . . . . .	43
Casi in cui preferire char* ad std::string . . . . .	43
Usare char[] . . . . .	43
Escape characters . . . . .	43
'\n' vs "\n" . . . . .	44
1.26 Funzioni . . . . .	45

return . . . . .	45
void . . . . .	46
main . . . . .	46
Funzioni ricorsive . . . . .	47
Argomenti passati per valore . . . . .	48
Argomenti passati per referenza . . . . .	49
Funzioni che ritornano puntatori . . . . .	50
Parametri di default . . . . .	51
Funzione constexpr . . . . .	52
1.27 Variables Scope . . . . .	53
Variabili Locali . . . . .	53
Parametri formali . . . . .	54
Variabili Globali . . . . .	54
Perché evitare le variabili globali . . . . .	55
1.28 Header files . . . . .	56
Only Once Headers   Header Guards   pragma once   ifndef . . .	56
Cosa sono le librerie? . . . . .	57
Header files libreria Standard . . . . .	57
Librerie create dagli utenti . . . . .	58
Differenza tra .h vs .hpp . . . . .	58
1.29 Namespaces . . . . .	59
std:: vs using namespace std . . . . .	60
Mai mettere using namespace in un header file! . . . . .	62
1.30 Strutture . . . . .	63
typedef . . . . .	64
Funzioni nelle strutture . . . . .	65
Strutture nelle strutture . . . . .	65
Puntatore ad una struttura . . . . .	66
Array di Strutture . . . . .	67
Strutture come parametri e come ritorno . . . . .	67
Strutture in C vs in C++ . . . . .	70
Costruttori e Distruttori nelle Strutture . . . . .	70
1.31 Union . . . . .	71
structure vs union . . . . .	73
1.32 Classi . . . . .	74
Costruttori e Distruttori . . . . .	75
Costruttori . . . . .	75
Initialization List . . . . .	75
Distruttori . . . . .	76
Proprietà del distruttore . . . . .	76

Quando viene chiamato il distruttore? . . . . .	77
Access modifiers . . . . .	77
Incapsulamento . . . . .	77
scope resolution operator :: . . . . .	79
Getters & Setters . . . . .	80
Ereditarietà . . . . .	80
this pointer . . . . .	83
Multi-Ereditarietà . . . . .	84
Forward Declaration . . . . .	85
Chiamata a funzione statica e a membro . . . . .	87
Static nelle Classi . . . . .	87
Funzioni e la keyword const . . . . .	88
Class vs Struct . . . . .	90
<b>2 Concetti Intermedi</b> . . . . .	<b>91</b>
2.1 Introduzione . . . . .	91
2.2 STL   Standard Template Library . . . . .	91
Che cos'è #include <bits/stdc++.h>? . . . . .	92
2.3 Templates . . . . .	93
2.4 std::vector<> . . . . .	95
2.5 Iteratori . . . . .	97
Range-based for loop   Foreach . . . . .	100
Ottenere l'elemento per valore o per referenza a costante .	101
begin vs cbegin . . . . .	101
rend() . . . . .	101
2.6 Virtual . . . . .	102
Virtual functions . . . . .	102
Virtual Destructors . . . . .	104
Virtual Inheritance . . . . .	105
2.7 Polimorfismo . . . . .	108
2.8 Overloading . . . . .	109
Function Overloading . . . . .	109
Operator Overloading . . . . .	110
Overloading vs Overriding . . . . .	111
2.9 Tipi di Casts . . . . .	113
static_cast<> . . . . .	113
const_cast<> . . . . .	113
dynamic_cast<> . . . . .	114
reinterpret_cast<> . . . . .	115
C-style & function-style cast o Regular Cast . . . . .	116
Ricapitolando . . . . .	116

2.10 Lambdas . . . . .	117
mutable . . . . .	119
2.11 Memoria dinamica . . . . .	120
Memoria Dinamica in C . . . . .	120
new e delete . . . . .	121
new . . . . .	121
array normali vs array con la new . . . . .	121
delete . . . . .	121
Evitare di usare new . . . . .	122
2.12 RAII   Resource Acquisition is initialization . . . . .	122
2.13 Constructor types   Rules of . . . . .	123
Rule of Zero . . . . .	123
Copy Constructor . . . . .	123
Copy Assign . . . . .	124
=default   Defaulted Functions . . . . .	125
=delete   Deleted Functions . . . . .	126
Preventing Object Copy . . . . .	127
Copy Constructor vs Copy Assignment Operator . . . . .	128
Rule of Three . . . . .	129
Move Constructor . . . . .	129
lvalues references & rvalues references . . . . .	131
xvalue, glvalue, prvalue . . . . .	133
Move Assignment Operator . . . . .	133
Rule of Five . . . . .	134
2.14 Move Semantics . . . . .	135
Fallbacks of move semantics . . . . .	136
syntax vs semantics . . . . .	137
Ricapitolando le Move Semantics . . . . .	138
2.15 Classi Astratte . . . . .	140
Pure Virtual Functions . . . . .	140
Pure Virtual Destructor . . . . .	140
Abstract Class . . . . .	141
Abstract class vs Interface . . . . .	143
2.16 Eccezioni . . . . .	144
try catch throw . . . . .	144
try . . . . .	144
catch . . . . .	144
std::throw exception . . . . .	145
Errori a compile time ed errori a runtime . . . . .	145
noexcept . . . . .	146

Assertions . . . . .	148
2.17 Operazioni di Input/Output . . . . .	150
Input-Output stream . . . . .	150
std::endl vs newline . . . . .	151
Meglio '\n' oppure "\n" . . . . .	152
Manipolazione degli stream . . . . .	152
Operazioni su file . . . . .	155
2.18 std::chrono . . . . .	158
Duration . . . . .	158
Clock . . . . .	159
Time point . . . . .	159
Calcolare il tempo di esecuzione di un blocco di codice . . . . .	159
2.19 Generatori di numeri pseudo-casuali . . . . .	161
Numeri casuali come in C . . . . .	161
Cosa vuol dire e perché pseudo-random? . . . . .	163
I vari tipi di generatori . . . . .	163
2.20 Map   Set   Pair   Hash Table . . . . .	173
Map . . . . .	173
unordered_map . . . . .	174
unordered_map function pointer . . . . .	175
multimap . . . . .	177
Set . . . . .	178
multiset . . . . .	179
Pair . . . . .	180
Tuple . . . . .	182
Pair vs Tuple . . . . .	185
Hash Table . . . . .	186
2.21 Type Traits . . . . .	187
decltype . . . . .	187
2.22 RTTI   Run-Time Type Information . . . . .	189
Runtime Casts . . . . .	189
2.23 Typeinfo . . . . .	190
typeid . . . . .	190
typeinfo . . . . .	192
2.24 Ellissi . . . . .	193
2.25 Inline Functions . . . . .	195
Funzioni Inline vs Funzioni constexpr . . . . .	198
2.26 Copy Elision . . . . .	198
2.27 Conversion Constructor & Explicit keyword . . . . .	199
Explicit Keyword . . . . .	199

Conversion Constructor . . . . .	200
2.28 Private Destructors . . . . .	201
2.29 Command Line Arguments . . . . .	203
2.30 Uniform Real Distribution . . . . .	205
<b>3 Concetti Avanzati . . . . .</b>	<b>207</b>
3.1 Introduzione . . . . .	207
3.2 Friend Keyword . . . . .	208
Friend Class . . . . .	208
Friend Function . . . . .	209
3.3 Smart Pointers . . . . .	211
Differenze con i puntatori normali . . . . .	211
unique pointers . . . . .	213
Differenza tra std::make_unique vs std::unique_ptr . . . . .	215
Ownership   move . . . . .	216
Operazioni sugli unique_ptr . . . . .	217
Passare uno unique_ptr ad una funzione . . . . .	218
Restituire un unique_ptr . . . . .	219
Membri delle classi: unique pointer vs raw pointer vs reference . . . . .	219
share pointers . . . . .	220
Differenza tra std::shared_ptr vs std::make_shared . . . . .	221
Operazioni sui shared pointers . . . . .	222
Distruzione degli shared pointers . . . . .	223
Passare gli shared pointers ad una funzione . . . . .	225
Restituire gli shared pointers . . . . .	226
static_pointer_cast . . . . .	227
enable_shared_from_this . . . . .	229
Performance degli shared pointers . . . . .	230
Shared pointers, unique pointers or raw pointers . . . . .	230
weak pointers . . . . .	231
Problema della dipendenza ciclica . . . . .	231
Quando usare i weak pointers? . . . . .	232
Operazioni sugli weak pointers . . . . .	232
3.4 Copy-And-Swap Idiom . . . . .	234
3.5 Tecniche per il debugging . . . . .	237
3.6 Puntatore a funzione . . . . .	238
3.7 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 . . . . .	249
3-way comparison operator   Spaceship Operator . . . . .	253
Virtual Constructor . . . . .	254
SFINAE . . . . .	256
Proxy . . . . .	259
<b>4 Le gemme degli Algoritmi</b>	<b>263</b>
4.1 Introduzione . . . . .	263
4.2 Operazioni su sequenze non-modificabili . . . . .	265
Condizioni . . . . .	265
Ricerca . . . . .	266
find vs search . . . . .	268
Contatori . . . . .	268
Altre operazioni . . . . .	268
4.3 Operazioni su sequenze modificabili . . . . .	271
Copiare sequenze   Copy . . . . .	271
Muovere   Move . . . . .	273
Scambiare   Swap . . . . .	274
Trasformare   Transform . . . . .	275
Rimpiazzare   Replace . . . . .	276
Riempire   Fill . . . . .	277
Generatori   Generate . . . . .	278
Rimozione   Remove . . . . .	279
Unico   Unique . . . . .	280
Invertire   Reverse . . . . .	281
Ruotare   Rotate . . . . .	282
Spostare   Shift . . . . .	283
Mischiare   Shuffle . . . . .	284
4.4 Operazioni su Partizioni . . . . .	285
4.5 Operazioni di Ordinamento . . . . .	289
4.6 Operazioni di Ricerca Binaria . . . . .	293
4.7 Operazioni di Fusione   Merge . . . . .	296
4.8 Operazioni sugli Insiemi . . . . .	299
4.9 Operazioni su Heap . . . . .	302
4.10 Operazioni di Min/Max . . . . .	307
4.11 Operazioni di Comparazione . . . . .	310

4.12 Operazioni su Permutazioni . . . . .	312
4.13 Operazioni Numeriche . . . . .	314
Differenza tra std::accumulate vs std::reduce . . . . .	317
4.14 Operazioni su memoria non inizializzata . . . . .	318
4.15 Optionals . . . . .	323
4.16 Execution Policies . . . . .	326
4.17 C++20 . . . . .	333
Ranges . . . . .	333
Tipi di Ranges . . . . .	333
Operazioni su sequenze non modificabili . . . . .	334
Operazioni su sequenze modificabili . . . . .	335
Operazioni su Partizioni . . . . .	337
Operazioni di Ordinamento . . . . .	339
Operazioni su ricerca binaria . . . . .	340
Operazioni di Fusione   Merge . . . . .	341
Operazioni sugli Insiemi . . . . .	342
Operazioni su Heap . . . . .	345
Operazioni su Min/Max . . . . .	347
Operazioni di Comparazione . . . . .	348
Operazioni su Permutazioni . . . . .	349
Operazioni numeriche . . . . .	350
Operazioni su memoria non inizializzata . . . . .	351
Tipi di Ritorno . . . . .	352
Concepts . . . . .	354
concept . . . . .	354
requires . . . . .	356
Concepts fixano, sposano e rimpiazzano SFINAE . . . . .	359
Modules . . . . .	360
Modules Partitions . . . . .	362
Module Implementation . . . . .	364
Coroutines . . . . .	366
A cosa servono le coroutines? . . . . .	367
Terminologia delle Coroutine . . . . .	367
Esempi di coroutines . . . . .	369
Stackfull & Stackless Coroutines . . . . .	375
Differenza tra threads vs coroutines . . . . .	375
<b>5 Linee guida del Linguaggio</b>	<b>377</b>
5.1 Introduzione . . . . .	377
5.2 Convenzioni del linguaggio . . . . .	378
NL   Naming and Layout suggestions . . . . .	378

NL.1: Non dire nei commenti quello che può essere semplicemente affermato nel codice . . . . .	378
NL.2: Indica l'intento nei commenti . . . . .	378
NL.3: Mantieni i commenti nitidi . . . . .	378
NL.4: Mantieni un'indentazione costante . . . . .	378
NL.5: Evita la codifica delle informazioni sui tipi dei nomi . . . . .	379
NL.7: Fai sì che la lunghezza di un nome sia proporzionale alla lunghezza del suo scope (raggio) . . . . .	379
NL.8: Usa uno stile dei nomi in modo costante . . . . .	380
NL.9: Usa ALL_CAPS (tutte maiuscole) solo per i nomi delle macro . . . . .	380
NL.10: Preferire nomi con lo stile_underscore (con i trattini bassi) . . . . .	381
NL.11: Rendi i letterali leggibili . . . . .	381
NL.15: Usa gli spazi con moderazione . . . . .	382
NL.16: Usa una convenzionale ordine di dichiarazione della classe . . . . .	382
NL.17: Usa il layout di K&R . . . . .	383
NL.18: Usa il layout dei dichiaratori del C . . . . .	384
NL.19: Evita nomi che possono essere facilmente fraintesi . . . . .	384
NL.20: Non piazzare due dichiarazioni sulla stessa riga . . . . .	385
NL.21: Dichiara un nome (solo) per dichiarazione . . . . .	385
NL.25: Non usare void come argomento . . . . .	385
NL.26: Usa la convenzionale notazione const . . . . .	385
Organizzazione . . . . .	387
Sezioni Maggiori . . . . .	387
In   Introduction . . . . .	388
P   Philosophy . . . . .	388
P.1: Esprimi le idee direttamente nel codice . . . . .	388
P.2: Scrivi in ISO Standard C++ . . . . .	388
P.3: Esprimi l'intento . . . . .	388
P.4: Idealmente, un programma dovrebbe essere staticamente type safe . . . . .	388
P.5: Preferire i controlli a compile-time rispetto che a run-time . . . . .	389
P.6: Quello che non può essere controllato a compile time dovrebbe essere controllato a run time . . . . .	389
P.7: Prendere gli errori a run-time in anticipo . . . . .	389
P.8: Non <i>leakere</i> nessuna risorsa . . . . .	390

P.9: Non perdere nè tempo nè spazio . . . . .	390
P.10: Preferire i dati immutabili a quelli mutabili .	390
P.11: Incapsula i costruttori disordinati, al posto di diffonderli nel codice . . . . .	391
P.12: Usa gli strumenti di supporto come è appropriato	391
P.13: Usa le librerie supportate in modo appropriato	391
<b>I   Interfaces . . . . .</b>	<b>392</b>
I.1: Fai delle interfacce esplicite . . . . .	392
I.2: Evita variabili globali non costanti . . . . .	392
I.3: Evita i singlettons . . . . .	392
I.4: Rendi le interfacce precise e fortemente typed .	392
I.5: Precondizioni di stato (se ce ne sono) . . . . .	392
I.6: Preferisci <b>Expects()</b> per esprimere le precondizioni . . . . .	392
I.7: Postcondizioni di stato . . . . .	392
I.8: Preferire <b>Ensures()</b> per esprimere postcondizioni	393
I.9: Se un'interfaccia è un template, allora documenta i suoi parametri usando i concepts .	393
I.10: Usa le eccezioni per segnalare un fallimento nel performare una attività necessaria .	393
I.11: Non trasferire mai l'ownership (la proprietà) di un raw pointer ( $T^*$ ) o di una referenza ( $T\&$ ) . . . . .	393
I.12: Dichiara un puntatore che <i>non</i> deve essere <i>null</i> come <i>not_null</i> . . . . .	393
I.13: Non passare un array come un singolo puntatore	393
I.23: Mantieni il numero di argomenti in una funzione basso . . . . .	393
<b>F   Functions . . . . .</b>	<b>394</b>
F.21: Per restituire molteplici "out" values, preferire il ritorno di una struttura o di una tupla	394
F.51: Se c'è possibilità di scelta, preferisci gli argomenti di default rispetto all'overloading	394
<b>C   Classes and class hierarchies . . . . .</b>	<b>395</b>
C.45: Non definire un costruttore di default solo per inizializzare dei dati membri; usa gli inizializzatori della classe piuttosto . . .	395
C.47: Definisci e inizializza le variabili membro nell'ordine della loro dichiarazione . . .	395

C.48: Preferisci gli inizializzatori nella classe al posto degli inizializzatori ai membri nei costruttori per gli inizializzatori costanti	395
Enum   Enumerations . . . . .	395
Enum.3: Preferire le classi enum al posto degli enum "scoperti"	395
R   Resource Management . . . . .	396
R.10: Evita la malloc() e la free()	396
R.11: Evita di chiamare <b>new</b> e <b>delete</b> in modo esplicito	396
ES   Expressions and Statements . . . . .	397
ES.10: Dichiara solo un nome per dichiarazione . .	397
ES.46: Evita le conversione aritmetiche che perdono dati (narrowing, truncating)	397
ES.50: Non togliere il const col cast . . . . .	397
Per   Performance . . . . .	399
Per.2: Non ottimizzare prematuramente . . . . .	399
Per.3: Non ottimizzare qualcosa che non è critico a livello di performance . . . . .	399
Per.4: Non presumere che del codice più complicato sia necessariamente più veloce . . . . .	399
Per.5: Non supporre che il codice a basso livello sia necessariamente più veloce di quello ad alto livello . . . . .	400
Per.7: Progetta per permettere l'ottimizzazione . .	400
Per.11: Sposta la computazione dal run time al compile time . . . . .	401
Per.19: Accedi alla memoria in modo prevedibile .	401
CP   Concurrency and Parallelism . . . . .	403
CP.2: Evita le <i>data races</i> . . . . .	403
CP.3: Minimizza la condivisione esplicita di dati scrivibili . . . . .	403
CP.4: Pensa in termini di tasks, al posto che in threads	403
CP.8: Non provate ad usare <i>volatile</i> per la sincronizzazione . . . . .	404
E   Error Handling . . . . .	405
E.1: Sviluppa una strategia per occuparti degli errori il prima possibile in un design . . . . .	405
E.6: Usa RAII per prevenire i <i>leaks</i> . . . . .	405
E.12: Usa <i>noexcept</i> quando esci da una funzione perché una <i>throw</i> è impossibile o inaccettabile	406

Con   Constants and immutability . . . . .	407
Con.1: Di default, poni gli oggetti immutabili . . . . .	407
Con.5: Usa <i>constexpr</i> per i valori che possono essere calcolati a tempo di compilazione (compile-time) . . . . .	407
T   Templates and generic programming . . . . .	408
T.1: Usa i <i>templates</i> per aumentare il livello di astrazione del codice . . . . .	408
T.5: Combina le tecniche OO (Object-Oriented) e quelle generiche per amplificare le loro forze, non i loro costi . . . . .	409
T.120: Usa la metaprogrammazione solo quando ti serve veramente . . . . .	411
CPL   C-style programming . . . . .	412
CPL.1: Preferire il C++ al C . . . . .	412
SF   Source Files . . . . .	413
SF.1: Usa il suffisso .cpp per i sorgenti e .h per le interfacce se il tuo progetto non richiede già un'altra convenzione . . . . .	413
SF.4: Includi i files .h nel file prima di altre dichiarazione . . . . .	414
SF.5: Un file .cpp deve includere il file(s) .h che definiscono la sua interfaccia . . . . .	414
SF.6: Usa le direttive namespaces per transizione, per librerie fondamentali (come la <i>std</i> ) o in uno scope locale (soltanto) . . . . .	415
SF.7: Non scrivere <i>using namespace</i> in uno scope globale in un header file . . . . .	416
SF.8: Usa le #include guards per tutti i files .h . . . . .	416
SF.9: Evita le dipendenze cicliche tra i file sorgenti . . . . .	417
SF.11: Gli header files dovrebbero essere indipendenti (self-contained) . . . . .	417
SF.12: Preferire gli #include con le virgolette per i files relativi e i simboli < > per il resto . . . . .	418
SL   The Standard Library . . . . .	420
SL.1: Usa le librerie quando possibile . . . . .	420
SL.2: Preferire la libreria standard alle altre librerie . . . . .	420
SL.3: Non aggiungere entità non-standard al namespace std . . . . .	420

SL.4: Usa la libreria standard in modo type-safe (sicuro per i tipi) . . . . .	420
SL.con.1: Preferire l'utilizzo di STL <i>array</i> o <i>vector</i> al posto degli array del C . . . . .	421
SL.str.1: Usare <i>std::string</i> per possedere le sequenze di caratteri . . . . .	421
SL.str.2: Usa <i>std::string_view</i> o <i>gsl::span&lt;char&gt;</i> per riferirti a sequenze di caratteri . . . . .	422
SL.str.4: Usa <i>char*</i> per riferirti ad un singolo carattere	422
SL.str.12: Usa il suffisso <i>s</i> per le stringhe letterali (string literals) intesi come stringhe della libreria standard . . . . .	422
SL.io.3: Preferire <i>iostream</i> per l'I/O (Input/Output)	423
SL.io.50: Evita l' <i>endl</i> . . . . .	423
SL.regex: Regex . . . . .	424
SL.chrono: Time . . . . .	424
SL.C.1: Non usare <i>setjmp longjmp</i> . . . . .	424
Conclusioni . . . . .	425



# Basi del Linguaggio

---

## *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, come sua evoluzione.

## *Versioni*

---

Il C++ è un linguaggio che esiste da tanto e quindi nel tempo ha ricevuto diversi aggiornamenti, di seguito una lista delle versioni del linguaggio:

Il numero affianco a C++ è l'anno in cui è uscita la versione.

- **C++98/03**

- *Nuove features* : RTTI, covariant return types, cast operators, mutable, bool, declarations in conditions, template instantiations, member templates, export
- *Aggiunte alle librerie* : locales, bitset, valarray, auto\_ptr, templated string, I/O streams, e complex numbers.
- *Basate sul STL* : containers, algorithms, iterators, function objects

- **C++11**

- Grosso numero di cambiamenti alle pratiche standard già esistenti e miglioramenti all'astrazione disponibile per i programmatorei.
- Numeri in virgola mobile dal IEEE 754: std::decimal::decimal32, std::decimal::decimal64, eccetera.
- Fondata la *The Standard C++ Foundation*.

- **C++14**

- *Nuove features* : variable templates, generic lambdas, lambda init-capture, new/delete elision, relaxed restrictions on constexpr functions, binary literals, digit separators, return type deduction for functions, aggregate classes with default non-static member initializers.
- *Nuove features delle librerie* : std::make\_unique std::shared\_timed\_mutex and std::shared\_lock, std::integer\_sequence, std::exchange, std::quoted, e molti altri piccoli miglioramenti alle librerie già presenti, come: two-range overloads for some algorithms, type alias versions of type traits, user-defined literals for basic\_string, duration and complex, eccetera.

- Altre estensioni alle librerie.
- C++17
  - *Nuove features* : fold-expressions, class template argument deduction, non-type template parameters declared with auto, compile-time if constexpr, inline variables, structured bindings, initializers for if and switch, u8 character literal, simplified nested namespaces, using-declaration declaring multiple names, made noexcept part of type system, new order of evaluation rules, guaranteed copy elision, temporary materialization, lambda capture of \*this, constexpr lambda, gli attributi dei namespaces non devono essere ripetuti, nuovi attributi: [[fallthrough]], [[maybe\_unused]], [[nodiscard]], \_\_has\_include
  - *Nuovi files di intestazione* : <any>, <charconv>, <execution>, <filesystem>, <memory\_resource>, <optional>, <string\_view>, <variant>
  - Ulteriori funzionalità nelle librerie, estensioni, rimozioni e deprecazione di altre funzionalità.
- C++20
  - *Nuove features* : Feature test macros, 3-way comparison operator <=> and operator==() = default, designated initializers, init-statements and initializers in range-for, char8\_t, Nuovi attributi: [[no\_unique\_address]], [[likely]], [[unlikely]], pack-expansions in lambda init-captures, removed the requirement to use typename to disambiguate types in many contexts, consteval, constinit, further relaxed constexpr, signed integers are 2's complement, aggregate initialization using parentheses, Coroutines, Modules, Constraints and concepts, Abbreviated function templates, DR: array new can deduce array size
  - *Nuovi headers* : <concepts>, <coroutine>, <compare>, <version>, <source\_location>, <format>, <span>, <ranges>, <bit>, <numbers>, <sync-stream>, Per il supporto della libreria dei threads: <stop\_token>, <semaphore>, <latch>, <barrier>
  - Ulteriori funzionalità alle librerie.
- C++23

Per determinare quale standard del linguaggio il compilatore sta utilizzando usufruiamo della macro: \_\_cplusplus e la stampiamo a schermata.

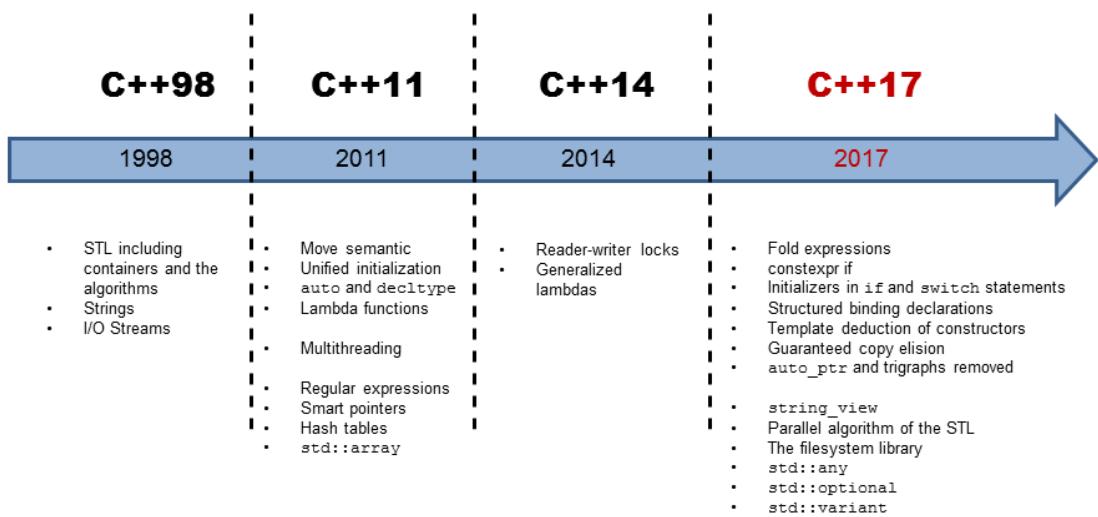


Figura 1.1: Versioni

---

## *Preprocessor*

---

Il **preprocessore** è composto da delle direttive che devono essere preprocessate dal compilatore prima dell'effettiva compilazione.

Per esempio di includere una libreria standard del linguaggio.

Le direttive del preprocessore iniziano con il carattere #.

```
// 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 codice per essere eseguito (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 è 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	Le variabili di referenza

C	C++
non sono supportate.	sono supportate.
Le funzioni virtual e friends non sono supportate.	Le funzioni virtual e 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.

---

## *Keywords*

---

Il C++ presenta oltre 60+ **keywords**, ovvero *parole chiavi* del linguaggio.

Le **keywords** sono delle parole riservate che hanno speciali funzionalità associate ad esse e sono già definite dal compilatore. Non si possono usare queste *parole chiavi* come identificatori, come nomi delle variabili, ecc.

Di seguito, un'immagine di parte delle **keywords** del linguaggio:

Table 4 — Keywords

alignas	continue	friend	register	true
alignof	decltype	goto	reinterpret_cast	try
asm	default	if	return	typedef
auto	delete	inline	short	typeid
bool	do	int	signed	typename
break	double	long	sizeof	union
case	dynamic_cast	mutable	static	unsigned
catch	else	namespace	static_assert	using
char	enum	new	static_cast	virtual
char16_t	explicit	noexcept	struct	void
char32_t	export	nullptr	switch	volatile
class	extern	operator	template	wchar_t
const	false	private	this	while
constexpr	float	protected	thread_local	
const_cast	for	public	throw	

---

## ***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 1.2: Tabelle dei tipi

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

<code>size_t</code>	<b>unsigned int</b> (se il compilatore è di 32 bit)
	<b>unsigned long long</b> (se il compilatore è da 64 bit)
<code>uint8_t</code>	<b>unsigned char</b>
<code>uint16_t</code>	<b>unsigned int</b>
<code>uint32_t</code>	<b>unsigned long</b>

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

<code>string</code>	<b>una stringa di caratteri</b> (includere <code>&lt;string&gt;</code> )
<code>wchar_t</code>	<b>wide char</b> (per caratteri più grandi di 255)

Altri tipi di dati:

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

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

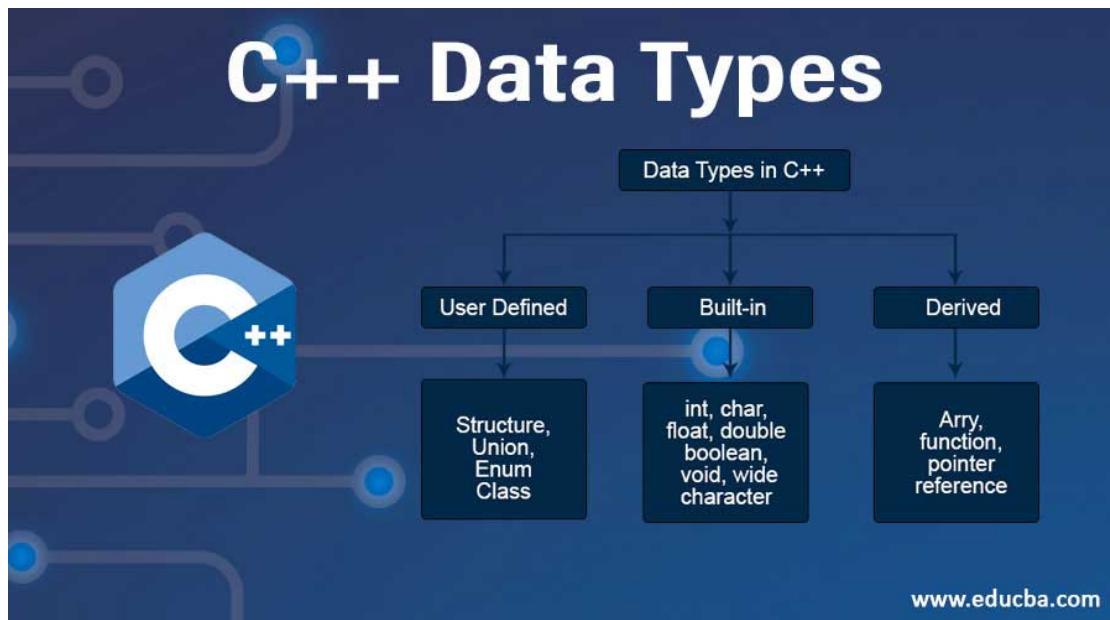


Figura 1.3: Tipi di dati

## 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 = 22.2f;
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 o anche chiamato C-cast (ovvero un cast come nel C), ci sono altre forme di cast un po' più complesse che vedremo in un altro capitolo a pag.112.

## *Importanza di scrivere la f nei float*

Per i valori **float** (numeri in virgola mobile in singola precisione) sarà necessario oltre al numero inserire una **f** dopo esso per indicare che questo è un numero in virgola mobile di tipo **float** (4 bytes, ovvero 32 bit).

Laddove noi non inseriscono la **f** nel numero con la virgola, quello verrebbe considerato un numero in virgola mobile in doppia precisione di tipo **double** (che occupa 8 bytes, ovvero 64 bit).

```
float x;
```

```
float y = x / 2.0; // Se non si mette la f a quel 2.0, allora la  
variabile di tipo float y verrà promossa a double, perché 2.0 senza la f è  
un double.
```

```
// Quindi dovremmo scrivere per restare con i float: 'float y = x / 2.0f;'
```

---

## *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*

<code>const</code>	<code>constexpr</code>
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 1.4: 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 degli 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.

---

## *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.

- **$\neq$**  : per controllare se due operandi non sono equivalenti.
- **$>$**  : per controllare se un operando è maggiore di un altro
- **$\geq$**  : per controllare se un operando è maggiore o uguale all'altro.
- **$<$**  : per controllare se un operando è minore di un altro.
- **$\leq$**  : 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 singoli 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.
- **$\wedge$  (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.
- **$\sim$  (bitwise NOT)** : prende un numero ed inverte tutti i bits.

Truth Table of Bitwise Operators					
In C++ boolean <i>true</i> is 1 and <i>false</i> is 0					
a	b	a & b	a   b	a ^ b	~ a
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Figura 1.5: Tabella della verità delle operazioni bitwise.

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

a & b; // Output: 00000001
a | b; // Output: 00001101
a ^ b; // Output: 00001100
~a; // Output: 11111010
b << 1; // Output: 00010010
b >> 1; // Output: 00000100
```

## *Operatori di Assegnamento e Operatori Unari*

### *Operatori di Assegnamento*

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

- = : Operatore di assegnamento di un valore ad una variabile.

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

```

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

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

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

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

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

```

## *Operatori Unari*

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

- **-** : nega il valore dell'operando.
- **++nome\_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

Truth Table of Logical Operators				
In C++ boolean <i>true</i> is 1 and <i>false</i> is 0				
a	b	a && b	a    b	! a
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Figura 1.6: Tabella della verità degli operatori logici.

```
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 nell'if statement 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.

```
#include <iostream>

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 nell'if.

### *if constexpr*

È possibile implementare un if statico sotto forma di **if constexpr** che valuta la condizione a *tempo di compilazione* e solo un blocco verrà compilato.

```

#include <iostream>

int main()
{
    const int a = 2, b = 3;
    if constexpr(a > b)
    {
        std::cout << a << " è maggiore di " << b <<
            std::endl;
    } else {
        std::cout << a << " è minore di " << b << std
            ::endl;
    }

    //Output: 2 è minore di 3
    return 0;
}

```

Un altro modo per fare un *if statico* è quello di usare `std::enable_if`.  
Questa espressione sarà utile nel *template metaprogramming*.

## *if vs if constexpr*

<b>if</b>	<b>if constexpr</b>
<i>Runtime</i>	<i>Compile Time</i>
Tutti i rami (branches) devono essere corretti sintatticamente	Solo il ramo (branch) selezionato deve essere corretto
No; tutti i rami vengono valutati a runtime	Sì; i rami non selezionati vengono eliminati
Condizioni a runtime, ad esempio, input utente	Condizioni a compile-time, ad esempio, template
Tutti i rami vengono istanziati	Solo il ramo selezionato viene istanziato
Possibili errori a runtime	Errori di compilazione se i rami non selezionati contengono errori
Possibile overhead a runtime	Potenziale per codice più efficiente
Meno chiara a causa dell'aspetto runtime	Spesso più chiara per condizioni di compile-time

## *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.

```
#include <iostream>

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.

```
#include <iostream>

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.

```
#include <iostream>

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.

```
#include <iostream>

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.

```
#include <iostream>

int a = 10;

LOOP: do {
    if( a == 15) {
```

```

    // skip the iteration.
    a = a + 1;
    goto LOOP;
}
std::cout << "value of a: " << a << std::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
}

```

## *Range-based for loop | Foreach*

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

Vengono trattati nel capitolo *Concetti Intermedi* a pag.100.

## *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.

```
#include <iostream>

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

## *Enumeratori*

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

```
#include <iostream>

enum Days {Lunedì, Martedì, Mercoledì, Giovedì,
           Venerdì, Sabato, Domenica};
Days day = Venerdì;

if (day == Venerdì){
    std::cout << "Oggi è venerdì!" << std::endl;
}

enum Year {
    GENNAIO = 1,
    FEBBRAIO,
    MARZO,
    APRILE,
    MAGGIO,
    GIUGNO,
    LUGLIO,
    AGOSTO,
```

```
SETTEMBRE,  
OTTOBRE,  
NOVEMBRE,  
DICEMBRE  
};  
  
Year mese = FEBBRAIO;  
std::cout << "Siamo nel " << mese << "o mese dell'anno  
" << std::endl;  
  
enum Colors {  
    ROSSO,  
    BLU,  
    VERDE,  
    GIALLO,  
    ARANCIONE,  
    GRIGIO,  
    VIOLA,  
    ROSA,  
    NERO,  
    BIANCO  
};  
  
Colors colore = Colors.ARANCIONE;  
std::cout << "Colore di indice: " << colore << std::endl;  
  
// Per poter vedere il nome dell'enum e non il suo valore, bisogna  
utilizzare una mappa o uno switch o altro.
```

---

## *Puntatori*

---

**Definizione:** Un puntatore è una variabile che contiene l'indirizzo di memoria di un'altra variabile. Si può dire che questa variabile punta ad un'altra.

Per creare un puntatore usiamo l'operatore `*` che è lo stesso usato anche per la moltiplicazione.

Usiamo l'operatore `&` (indirizzo) per ottenere l'indirizzo di una variabile.

Per ottenere il valore della variabile a cui il puntatore punta usiamo l'operatore `*` (dereferenza).

```
#include <iostream>

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 a un puntatore.
- Sottrazione di un intero a un puntatore.
- Sottrazione di due puntatori dello stesso tipo.

```
#include <iostream>

// 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.

```
#include <iostream>

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.

```
#include <iostream>

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 the following details:

- Code View:** Shows the C++ code for the `ComposeFrame()` method. A red circle highlights the assignment statement `r = v;`.
- Output View:** Displays the command-line output: `'Engine.exe' (Win32): Loaded 'C:\Windows\S...'.`
- Autos View:** Shows the current values of variables:
 

Name	Type	Value
r	int &	69
this	Game *	0x00def7a4 (wnd=(kbd=(autorepeatEnabled=false keystates=( size=256 ) keybuffer...))
v	int	69

Figura 1.7: Reference

The screenshot shows a debugger interface with the following details:

- Code View:** Shows the C++ code for the `ComposeFrame()` method. A red circle highlights the assignment statement `r = v;`.
- Output View:** Displays the command-line output: `'Engine.exe' (Win32): Loaded 'C:\Windows\S...'.`
- Autos View:** Shows the current values of variables:
 

Name	Type	Value
r	int &	489
this	Game *	0x00def7a4 (wnd=(kbd=(autorepeatEnabled=false keystates=( size=256 ) keybuffer...))
v	int	489

Figura 1.8: Reference

## *References vs Puntatori*

Reference	Pointers
Riferiscono a una variabile con un altro nome.	Memorizzano un indirizzo di una variabile.
Non possono avere un valore <code>NULL</code> .	Possono avere un valore <code>NULL</code> .
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.

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

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

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

// Esempio 2
#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-stringa (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	64	@	96	'
01	SOH (Start of Header)		33	!	65	A	97	a
02	STX (Start of Text)		34	"	66	B	98	b
03	ETX (End of Text)		35	#	67	C	99	c
04	EOT (End of Trans.)		36	\$	68	D	100	d
05	ENQ (Enquiry)		37	%	69	E	101	e
06	ACK (Acknowledgement)		38	&	70	F	102	f
07	BEL (Bell)		39	'	71	G	103	g
08	BS (Backspace)		40	(	72	H	104	h
09	HT (Horizontal Tab)		41	)	73	I	105	i
10	LF (Line feed)		42	*	74	J	106	j
11	VT (Vertical Tab)		43	+	75	K	107	k
12	FF (Form feed)		44	,	76	L	108	l
13	CR (Carriage return)		45	-	77	M	109	m
14	SO (Shift Out)		46	.	78	N	110	n
15	SI (Shift In)		47	/	79	O	111	o
16	DLE (Data link escape)		48	0	80	P	112	p
17	DC1 (Device control 1)		49	1	81	Q	113	q
18	DC2 (Device control 2)		50	2	82	R	114	r
19	DC3 (Device control 3)		51	3	83	S	115	s
20	DC4 (Device control 4)		52	4	84	T	116	t
21	NAK (Negative acknowl.)		53	5	85	U	117	u
22	SYN (Synchronous idle)		54	6	86	V	118	v
23	ETB (End of trans. block)		55	7	87	W	119	w
24	CAN (Cancel)		56	8	88	X	120	x
25	EM (End of medium)		57	9	89	Y	121	y
26	SUB (Substitute)		58	:	90	Z	122	z
27	ESC (Escape)		59	;	91	[	123	{
28	FS (File separator)		60	<	92	\	124	
29	GS (Group separator)		61	=	93	]	125	}
30	RS (Record separator)		62	>	94	^	126	~
31	US (Unit separator)		63	?	95	-		
127	DEL (Delete)							

Figura 1.9: Tabella ASCII

## std::string

**Definizione:** Il C++ ha una propria definizione per rappresentare una sequenza di caratteri come un oggetto di una classe. Questa classe è chiamata std::string. Questa memorizza i caratteri come una sequenza di bytes con la funzionalità di poter accedere al singolo carattere byte.

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

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

Queste sono solo alcune delle funzioni della classe string.

```
#include <iostream>

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

'\n' vs "\n"

'\n' è un carattere.

"\n" è un puntatore ad un array di caratteri (una stringa), ovvero l'equivalente di { '\n', '\0' } (\n più il terminatore nullo).

Quale è meglio utilizzare dipende dalla situazione: in una std::string per esempio un **char** non si può inserire (quindi '\n' non funzionerebbe), ma un **const char\*** sì.

---

## *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 e racchiuso tra due parentesi graffe c'è 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 e 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 alcun 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 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;
}
```

Tutti i programmi in C++ devono avere un **main**, senza non funzionano.

Senza il **main** un errore verrebbe lanciato dal compilatore: *undefined reference to 'main'*.

Se modificassimo il nome del **main** verrebbe lanciato anche in questo caso un errore perché il **main** non esisterebbe.

Il **main** viene invocato automaticamente quando il programma viene eseguito; è il punto di origine di ogni programma.

## *Funzioni ricorsive*

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

```
#include <iostream>

// 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.

```
#include <iostream>

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

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

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

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

```

        :: endl; // Output : 3
std::cout << "Valore di y dopo lo swap: " << y << std
        :: endl; // Output : 5

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

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

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

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

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

```

## *Funzioni che ritornano puntatori*

Si possono, naturalmente, ritornare i puntatori dalle funzioni.

```
#include <iostream>

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

```

### *Parametri di default*

Nel caso un programmatore non passi un argomento, possiamo impostare un **argomento di default** così che se non viene passato, viene utilizzato l'**argomento di default**.

Per settarli bisogna porre l' = e poi il valore di default dopo il nome dell'argomento.

```

#include <iostream>

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

// L'argomento b ha un valore di default di 0 (avremmo potuto mettere
// qualsiasi altro valore).
int somma2(int a, int b = 0)
{
    return a + b;
}

int somma3(int a = 2, int b = 7)
{
    return a + b;
}

int somma4(int a = 6, int b)
{
    return a + b;
}

int main()
{

```

```

std :: cout << somma(5) << std :: endl; //Output: Errore
    perché è stato passato solo un parametro su due.

std :: cout << somma2(5) << std :: endl; //Output: 5
    (perché il secondo parametro è di default 0)

std :: cout << somma2(5, 3) << std :: endl; //Output: 8
    (perché in questo caso abbiamo passato un secondo parametro e
     quindi viene utilizzato quello al posto del parametro di default)

std :: cout << somma3() << std :: endl; //Output: 9
    (perché abbiamo impostato dei valori di default di 2 e 7)

std :: cout << somma4(, 10) << std :: endl; //Output:
    Errore questo non è possibile! e neppure somma4(10);
return 0;
}

```

Nel capitolo *Le gemme degli Algoritmi* a pag.323 useremo gli **std::optional** che ci aiuteranno con i parametri di default.

## ***Funzione constexpr***

Porre **constexpr** in una funzione significa: che quella funzione può essere valutata a tempo di compilazione e l'espressione può essere usata in altre espressioni costanti.

```

#include <iostream>

constexpr int product(int x, int y)
{
    return x * y;
}

int main()
{
    const int x = product(5, 4);

    std :: cout << x << std :: endl; //Output: 20
    return 0;
}

```

- Nel C++11 una funzione `constexpr` non può avere più di un `return` a differenza del C++14 e avanti.
- Queste funzioni dovrebbero riferire solo a variabili globali costanti.
- Queste funzioni possono soltanto chiamare altre `funzioni constexpr`, non le normali funzioni.
- Queste funzioni non dovrebbero essere di tipo `void`.
- Alcuni operatori come l'incremento prefisso (`++i`) non sono permessi nel C++11, ma dal C++14 in avanti sì.

## *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.

```
#include <iostream>

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.

```

#include <iostream>

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.

```

#include <iostream>

int x = 10;

```

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

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

### *Perché evitare le variabili globali*

In generale, sarebbe meglio evitare di usare variabili globali perché:

- Tutte le funzioni hanno accesso alle variabili globali, il che renderebbe difficile, nel lungo andare, capire chi scrive e legge queste variabili.
- Non c'è un controllo su chi può accedervi.
- Peggiorano la chiarezza del codice.

---

## *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 definizioni 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. (a meno 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 | Header Guards | pragma once | ifndef*

**Definizione:** Se un file header viene incluso due volte, il compilatore processerà il suo contenuto due volte, il che risulterà in un errore. Per evitare questo c'è una procedura standard, chiamate *Header Guards*, 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 così fosse, allora lo aggiunge, altrimenti salta il suo contenuto così che non verrà aggiunto due volte. Inoltre, per fare questa stessa operazione, ma più semplice e corta esiste una direttiva non-standard: **#pragma once**.

```
#pragma once
```

// Contenuto dell'header.

## *Cosa sono le librerie?*

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

### *Header files libreria Standard*

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

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

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

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

## *Librerie create dagli utenti*

Gli utenti possono comporre 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 è 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 <i>extern "C"</i> Esprime l'intento che si usa il C (o perlomeno si può pensare così) Dal punto di vista del C, il codice C sarà visibile, mentre quello del C++ sarà invisibile.	È solo per C++ Non funzionerà con il C.  Esprime l'intento che si usa il C++ (o perlomeno si può pensare così)

---

## *Namespaces*

---

**Definizione:** I 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**.

```
#include <iostream>

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;**

```
#include <iostream>

// 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 1.10: 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** utilizzarlo.

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.

```
#include <iostream>
```

```
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.

```
#include <iostream>

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.

```
#include <iostream>

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.

```
#include <iostream>

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.

```

#include <iostream>

#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 membro.
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.

## *Costruttori e Distruttori nelle Strutture*

È possibile definire dei **costruttori** e dei **distruttori** nelle **strutture**.

I **costruttori** sono delle speciali funzioni con lo stesso nome della struttura che vengono invocate quando un'istanza della struttura viene creata.

I **distruttori** sono delle speciali funzioni con lo stesso nome della struttura che servono per deallocare della memoria precedentemente allocata all'interno della struttura. Per indicare un **distruttore** mettiamo davanti al nome della struttura anche ~ (tilde).

```
#include <iostream>

struct MyStruct {
    int x;
    // Costruttore con lista di inizializzazione, la variabile y passata al
    // costruttore verrà poi data alla variabile membra x.
    MyStruct(int y) : x(y) {}
    ~MyStruct()
    {
        // Corpo del distruttore
    }
};
```

```

int main()
{
    struct MyStruct instance(5);

    std::cout << instance.x << std::endl; //Output: 5

    instance.x = 2;

    std::cout << instance.x << std::endl; //Output: 2

    return 0;
}

```

Questi concetti verranno approfonditi nella sezione sulle *Classi*.

## ***Union***

**Definizione:** La **union** è un tipo di struttura dove l’ammontare di memoria è un 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 a 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.

```

#include <iostream>

union nome_della_union {
    // tipi di dati
}; // Da notare il ; proprio come nelle strutture.

```

```
union var {
    int iVar;
    char cVar;
    float fVar;
};

int main()
{
    // In C++ non serve la keyword union.
    union var V1, V2, V3;

    V1.iVar = 33;
    V2.cVar = 33;
    V3.fVar = 33.33;

    std::cout << "V1 var: " << V1.iVar << std::endl;
    // Output: V1 var: 33
    std::cout << "V2 var: " << V2.cVar << std::endl;
    // Output: V2 var: !
    std::cout << "V3 var: " << V3.fVar << std::endl;
    // Output: V3 var: 33.33

    return 0;
}
```

### ***structure vs union***

<b>Structure</b>	<b>Union</b>
Usa la keyword <b>struct</b>	Usa la keyword <b>union</b>
<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>La memoria allocata è condivisa con i membri individuali della union.</p>
<p>Modificare un membro della struttura non modificherà gli altri membri.</p>	<p>Modificare un membro della union modificherà gli altri membri.</p>
<p>I membri individuali possono essere acceduti ad ogni momento.</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 pilastro, la trave portante, la pietra miliare della programmazione ad oggetti. È un tipo di dato definito dall'utente che contiene i propri dati e funzioni membro 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.

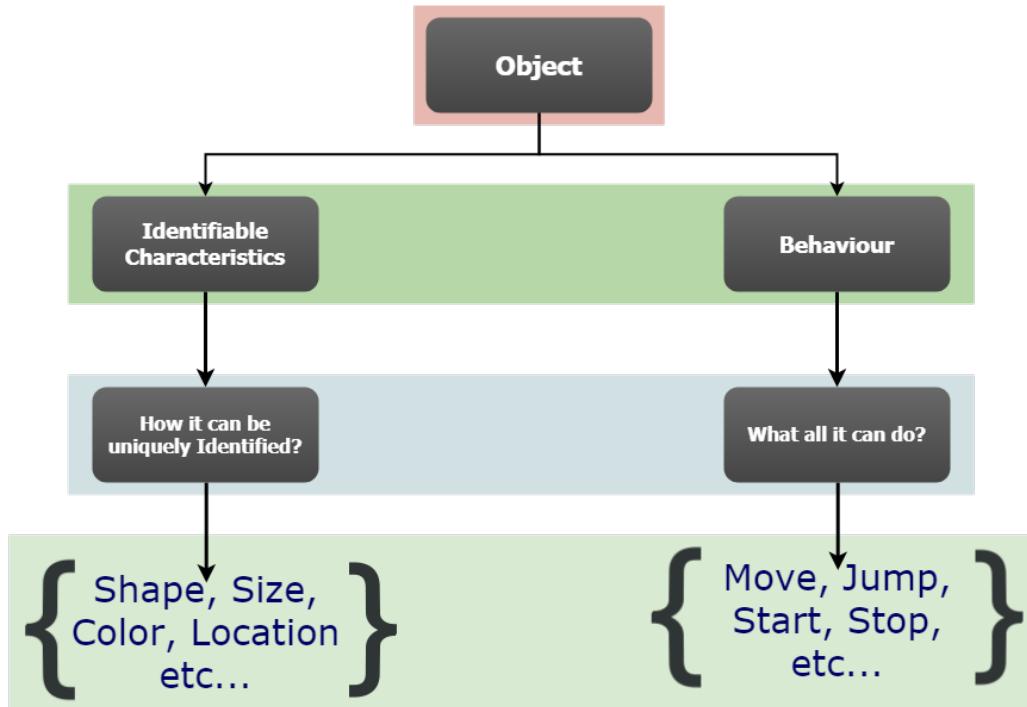


Figura 1.11: Dati e comportamenti di una classe

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 ha 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 virgolette.

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 : inizialization list
    Demo(int& val) : mVal(val)
    {

    }

    const int& mVal;
};

```

### ***Distruttori***

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

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

### ***Proprietà del distruttore***

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

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

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

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

### ***Access modifiers***

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

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

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

### ***Incapsulamento***

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

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

I dati dovrebbero perciò essere posti come **private**, mentre le funzioni che manipolano o recuperano questi dati come **public**.

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

```
#include <iostream>

class Sommatore {
    // con public sono accessibili da tutti.
public:
    // Costruttore.
    Sommatore( int i = 0){ // i = 0 vuol dire che assegniamo un
        // valore di default, casomai l'utente non voglia inserirne uno.
        totale = i;
    }

    // Interfaccia al mondo esterno.
    void aggiungiNumero( int numero){
        totale += numero;
    }

    // Interfaccia al mondo esterno.
    int getTotale(){
        return totale;
    }

private:
    // Dati nascosti al mondo esterno.
    int totale;
};

int main(){
    Sommatore s;

    s.aggiungiNumero(3);
    s.aggiungiNumero(6);
    s.aggiungiNumero(9);

    std::cout << "Totale: " << s.getTotale() << std::
        endl; // Output: Totale: 18
```

```
    return 0;
}
```

### ***scope resolution operator ::***

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

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

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

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

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

    void display();

    int get_num(){
        return num;
    }

private:
    int num;
};

void MyClass :: display(){
    std::cout << "Il valore di num 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.

Questo tipo di relazione è chiamata relazione **is-a** (è una).

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

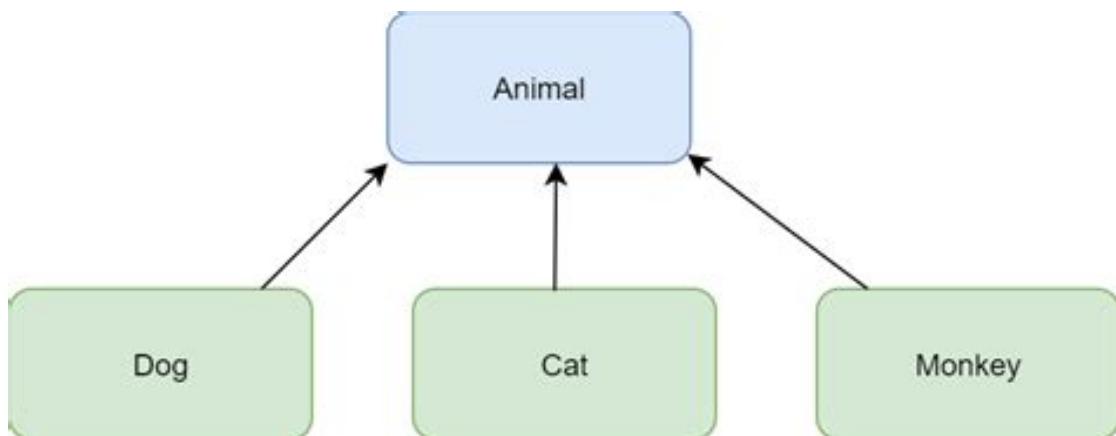


Figura 1.12: Concetto dell'ereditarietà

```
#include <iostream>

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.

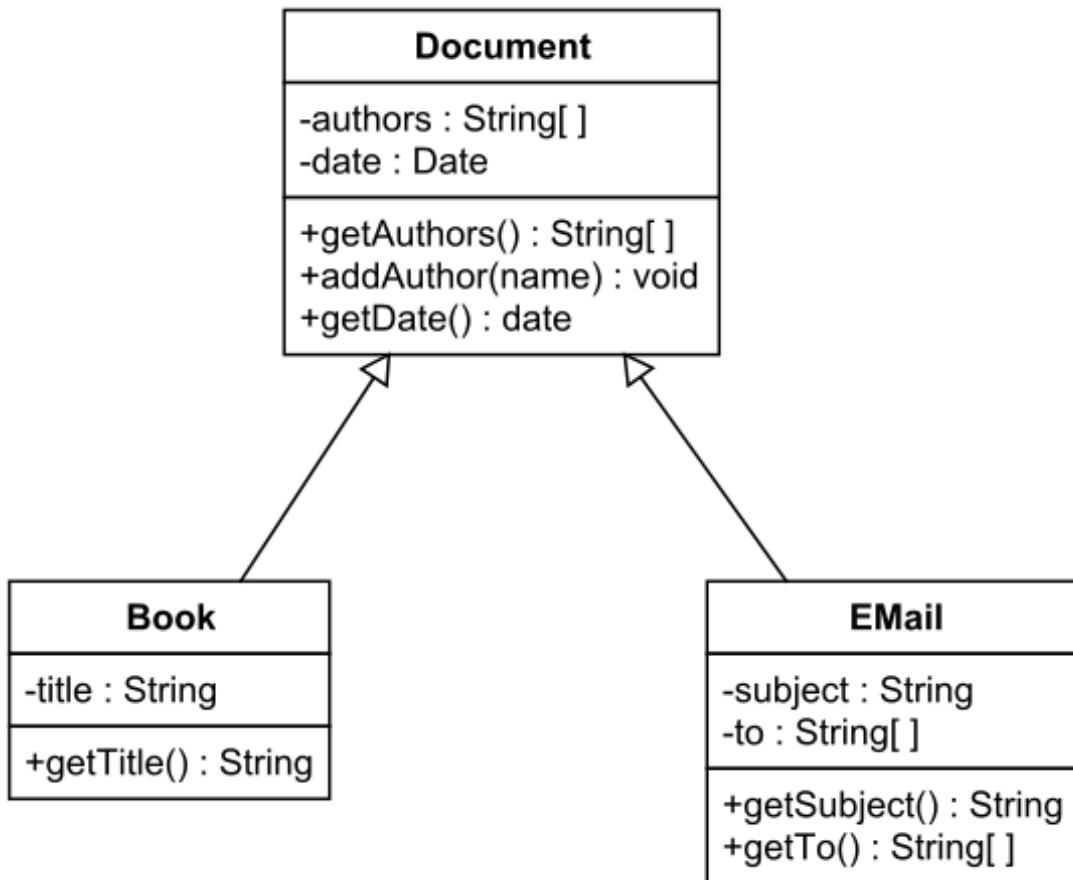


Figura 1.13: Ereditarietà in un diagramma UML

### ***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.

```

#include <iostream>

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

class B
  
```

```

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

class C: public B, public A // Da notare l'ordine.
{
    public:
C() { std::cout << "C's constructor called" <<
      std::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, ecc. 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 ad assicurarsi che non ci siano 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.h
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 li 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 1.14: Forward Declaration

Inoltre, potrebbe essere utile per evitare di includere molteplici headers in un header file.

#### Esempio:

Abbiamo l'header A, l'header B, l'header C, l'header D:

- Se nell'header A includiamo l'header B, questo prende spazio/aumenta il tempo di build.
- Se poi, l'header C ha bisogno di certi oggetti della classe B, includendo l'header B; include, di conseguenza, anche l'header A, anche se non aveva bisogno dell'header A, ma solo dell'header B.
- Se poi abbiamo l'header D che ha bisogno dell'header C e lo include, di conseguenza si prenderà anche l'header A e l'header B, prendendo ulteriore spazio/aumenta il tempo di build.
- Quindi, essenzialmente creiamo una matriosca che occupa spazio/aumenta il tempo di build.
- Ora, questo non dovrebbe essere un grosso problema se si includono le *header guards* a pag.56 che impediscono che uno stesso header file venga incluso più di una volta, quando avviene la fase di linking.

Se invece al posto usiamo la *Forward Declaration*, possiamo fare così:

Se nell'header A abbiamo bisogno di una classe B, al posto di includere l'header B, potremmo usare la *Forward Declaration* in questo modo:

```
// A.h
class A {
public:
    A() {}
    ~A() {}

    int x;
    int y;
    class B* b; // Forward Declaration (al posto di fare
                  #include "B.h", mettiamo class B, indicando al compilatore
                  che useremo una classe B, ma che non è importante, per ora,
                  sapere come questa classe B è implementata)
};
```

Quindi, nel nostro header A, inseriamo la dichiarazione della classe B, ma per l'implementazione di questa bisognerà andare nel file di implementazione (.cpp) dove verrà effettivamente incluso l'header B e quindi la definizione della classe B sarà presente.

```
// A.cpp
#include "A.h"
#include "B.h"

class A {
    A() {}
    ~A() {}

    int x;
    int y;
    B* b = new B();
};
```

Lo stesso possiamo fare per gli altri due headers C e D.

## *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 :: (scope resolution operator).

```
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;
}
```

The screenshot shows a code editor window with a dark theme. It displays a C++ file named 'Graphics.h'. The code defines a class 'MyClass' with a constructor and a static member function 'calcola\_qualcosa'. In the 'main' function, the static member function is called using the scope resolution operator ::. The code editor has syntax highlighting and shows line numbers 1 through 77.

```
if (left < 0)
{
    x_mobile = 5;
}
else if (right >= Graphics::ScreenWidth)
{
    x_mobile = gfx.ScreenWidth - 6; // Avrei potuto scrivere Graphics::ScreenWidth ; Graphics è il nome della classe e
    // ScreenWidth è una costante intera statica, ed è proprio 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)
{
    y_mobile = 5;
}

static constexpr int ScreenWidth = 800;
static constexpr int ScreenHeight = 600;
```

Figura 1.15: 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.

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

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

Figura 1.16: 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
private:
    int x;
    int y;
    int vx;
    int vy;
```

Figura 1.17: 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 <b>class</b> .	Si dichiara usando la keyword <b>struct</b> .

# 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 di 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 qualcosa' 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:** Un **vector** è un contenitore rappresentante un 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	->	= *i		++	==, !=
Output			*j=	++	
Forward	->	= *i	*j=	++	==, !=
Bidirectional		= *i	*j=	++, --	==, !=,
Random-Access	->[]	= *i	*j=	++, --, +=, -=, +, -	==, !=, <, >, <=, >=

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

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

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

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

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

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

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

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

```

    std :: cout << "Elemento: " << *it << std :: endl ;
}
```

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

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

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

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

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

std :: advance(it, 2);

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

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

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

```

// Copio gli elementi di v2 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.

```

## ***Range-based for loop | Foreach***

Questo tipo di loop, usufruisce sempre della keyword **for**, itera su un contenitore.

```

#include <iostream>
#include <vector>

for(tipo nomeVariabile : nomeContenitore)
{
    // Corpo del range-based for loop o foreach
}

std::vector<int> v{ 3, 7, 9, 12};

// La i (avremmo potuto darci qualsiasi altro nome apparte lo stesso nome
// del contenitore) è la variabile che rappresenta un elemento del
// contenitore v ad ogni iterazione del ciclo.
// Quindi alla prima iterazione i corrisponderà al primo elemento del
// contenitore, alla seconda il secondo elemento, così via finchè i
// corrisponderà all'ultimo elemento + 1 del contenitore v.
for(const int &i : v)
{
    std::cout << i << " ";
}

//Output: 3 7 9 12

```

i è la variabile che rappresenta ad ogni iterazione del ciclo un elemento del *container* in questo esempio.

Se non si conosce la tipologia del contenitore o per far prima si può anche utilizzare la keyword **auto**.

Questo è un loop molto efficiente quando utilizzato con i contenitori della libreria standard.

### ***Ottenere l'elemento per valore o per referenza a costante***

Quando è meglio utilizzare il **range-based for loop** per valore, per referenza e costante?

- **auto x** : quando si vuole lavorare con le copie.
- **auto &x** : quando si vuole lavorare con gli elementi originali e li si vuole modificare.
- **const auto &x** : quando si vuole lavorare con gli elementi originali, ma non si vuole modificarli.

### ***begin vs cbegin***

**begin** restituisce un *iterator* all'inizio mentre **cbegin** restituisce un *const\_iterator* all'inizio.

Quindi, **begin** che restituisce un *iterator* permette di modificare ciò a cui si sta puntando, mentre **cbegin**, ovvero il *const\_iterator* non ti permetterà di modificare il valore di ciò a cui sta puntando.

Lo stesso vale per **end()** e **cend()**.

### ***rend()***

**rend()** permette di restituire un iteratore inverso (quella r sta per *reverse*). (quindi parte dalla fine e finisce all'inizio)

---

## 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:** L' 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.

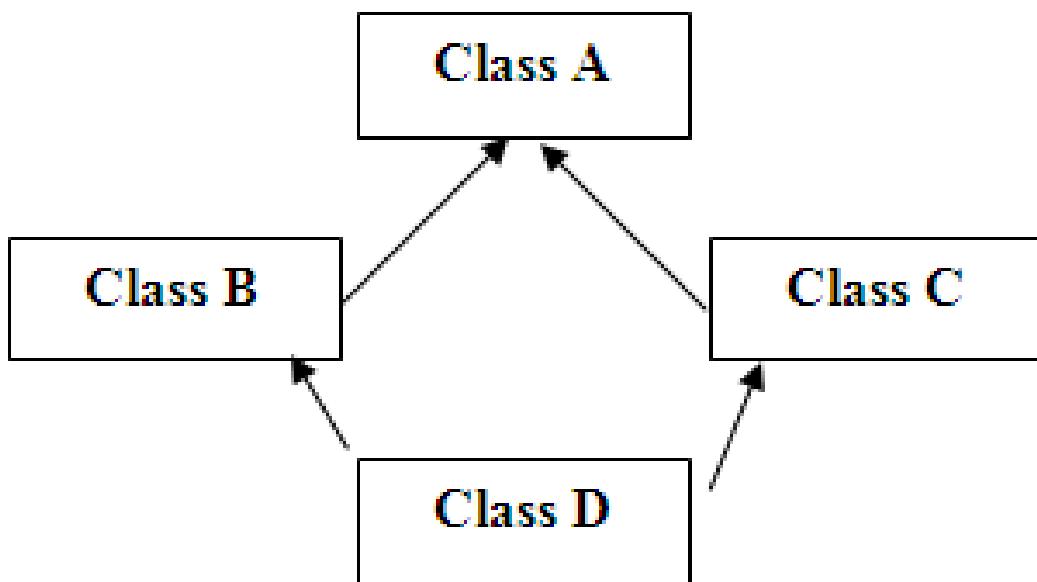


Figura 2.2: Problema del diamante

Come possiamo notare i dati e le funzioni della **classe A** sono ereditati 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.105).

```
#include <iostream>

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

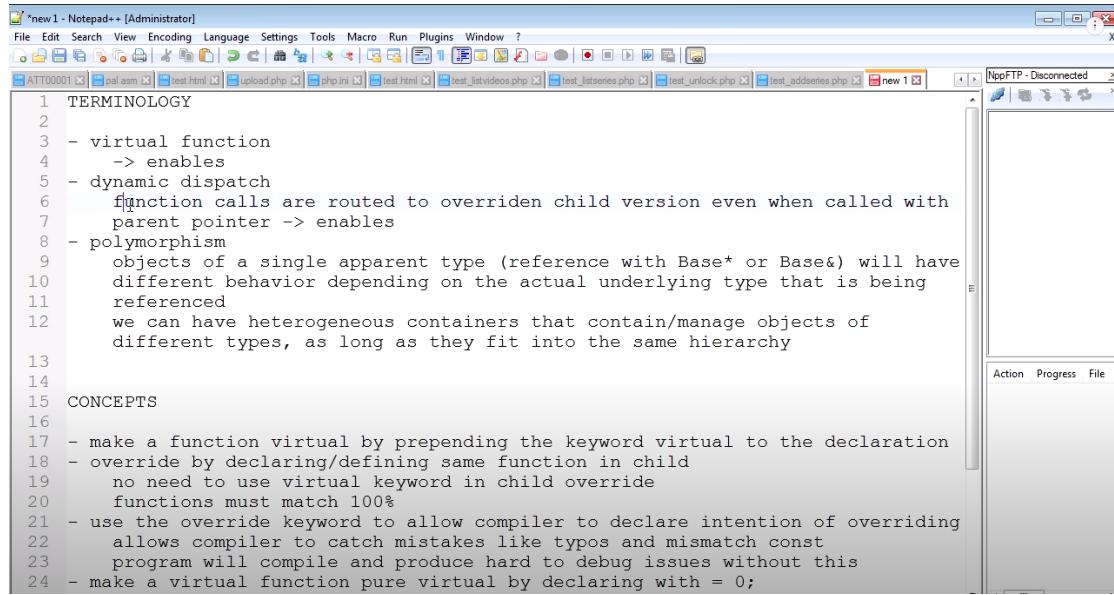
class B : public virtual A {
};

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**.

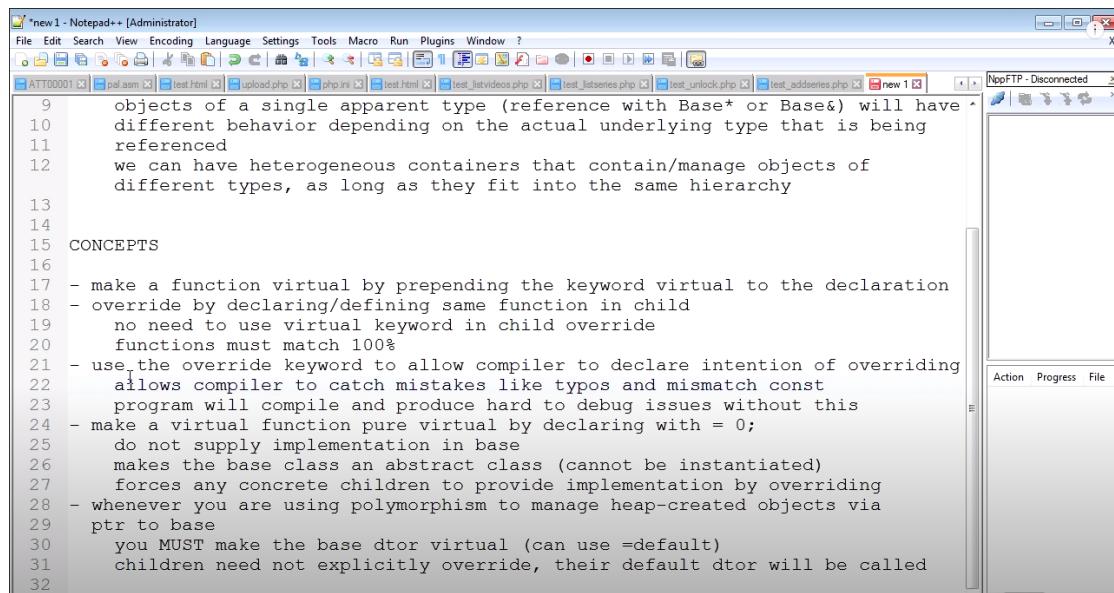


```

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

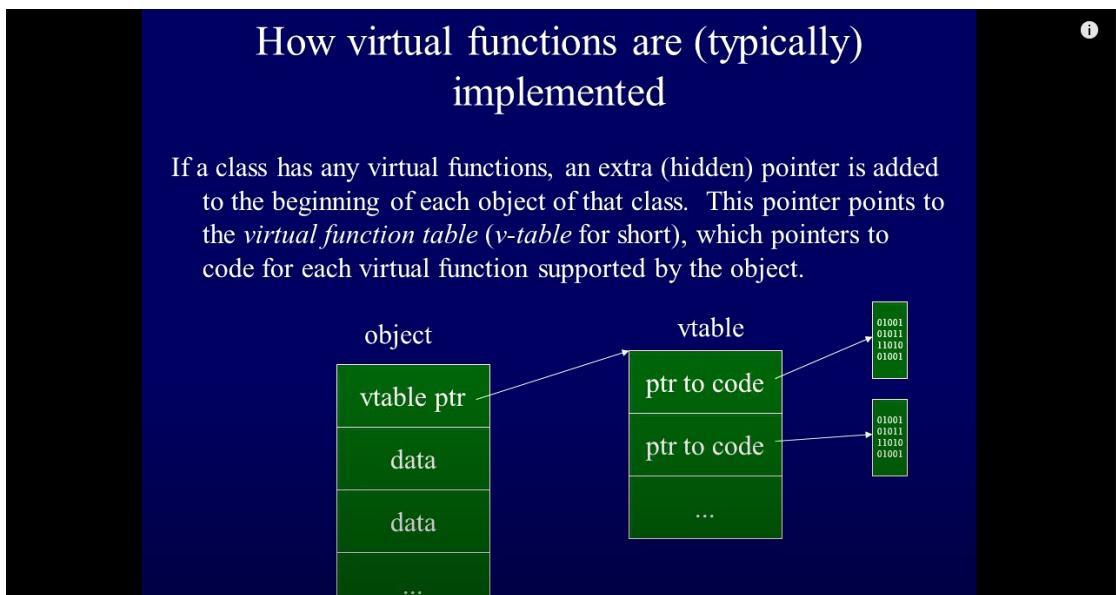


Figura 2.5: Virtual Functions Implementazione

---

## 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**).

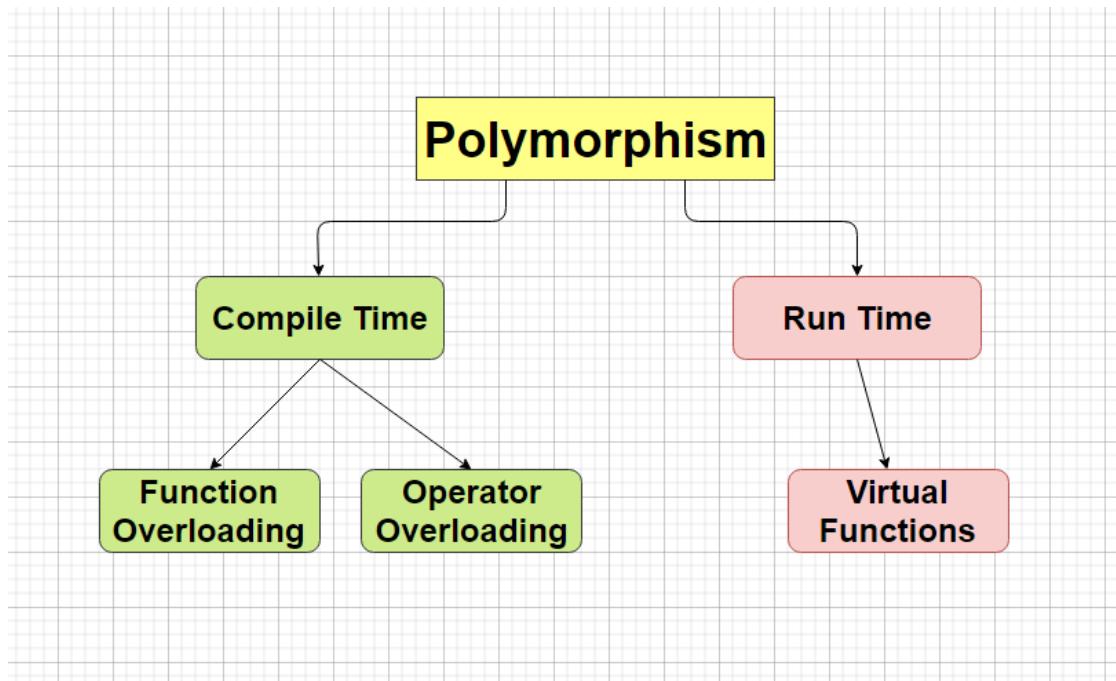


Figura 2.6: Polimorfismo

---

## *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 *overloadare*.

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

## *Overloading vs Overriding*

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

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

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

Può essere fatta solo nelle classi derivate.

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

<b>Overloading</b>	<b>Overriding</b>
Nessuna keyword è usata.	Keyword <b>override</b> .
Il prototipo cambia in base ai parametri.	Il prototipo non cambia.
Occorre durante compile time.	Occorre durante runtime.
I costruttori possono essere "overloaddati".	
I distruttori non possono essere "overloaddati".	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 <b>virtual</b> nella classe base.
Le funzioni verrebbero ridefinite con lo stesso nome, ma differente numero o tipo di parametri.	
	L'indirizzo dell'oggetto della classe è assegnato al puntatore la cui funzione è chiamata dal puntatore.
	Quando la funzione è definita viene preceduta dalla keyword <b>virtual</b> nel main.

---

## *Tipi di Casts*

---

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

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

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

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

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

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

```
#include <iostream>

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

int main()
{
    const char* c = "testo";
    // Ci serve per poter passare un puntatore a char const ad una
    // funzione che prende un puntatore a char senza const.
    print(const_cast<char*>(c)); // Output: testo
    return 0;
}
```

## *dynamic\_cast<>*

- **dynamic\_cast<>** : Serve esclusivamente per i casts riguardanti il polimorfismo. Puoi castare un puntatore o una reference a qualsiasi altro tipo di classe. Non solo si può fare un casting verso il basso, ma anche verso l'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
<b>dynamic_cast</b>	per convertire puntatori/references in una gerarchia di ereditarietà.
<b>static_cast</b>	per le conversioni di tipi ordinari.
<b>reinterpret_cast</b>	per reinterpretare bit patterns di basso livello. Usare con cautela.
<b>const_cast</b>	per aggiungere/rimuovere <b>const</b> al cast.

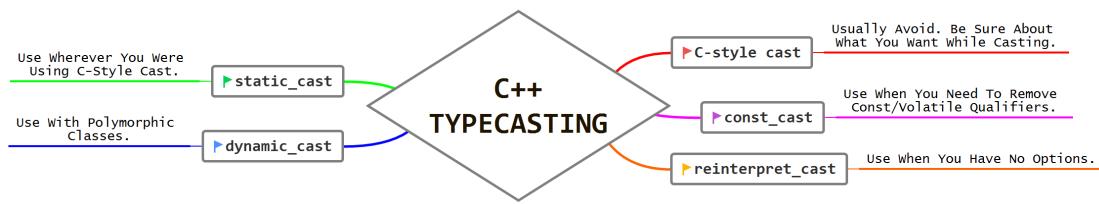


Figura 2.7: Typecasting

```

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>[&amp;]</code>	accedere a tutte le variabili per reference.
<code>[nomeVariabile1, &amp;nomeVariabile2]</code>	"cattura" nomeVariabile1 per valore e nomeVariabile2 per referencia.

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

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

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

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

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

int n = 7;
// trova il primo numero maggiore di n.
```

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

std::cout << "Il primo numero maggiore di n e': " <<
*p << std::endl; // Output: Il primo numero maggiore di n e': 8

// Qui [=] vuol dire che possiamo accedere a tutte le variabili.
int countN = std::count_if(v1.begin(), v1.end(), [=](int a)
{
    return a >= n;
});

std::cout << "Il numero di elementi piu' grandi o
uguali ad n sono: " << countN << std::endl; //
Output: Il numero di elementi più grandi o uguali ad n sono: 4
(perchè abbiamo inserito anche il 24 nell'operazione precedente).
```

## ***mutable***

**Definizione:** La parola chiave **mutable** serve per modificare certi dati membri di una classe o struttura attraverso una funzione costante o una lambda (quando le variabili sono catturati per valore) anche se non vuoi modificare gli altri membri.

```
#include <iostream>

int main()
{
    int a = 11;
    auto func = [a]() mutable -> int { ++a; std::cout
        << a << '\n'; return a; };

    func(); //Output: 12
    return 0;
}
```

---

## *Memoria dinamica*

---

**Definizione:** Riguarda l'allocazione di memoria allocata 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.

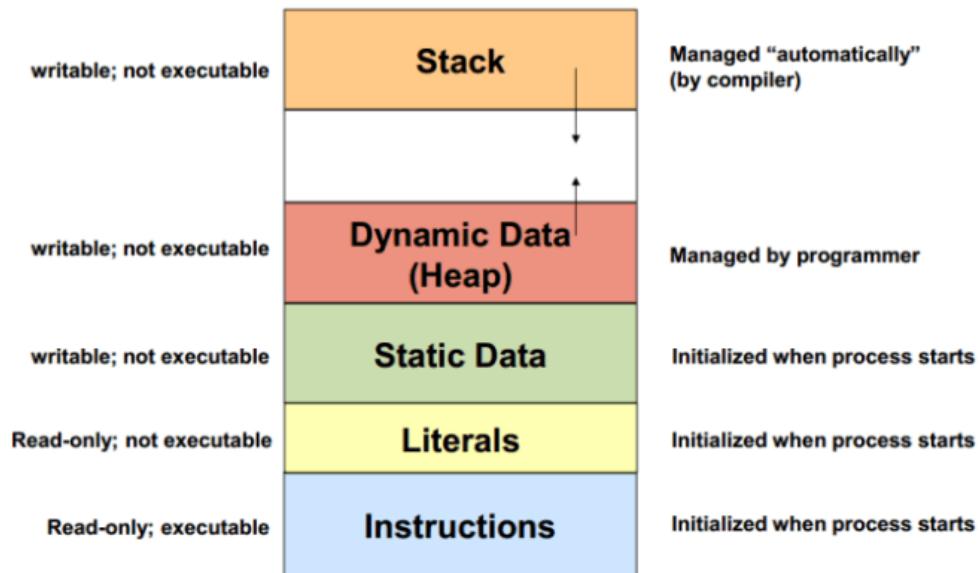


Figura 2.9: Memoria

## *Memoria Dinamica in C*

In C per l'allocazione dinamica della memoria usufruivamo di 4 diverse funzioni: **malloc()** (per allocare), **calloc()** (alloca e setta la memoria a zero a differenza della **malloc**), **realloc()** (per riallocare), **free()** (per liberare la memoria).

Tutte questi funzioni del C, esistono anche nel C++, ma questo ha un suo modo per allocare memoria dinamicamente.

***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.

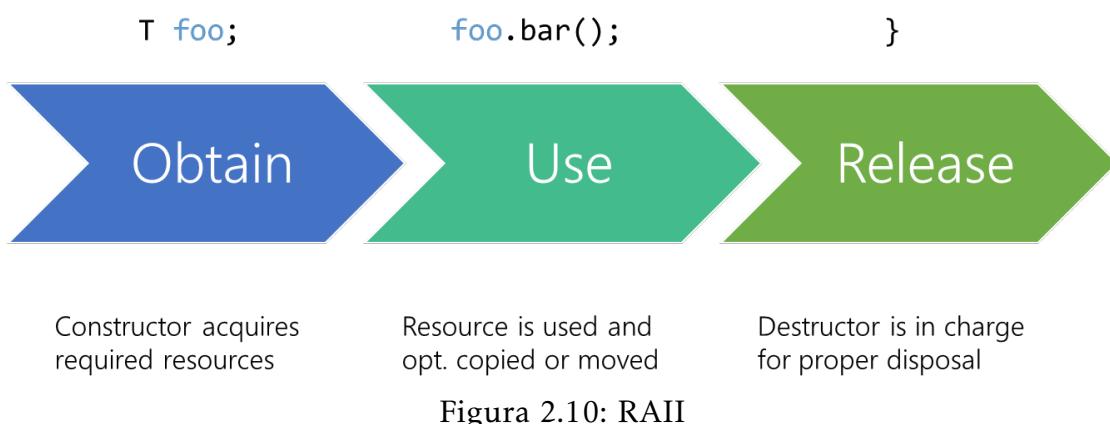


Figura 2.10: RAII

---

## *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:

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
// Assegniamo 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 permettere la copiatura di
                  // oggetti.
    b2 = b1; // Chiama il copy assignment operator; da errore perché
              // abbiamo messo = delete per non permettere 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 bitwise viene creata se l'assignment operator non viene overloaded.

## *Rule of Three*

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

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

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

## *Move Constructor*

**Definizione:** I **Move 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 le 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 assegnato 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**'.

## *xvalue, glvalue, prvalue*

Un **xvalue** (*eXpring*) riferisce ad un oggetto, di solito vicino la fine del suo tempo di vita.

Un **xvalue** è il risultato di certi tipi di espressioni riguardanti le referenze **rvalues**.

Un **gvalue** (*generalized lvalue*) è un **lvalue** o un **xvalue**.

Un **prvalue** (*pure rvalue*) è un **rvalue** che non è un **xvalue**. (Per esempio: il risultato di chiamare una funzione il cui tipo di ritorno non è una reference è un **prvalue**).

Category	Can be moved from ( m )	Can't be moved from ( M )	
Have identity ( i )	xvalue	lvalue	→ glvalue ( i )
Have not identity ( I )	prvalue		
			rvalue ( m )

Figura 2.11: Categorie di valori

## *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 overloaddato.

    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 2.12: 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.

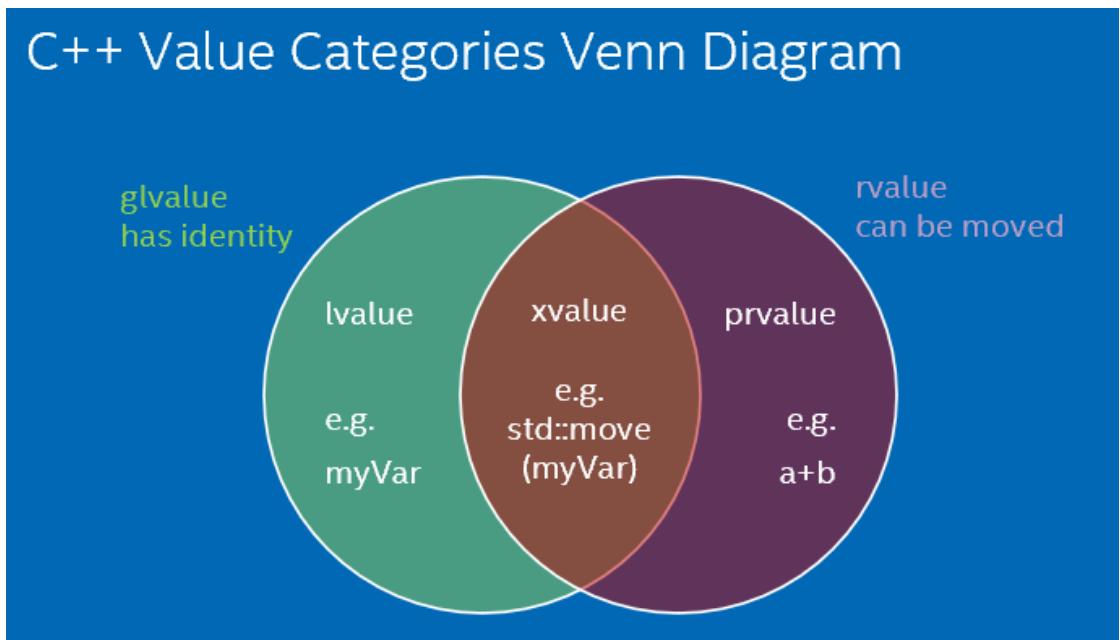


Figura 2.13: Value Categories

### *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.

## *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 semplici 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 cout dopo il return.
    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.

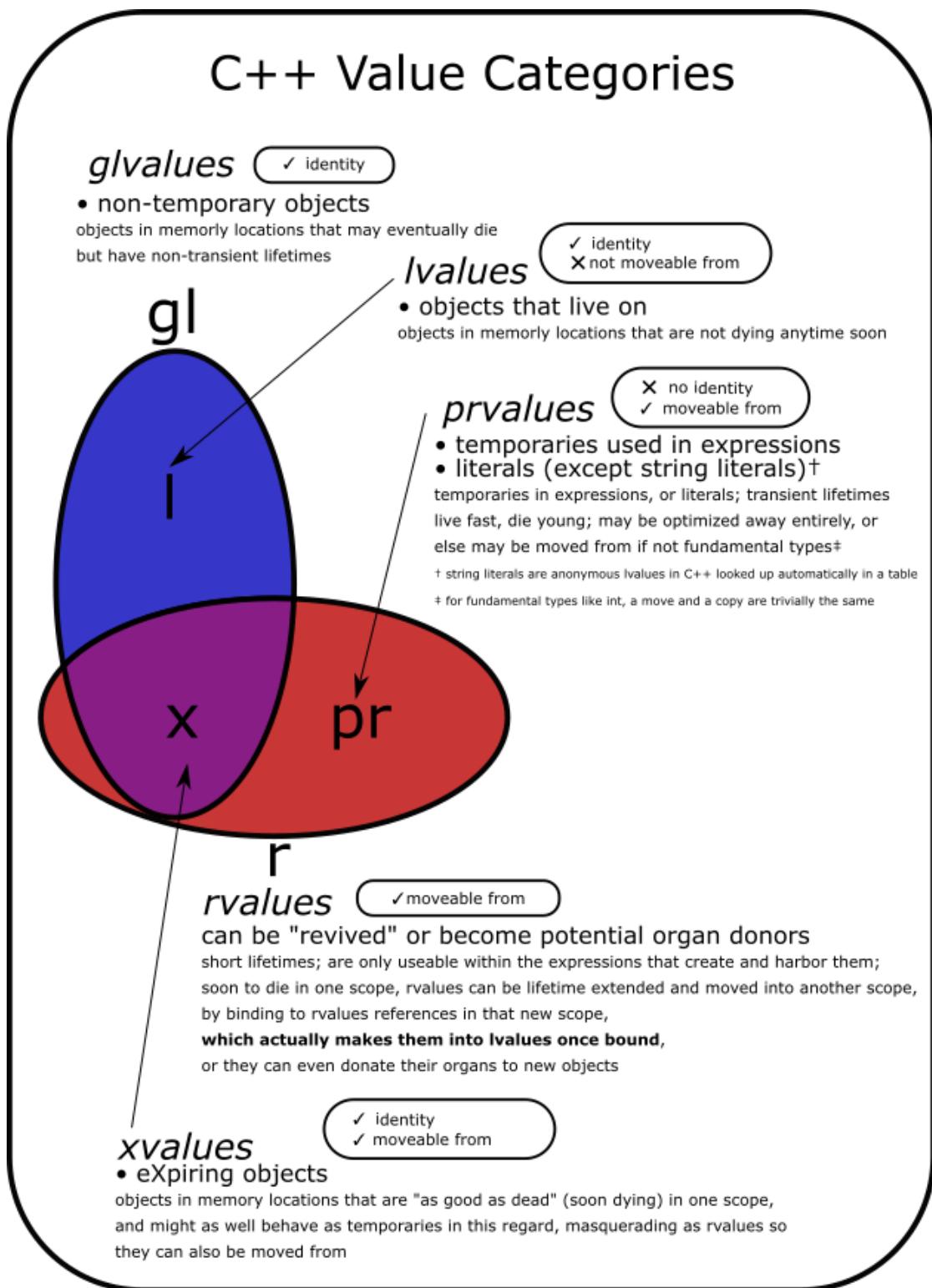


Figura 2.14: Value Categories

---

## *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 definire (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 l' *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 soltanto se un errore sarà accorso nel **try**.

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

```
// Blocco di codice del catch.  
}
```

C'è uno speciale 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 (sistematici) 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 2.15: Recap Eccezioni

## *Assertions*

**Definizione:** Le **assertions** (asserzioni) 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> // si potrebbe anche usare <cassert>, ma
                    sembra non avere molto senso utilizzarlo (dal C++11 meglio non
                    inquinare il namespace globale con qualsiasi "c.." header file).
```

```
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 questo 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 a std::endl a meno che non sia necessario ripulire il buffer dello stream.

### *Meglio '\n' oppure "\n"*

Come trattato nel capitolo *Basi del Linguaggio* a pag.44 il '\n' è un carattere, mentre "\n" è un array di due caratteri.

Quindi, nonostante questo probabilmente non avrà un grande impatto, comunque è probabilmente meglio usare '\n' quando lo si vuole usare al posto dell'endl.

- '\n' : per finire la frase (quando si usa cout).
- "\n" : in una stringa.
- std::endl : quando si vuole terminare la frase e pulire (flush) 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 <istream>
#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 `<iostream>` :

- **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 2.16: 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 <fstream>
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 2.17: Otttenere 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 serve
// più spazio
delete [] warp_string; // Dealloco la memoria
warp_string = nullptr; // Pongo il puntatore a null

```

Figura 2.18: Otttenere il file size

---

## ***std::chrono***

---

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

Fornisce una *precisione neutrale*, attraverso la separazione delle 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 referenza 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 2.19: 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> // si può anche includere <ctime>.

// 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:

<b>linear_congruential_engine</b>	Implements linear congruential algorithm
<b>mersenne_twister_engine</b>	Implements Mersenne twister algorithm
<b>subtract_with_carry_engine</b>	Implements subtract with a carry algorithm

Figura 2.20: Motori di numeri pseudo-casuali

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

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

int main()
{
    // Trova il tempo tra il clock del sistema (tempo corrente) e l'epoch
    // del clock (ovvero 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 istanze di motori di generatori e adattatori:

<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 2.21: Motori di numeri pseudo casuali

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);

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 ne 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>.

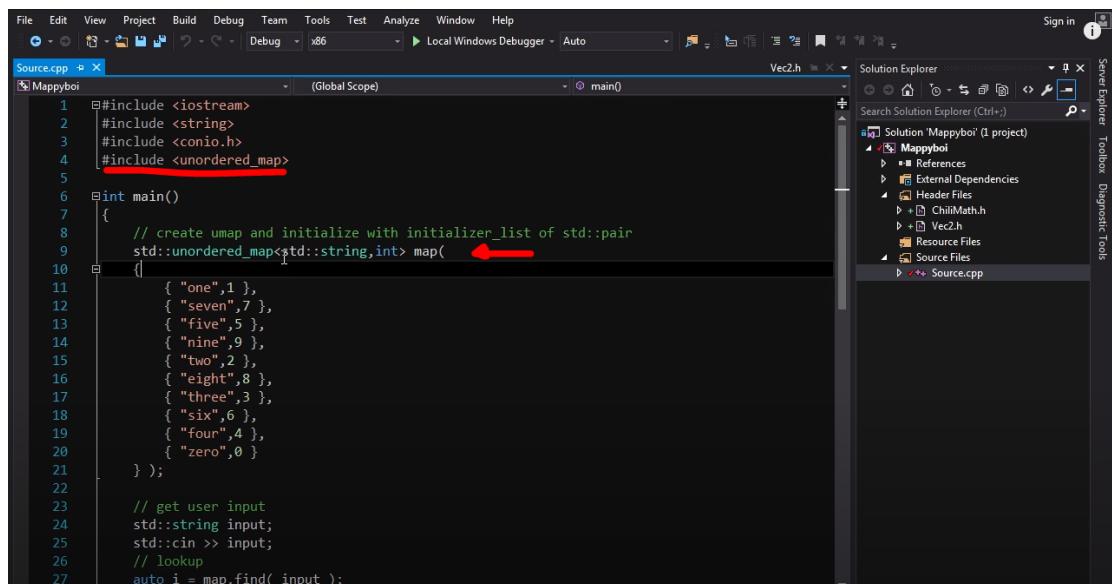
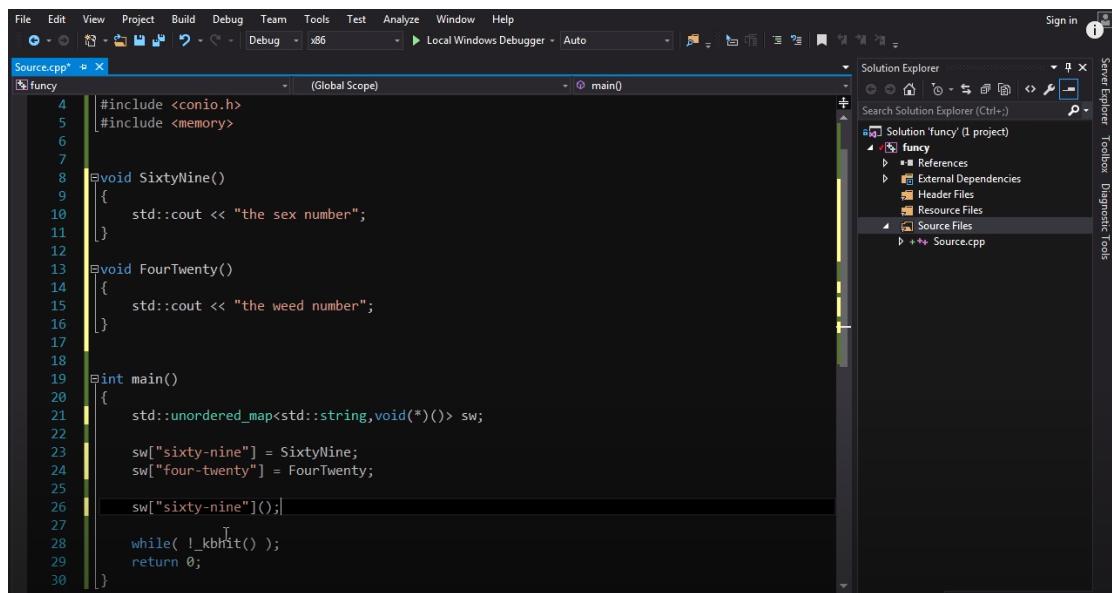


Figura 2.22: unordered\_map



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 file operations like Open, Save, and Build. The status bar at the bottom indicates "Local Windows Debugger - Auto". The main window displays the "Source.cpp" file under the "fancy" project. The code uses an `std::unordered_map<std::string, void(*)()>` to map strings to function pointers. The "Solution Explorer" pane on the right shows the project structure with files like "Source.cpp" and "Header Files".

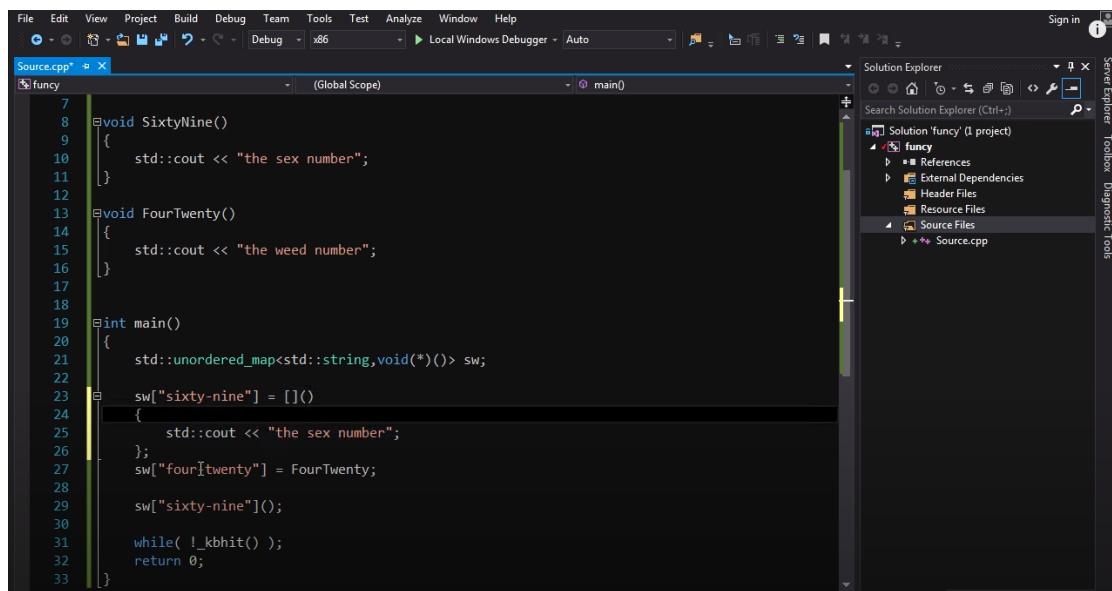
```

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 }

```

Figura 2.23: *unordered\_map* function pointer

### *unordered\_map function pointer*



This screenshot is similar to Figura 2.23, showing the same code in the "Source.cpp" file. The key difference is in line 23, where a lambda expression is used instead of a regular function pointer: `sw["sixty-nine"] = []()`. The rest of the code remains the same, demonstrating the use of `std::unordered_map` with both function pointers and lambda functions.

```

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"] = []()
24     {
25         std::cout << "the sex number";
26     };
27     sw["four[twenty]"] = FourTwenty;
28
29     sw["sixty-nine"]();
30
31     while( !_kbhit() );
32     return 0;
33 }

```

Figura 2.24: *unordered\_map* function pointer

```

File Edit View Project Build Debug Team Tools Test Analyze Window Help
Source.cpp -> X
funcy
4 #include <conio.h>
5 #include <memory>
6 #include <functional> // Red arrow points here
7
8
9 EVoid SixtyNine()
10 {
11     std::cout << "the sex number";
12 }
13
14 EVoid FourTwenty()
15 {
16     std::cout << "the weed number";
17 }
18
19
20 Eint 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["sixty-nine"]();

```

**WRAPPER ARAND CALLABLE**

Figura 2.25: unordered\_map function pointer

```

File Edit View Project Build Debug Team Tools Test Analyze Window Help
Source.cpp -> X
funcy
16     std::cout << "the weed number";
17 }
18
19
20 Eint 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
34 }

```

Figura 2.26: unordered\_map function pointer

```

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          // EMPTY FUNCTION THAT DOES NOTHING
25      }
26  }
27  private:
28  std::unordered_map<std::string, std::function<void()>> map;
29  std::function<void()> def = []{}();
30 };
31
32 int main()
33 {
34     while( !_kbhit() );
35     return 0;
36 }

```

Figura 2.27: 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:** Un **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 insiemi 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()**.

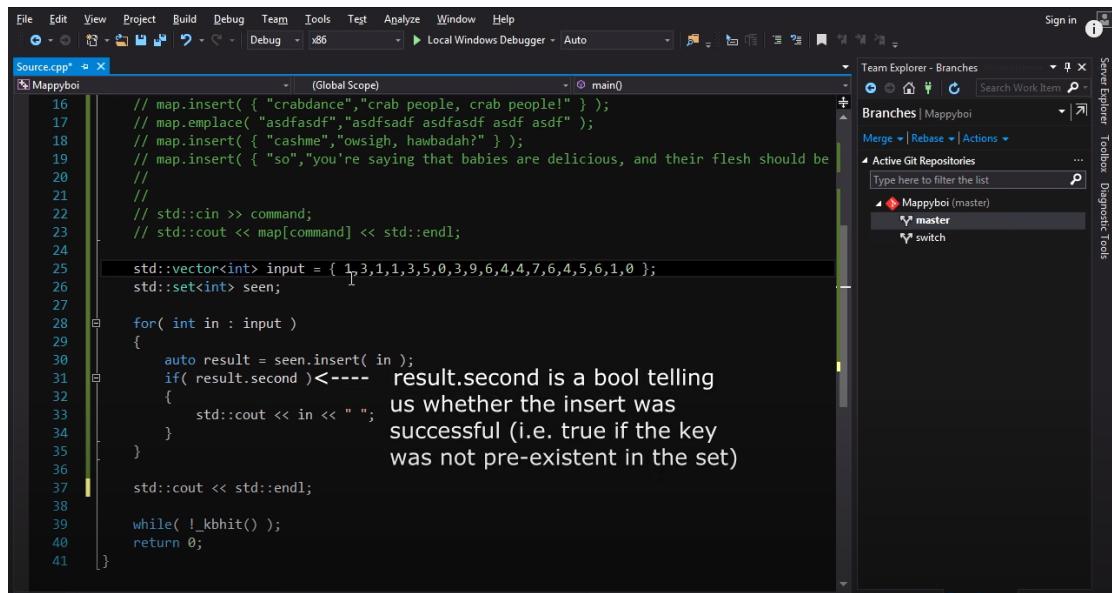


Figura 2.28: set

### *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, spaccettare 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 spaccettato.

```

#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 dela
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 spacciati della tupla
sono: " << c << " " << x << " " << f << std::
endl; //Output: I valori spacciati 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*

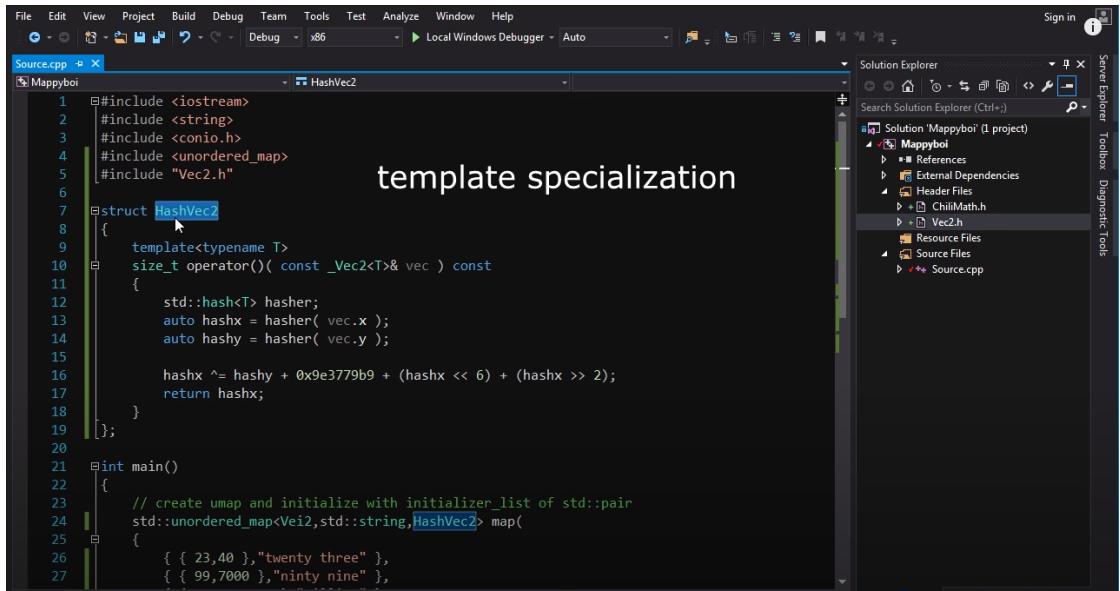
Quali sono le differenze tra **pair** e **tuple**?

Pair	Tuple
Può contenere soltanto una coppia di valori.	Può contenere molteplici valori.
Un Pair è uno 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 top menu bar includes File, Edit, View, Project, Build, Debug, Team, Tools, Test, Analyze, Window, Help, and a Local Windows Debugger dropdown set to Auto. The status bar at the bottom right shows 'Sign in' and a help icon.

The main window displays the 'Source.cpp' file under the 'Mappybot' project. The code is as follows:

```

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, std::string, HashVec2> map(
25     {
26         { { 23,40 }, "twenty three" },
27         { { 99,7000 }, "ninety nine" },
28     }
29     );
30
31     for (const auto& p : map)
32     {
33         cout << p.first.x << ", " << p.first.y << ": " << p.second << endl;
34     }
35 }

```

A callout box labeled 'template specialization' points to the line 'template<typename T>' in the HashVec2 class definition.

The Solution Explorer on the right shows the 'Mappybot' solution with files like 'ChilliMath.h', 'Vec2.h', and 'Source.cpp'.

Figura 2.29: 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.

```

323 void DoSpecials( MemeFighter& f1,MemeFighter& f2 ){ ... }
339
340 bool AreSameType( MemeFighter& f1,MemeFighter& f2 )
341 {
342     if( typeid(f1) == typeid(f2) )
343     {
344         return true;
345     }
346     return false;
347 }
348
349 int main()
350 {
351     std::vector<MemeFighter*> t1 = {
352         new MemeFrog( "Dat Boi",new Fists ),
353         new MemeStoner( "Good Guy Greg",new Bat ),
354         new MemeCat( "Haz Cheeseburger",new Knife ),
355     };
356     std::vector<MemeFighter*> t2 = {
357         new MemeCat( "NEDM",new Fists ),
358         new MemeStoner( "Scumbag Steve",new Bat ),
359         new MemeFrog( "Pepe",new Knife )
360     };
361
362     const auto alive_pred = []() MemeFighter* pf ) { return pf->IsAlive(); };
363     while(
364         std::any_of( t1.begin(),t1.end(),alive_pred ) &&

```

Figura 2.30: typeid

```

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: 'Duuuuuude.\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 2.31: 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:** L' **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* (che significa sovrastante, 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 qualificatore **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.

## ***Funzioni Inline vs Funzioni constexpr***

Sono entrambi per migliorare le *performance*.

Le **funzioni inline** richiedono al compilatore di espandere a tempo di compilazione e di risparmiare gli *overhead* delle chiamate alle funzioni.

Nelle **funzioni inline** le espressioni sono sempre valutate a *run time*, mentre le **funzioni constexpr** vengono valutate a *compile time*.

## ***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, 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.

Essenzialmente con questa keyword non permettiamo di passare al costruttore un argomento di tipologia diversa da quella richiesta. (se non la mettessimo, la tipologia passata verrebbe implicitamente convertita alla tipologia richiesta dal costruttore).

Se, per esempio, il costruttore richiedesse un **int** e noi passassimo un **char**, senza la keyword **explicit**, il **char** verrebbe implicitamente convertito in **int**.

```

#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 quindi
// per poter far questo dobbiamo utilizzare la keyword "explicit" nel costruttore.

```

Figura 2.32: 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* );
};

// Dichiarata all'esterno, ma 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 allocato.

    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, ovvero 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.

---

## 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.

# 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 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 membro 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 di 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 deallocata, 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 deallocata, perché ci siamo scordati di usare *delete p*; alla fine della funzione.

Ciò è un problema perché verrà sempre allocata altra memoria e mai deallocata, 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 *deprecati* (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));

```

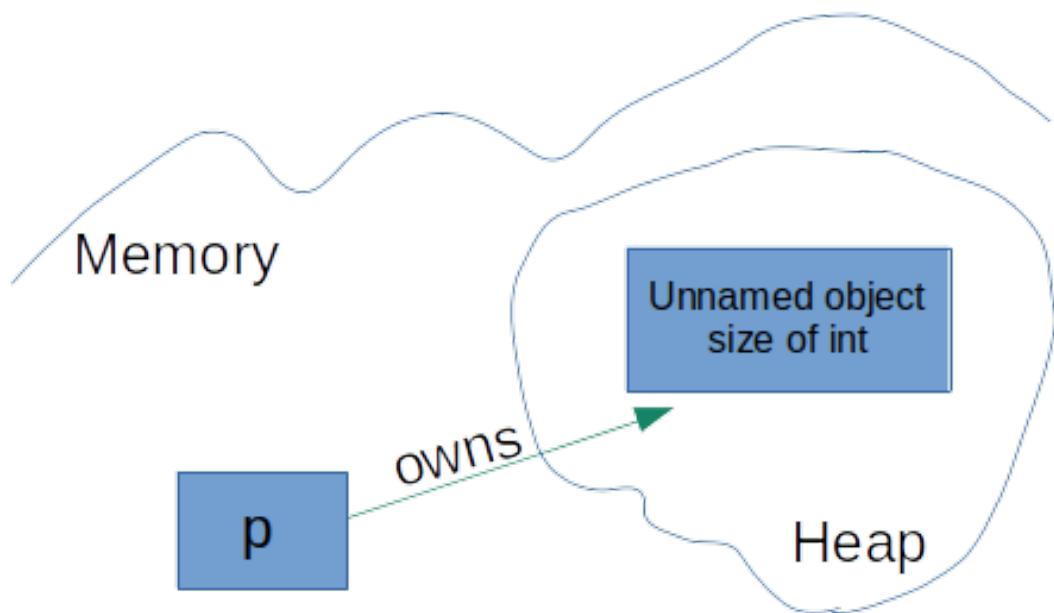


Figura 3.1: Unique ptr

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

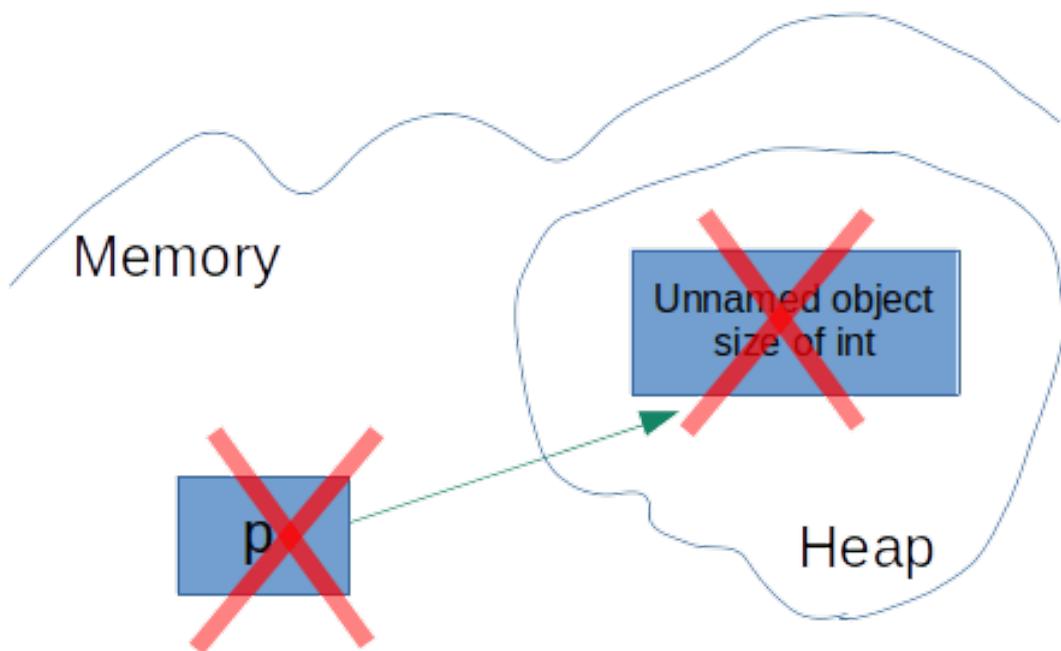


Figura 3.2: Unique ptr delete

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, questo 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;
}
```

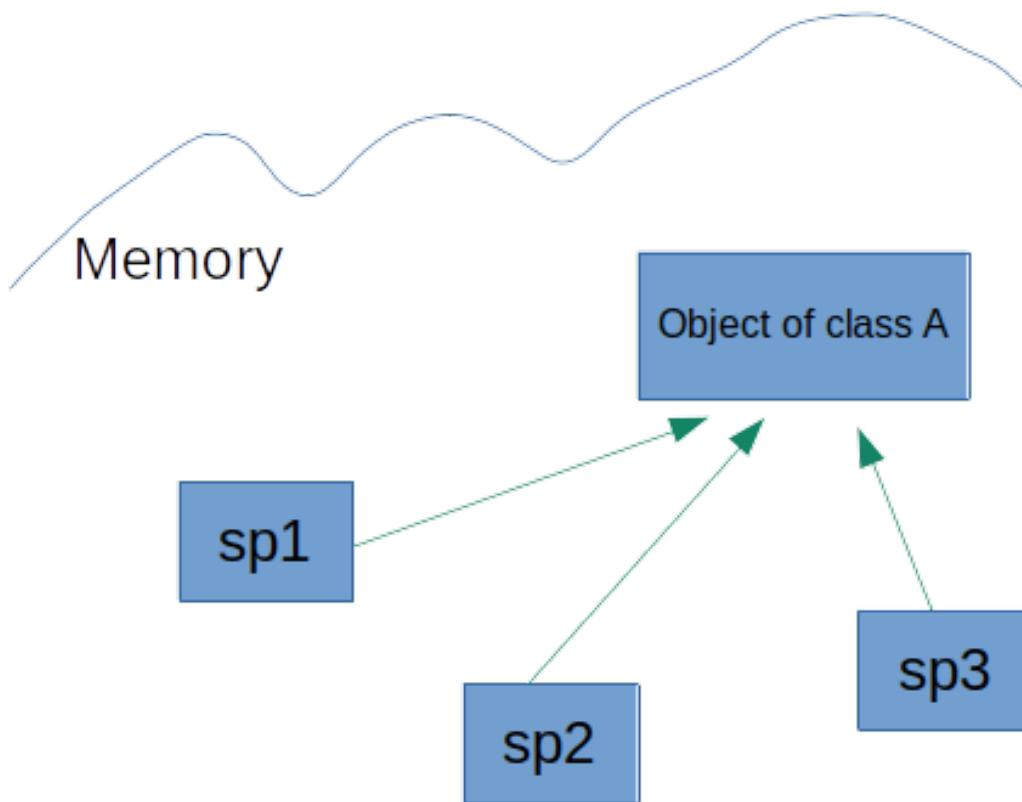


Figura 3.4: Shared ptr

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.

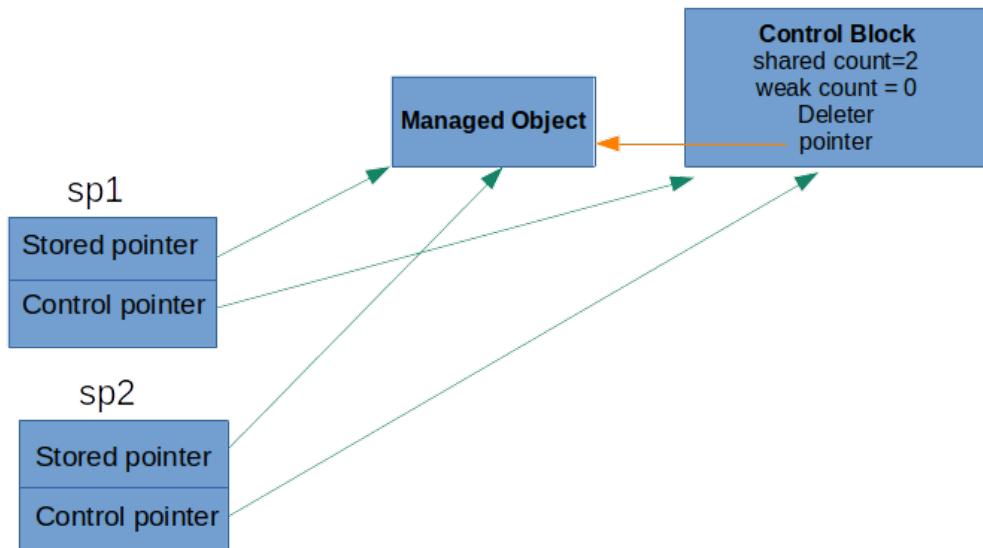


Figura 3.5: Shared ptr structure

### *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. (e quindi quando il *reference counter* arriverà a 0. )

```
#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
}

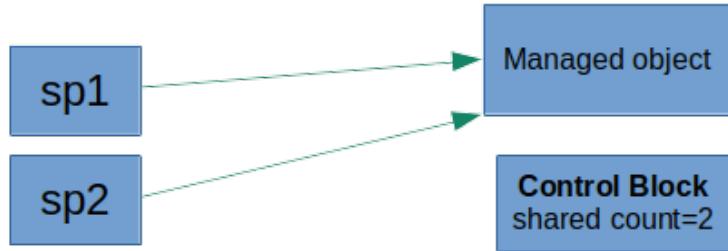
std::cout << shrPtr1.use_count() << std::endl; //
Output: 3

shrPtr3 = std::make_shared<A>(8); // shrPtr3 punta ad
un altro oggetto.x

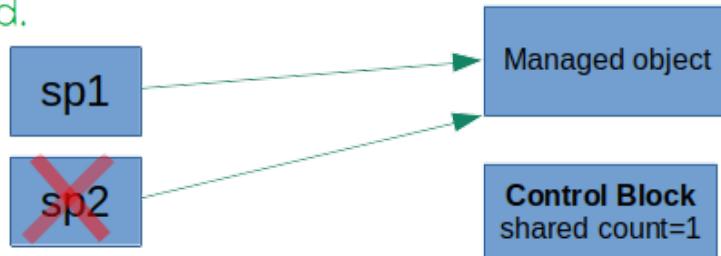
shrPtr2.reset(); // shrPtr2 scollegato e svuotato.

std::cout << shrPtr1.use_count() << std::endl; //
Output: 1
return 0;
}
```

Initial state.



Sp2 destructed.



Sp1 destructed.

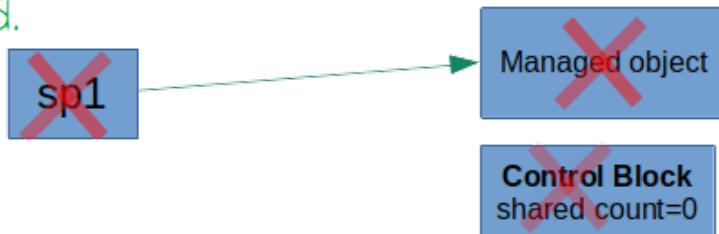


Figura 3.6: Shared ptr destruction

### *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 a 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.

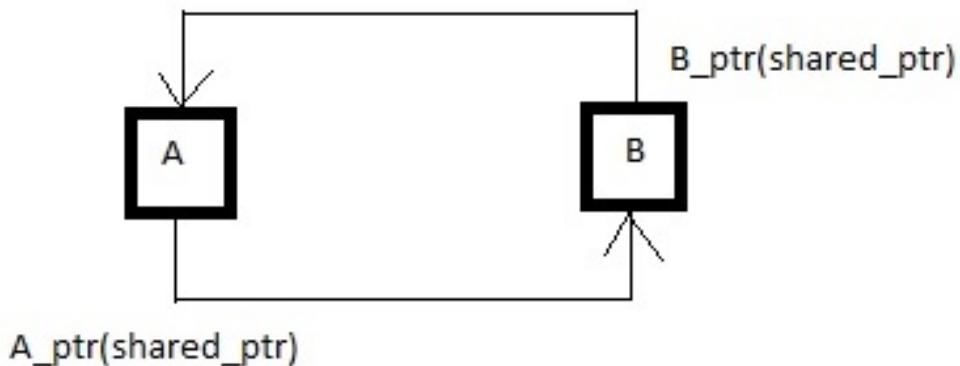
## *weak pointers*

**Definizione:** Un **weak pointer** è un tipo di **smart pointer** 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 creerebbe 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.

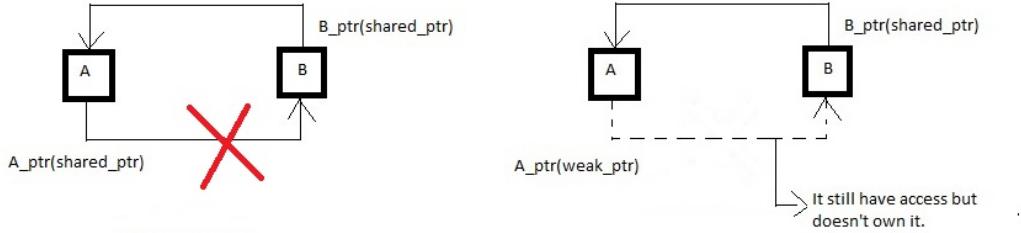


Figura 3.9: Cyclic dependency

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** nell' *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;
}
```

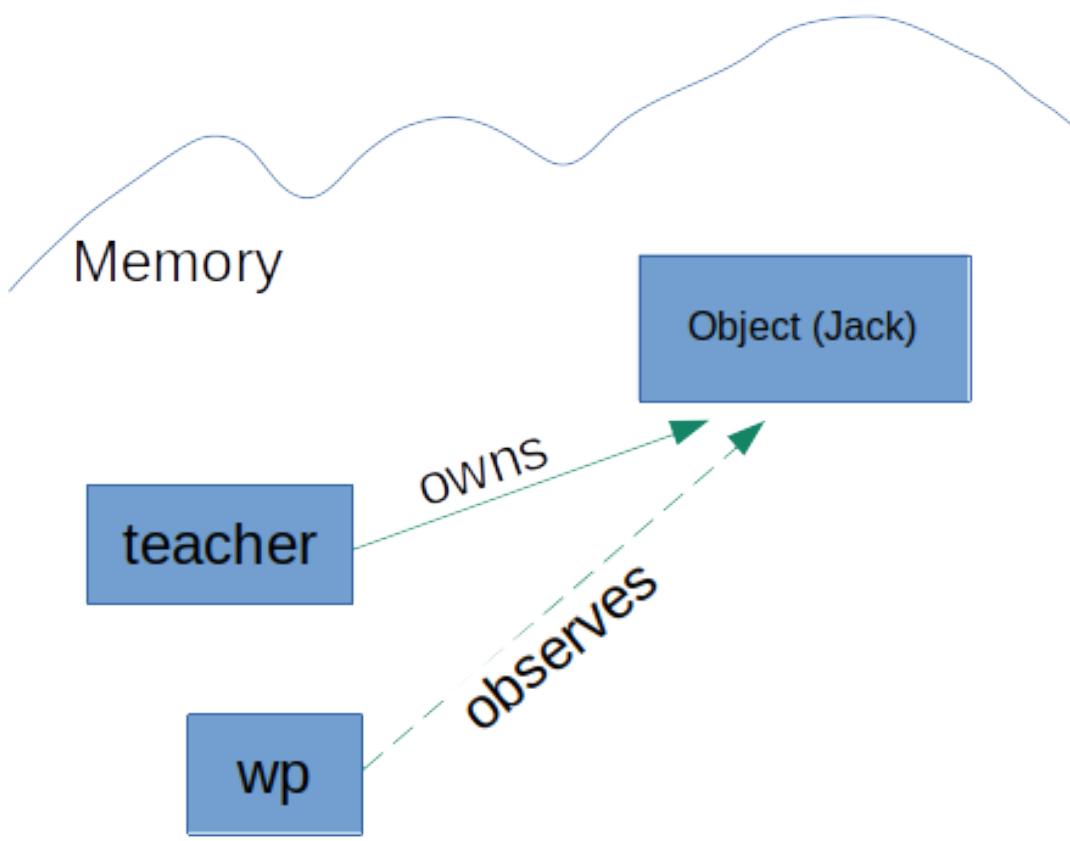


Figura 3.10: Weak pointers

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.

---

### *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 programmati per cercare di correggere gli errori, i bugs di un programma.

Una metodologia di base è questa:

- Osservare il problema.
- Identificare la natura del problema.
- Identificare la condizione (o condizioni) che innesca il problema.
- Identificare il meccanismo di guasto, fallimento.
- Fixare 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

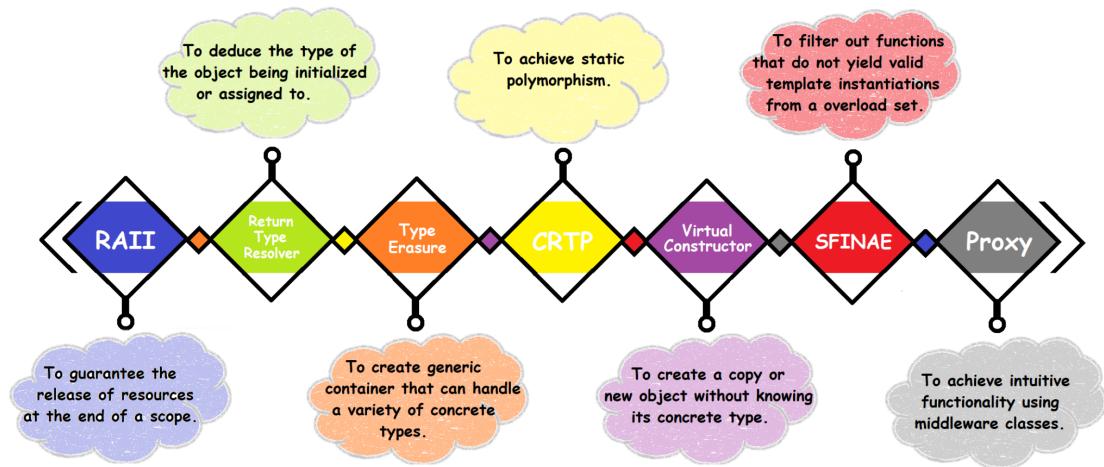


Figura 3.11: 7 Concetti Avanzati

### 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).
    // ...
}

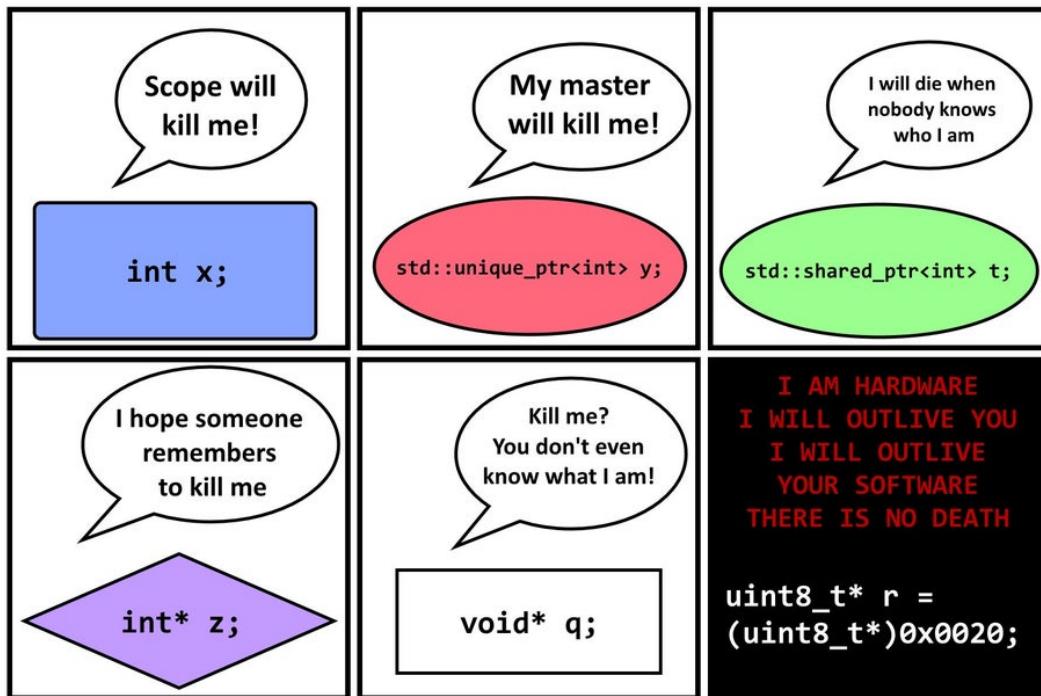
int main()
{
    func();
    return 0;
}

```

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 RAII

## 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 linguaggi, 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’istanziazione 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
// castare con il dynamic_cast.
void call(Base* ptr)
{
    ptr->method();
}
```

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

### *Type erasure usando le union*

```
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
void*	Yes	No	Yes	No
OO	No	Yes	Yes	Yes
Templates	Yes	Yes	No	No

Figura 3.13: Type Erasure

## CRT<sup>P</sup>

**Definizione:** Il CRTP (*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));
}

/* Lo stesso vale per gli altri operatori
== :: 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;
}

```

CRTP 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>&lt;=&gt;</code>
Secondario: <code>!=</code>	<code>&lt;, &gt;, &lt;=, &gt;=</code>

$(A <=> B) < 0$  // è vera se  $A < B$ .

$(A <=> B) > 0$  // è vera se  $A > B$ .

$(A <=> B) == 0$  // è vera se  $A$  e  $B$  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 polimorfa 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 falliscono, 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 voler 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 (frammento di codice) sopra è un esempio di come sfruttare **SFINAE** attraverso l'uso di `std::enable_if`, nella quale le 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 SFINAE avresti un errore dal compilatore, qualcosa del tipo: «*O cannot be converted to member pointer for a non-class type int*» visto che l’*overload* di `test` differisce solo del tipo di ritorno e visto che `int` non è una classe non può avere membro puntatore di tipo `int int::*`.

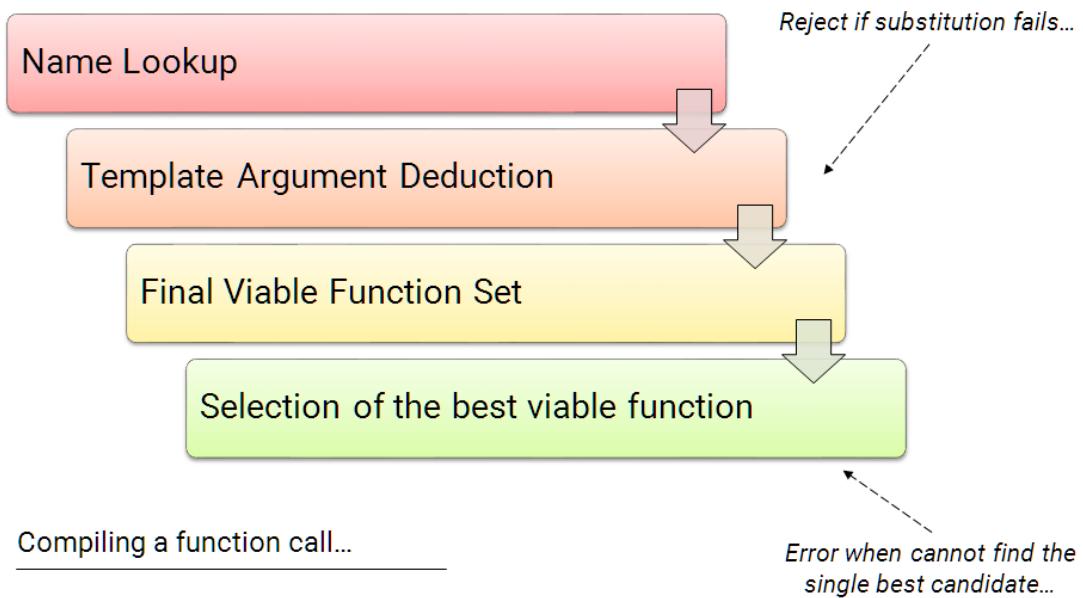


Figura 3.14: SFNAE

SFINAE è stato praticamente rimpiazzato dai **Concepts** del C++20 che fanno la stessa cosa, ma molto meglio. Guardare pag.354 del capitolo *Le gemme degli Algoritmi*.

## 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 intermediario 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.

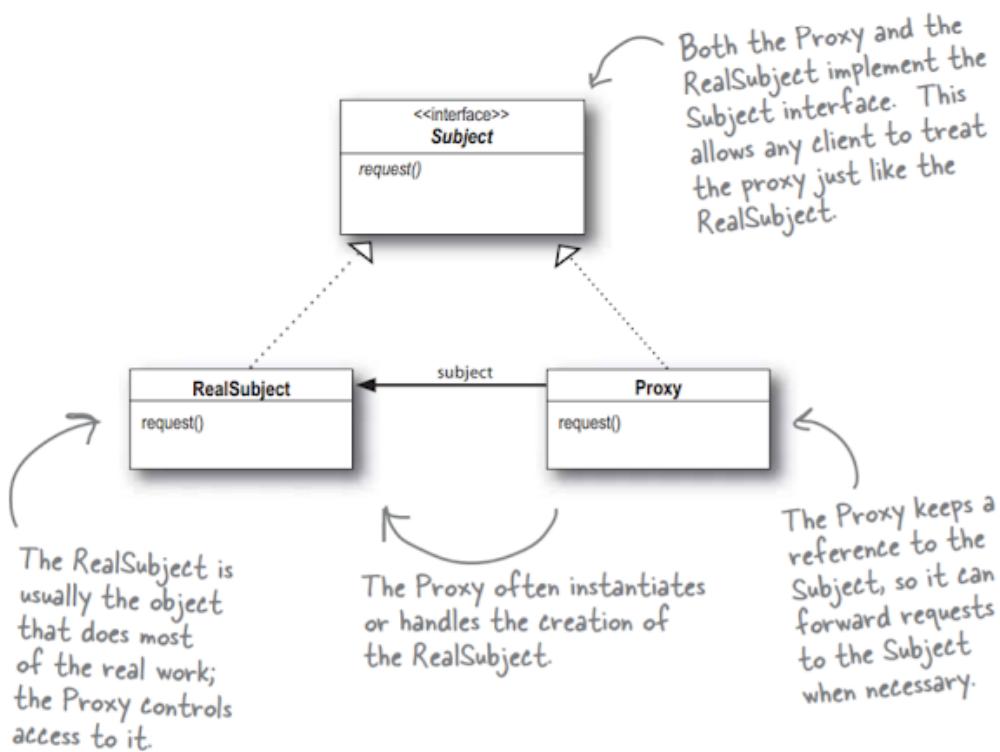


Figura 3.15: Proxy Pattern



# Le gemme degli Algoritmi

4

---

## *Introduzione*

---

È di vitale importanza, per ogni buon programma, sapere e conoscere bene la **Libreria degli Algoritmi**, fornisce delle vere e proprie gemme, per una varietà di scopi: ricerche, ordinamento, contare, fondere, permutare, partizionare, comparare, manipolare su *ranges* ( un range è una sequenza di oggetti a cui si può accedere attraverso iteratori o puntatori) di elementi e molto altro ancora.

Questa libreria è molto vasta, perciò non tratterò proprio tutto tutto, ma una buona parte di essa.

Inoltre, tratterò anche argomenti al di fuori della libreria, ma che legano con essa e sono molto d'aiuto.

Qui una lista di argomenti della libreria degli Algoritmi che tratteremo:

- Operazioni su sequenze non modificabili
- Operazioni su sequenze modificabili
- Operazioni di partizionamento
- Operazioni di Ordinamento
- Operazioni di ricerca binaria
- Operazioni sugli Insiemi
- Operazioni sugli Heap
- Operazioni di Min/Max

- Operazioni di Comparazione
- Operazioni su permutazioni
- Operazioni numeriche

La maggior parte di queste sono definite nell'header `<algorithm>`, ma alcune anche in `<numeric>`, `<execution>` ed altre.

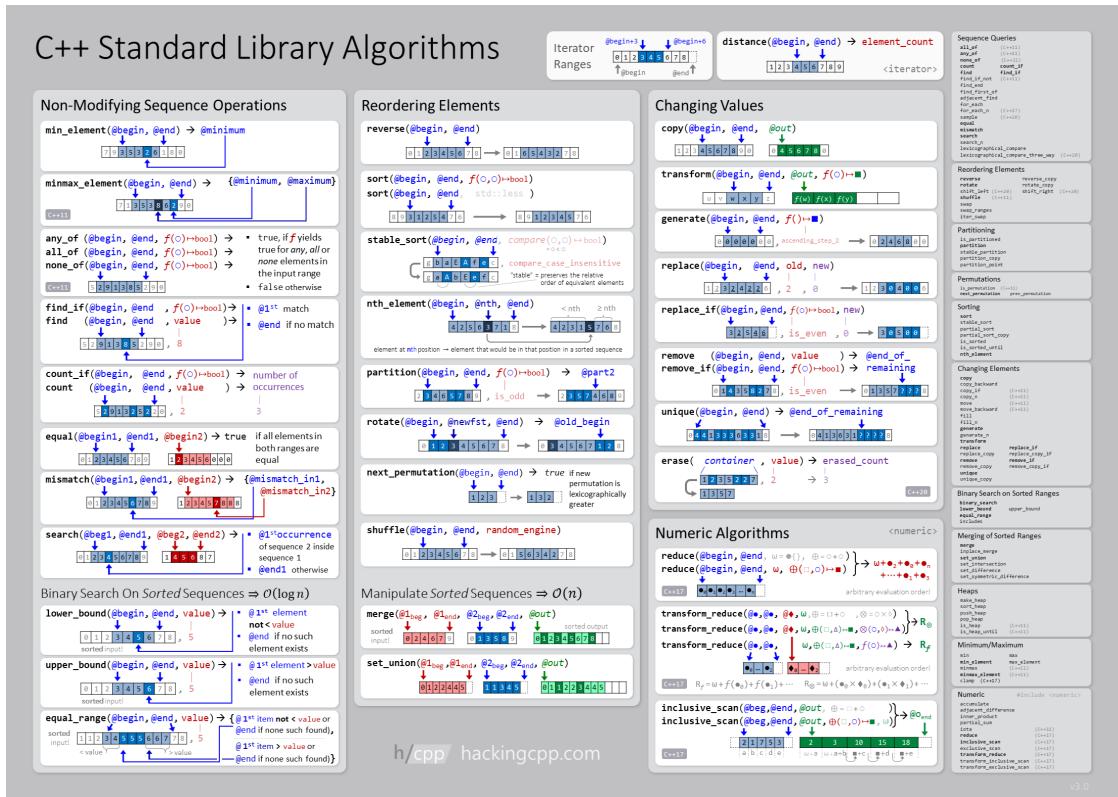


Figura 4.1: Algorithm Library

---

## *Operazioni su sequenze non-modificabili*

---

**Definizione:** Le operazioni su sequenze non-modificabili, da come si intende, sono quelle operazioni che non modificano la sequenza, ma che attuano, compiono ricerche per trovare determinati elementi, contano gli elementi, testano varie condizioni, eccetera.

### *Condizioni*

Possiamo testare se degli elementi sono presenti o no attraverso queste funzioni:  
`std::all_of`, `std::any_of`, `std::none_of`.

- **any\_of**: ha bisogno che anche solo 1 sia vero (sia presente).
- **all\_of**: ha bisogno che tutti quelli considerati siano veri.
- **none\_of**: ha bisogno che nessuno sia presente (tra quelli cercati) (che siano tutti falsi).

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

int main()
{
    std::vector<int> a = { 6, 1, 7, 3, 2, 5, 4, 9, 12
                          };
    std::vector<int> b = { 1, 4, 5, 8, 21, 11, 7, 0,
                          17 };

    // Ci assicuriamo che ci sia almeno un valore minore o uguale a 3
    // nel vettore a.
    std::cout << std::boolalpha << std::any_of( a.
        cbegin(), a.cend(), [](auto n) { return n <= 3;
    }) << "\n"; //Output: true
```

```

// Ci assicuriamo che non ci siano valori maggiori di 33 nel vettore
// b.
std::cout << std::boolalpha << std::none_of( b.
    cbegin(), b.cend(), []( auto n) { return n > 33;
}) << "\n"; //Output: true

std::vector<int> c = {0, 2, 4, 6, 8, 10};

// Controlliamo se tutti i valori sono pari.
if( std::all_of( c.begin(), c.end(), []( int i) {
    return i % 2 == 0;)))
{
    std::cout << "Tutti i numeri sono pari" << "\n"
    ";
}

//Output: Tutti i numeri sono pari.

return 0;
}

```

## Ricerca

Queste operazioni servono per cercare degli elementi all'interno delle sequenze:

`std::find, std::find_if, std::find_if_not, find_end, find_first_of, adjacent_find.`

Inoltre ci sono anche: `std::search, search_n.`

```

#include <iostream>
#include <string>
#include <algorithm>
#include <vector>

int main()
{
    // ESEMPIO FIND
    std::vector<std::string> a = { "zero", "one", "two",
        "three", "four", "five", "six", "seven",
        "eight", "nine", "ten" };
    std::vector<std::string> b = { "0", "1", "2", "3",
        "4", "5", "6", "7", "8", "9", "10" };

```

```

// rbegin() e rend() servono per invertire (reverse) l'iteratore.
const auto k = std::find( a.rbegin(), a.rend(), "one");
std::cout << "Indice dell'ultimo 'one': " << (a.rend() - k) - 1 << std::endl; //Output: Indice dell'ultimo 'one': 1

// ESEMPIO find_first_of
const std::string s = "one;two,three:four";
const std::string delimiter = ";,:";

const auto i = std::find_first_of( s.cbegin(), s.cend(), delimiter.cbegin(), delimiter.cend());
std::cout << "Indice del primo delimitatore: " << i - s.cbegin() << std::endl; //Output: Indice del primo delimitatore: 3

// ESEMPIO adjacent_find
const std::string haystack = "as55jsdjflkadfkjsadlfs5j";
const std::string needle = "s5j";

const auto j = std::adjacent_find( haystack.cbegin(), haystack.cend() );
std::cout << j - haystack.cbegin() << std::endl;
//Output: 2

// ESEMPIO SEARCH
// Cerchiamo una sequenza nella stringa.
const auto w = std::search( haystack.begin(), haystack.end(), needle.begin(), needle.end() );
std::cout << w - haystack.begin() << std::endl; // Output: 21
return 0;
}

```

Questi sono alcuni esempi dell'utilizzo di questi funzioni, non li faccio tutti, ma gli altri sono intuitivi.

### *find vs search*

La differenza è che **find** cerca un singolo elemento nella sequenza, mentre **search** cerca per un'intera sequenza nella sequenza.

## *Contatori*

Ci sono un paio di funzioni per contare gli elementi di una sequenza: **std::count**, **std::count\_if**.

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

int main()
{
    std :: vector<int> v = { 8, 4, 9, 2, 3, 6, 5, 5, 1,
                           2, 4, 9, 1, 2};

    std :: cout << std :: count( v.begin(), v.end(), 3 ) <<
        std :: endl; //Output: 1

    std :: cout << std :: count_if( v.begin(), v.end(),
        [] (auto n) { return n <= 7; } ) << std :: endl; // Output: 11
    return 0;
}
```

## *Altre operazioni*

Ulteriori operazioni possibili sono:

- **mismatch**: restituisce la prima posizione in cui due sequenze differiscono.
- **equal**: per controllare se due sequenze sono uguali (lo tratterò anche nelle *Operazioni di Comparazione*).
- **is\_permutation**: testa se la sequenza è una permutazione (lo tratterò anche nelle *Operazioni di Permutazione*).

```

#include <iostream>
#include <algorithm>
#include <vector>

int main()
{
    // ESEMPIO MISMATCH
    std::vector<std::string> a = { "0", "1", "2", "3",
        "4", "5", "6", "7", "8", "9", "10" };
    std::vector<std::string> b = { "0", "1", "2", "3",
        "4", "&", "6", "7", "8", "9", "10" };

    const auto i = std::mismatch( a.cbegin(), a.cend(),
        b.cbegin()).first;
    std::cout << i - a.cbegin() << std::endl; //Output:
    5

    // ESEMPIO EQUAL
    std::vector<int> v1 = { 1, 2, 3, 4, 5, 6 };
    std::vector<int> v2 = { 1, 2, 3, 4, 5, 6 };
    std::vector<int> v3 = { 1, 2, 4, 3, 5, 6 };

    std::cout << std::boolalpha << std::equal( v1.
        cbegin(), v1.cend(), v2.cbegin()) << std::endl;
    //Output: true
    std::cout << std::boolalpha << std::equal( v1.
        cbegin(), v1.cend(), v3.cbegin()) << std::endl;
    //Output: false

    // ESEMPIO IS_PERMUTATION
    std::vector<int> vec = { 1, 2, 3, 4 };
    std::vector<int> vec2 = { 2, 3, 4, 1 };
    std::vector<int> vec3 = { 2, 3, 2, 2 };

    std::cout << std::boolalpha << std::is_permutation
        (vec.begin(), vec.end(), vec2.begin()) << std::
        endl; //Output: true
    std::cout << std::boolalpha << std::is_permutation
        (vec.begin(), vec.end(), vec3.begin()) << std::
        endl; //Output: false

```

```
    return 0;  
}
```



Per tutte queste operazioni c'è un equivalente per *ranges* del C++20 a pag. 334

---

## *Operazioni su sequenze modificabili*

---

**Definizione:** Queste, invece sono quelle operazioni che ti permettono di modificare la sequenza originaria.

### *Copiare sequenze | Copy*

Queste operazioni ti permettono di copiare parti o intere sequenze: `std::copy`, `std::copy_n`, `std::copy_if`, `std::copy_backward`.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
#include <iterator> // per usare gli iteratori nei loops.
#include <cctype> // per usare std::isupper.

int main()
{
    // ESEMPIO COPY_N
    std::vector<std::string> b = { "0", "1", "2", "3",
        "4", "5", "6", "7", "8", "9", "10" };

    std::vector<std::string> c;

    c.resize(9);

    std::copy_n( b.begin(), 9, c.begin() );

    // In questo caso devo usare begin() ed end(), non posso usare
    // cbegin() e cend().
    std::cout << "il vettore c contiene: ";
    for (std::vector<std::string>::iterator it = c.
        begin(); it!=c.end(); ++it)
    {
        std::cout << ' ' << *it;
    }
}
```

```

std :: cout << '\n' ;

//Output: il vettore c contiene: 0 1 2 3 4 5 6 7 8

// ESEMPIO COPY_N 2
std :: string in = "1234567890";
std :: string out;

std :: copy_n(in.begin(), 4, std :: back_inserter(out));
std :: cout << out << '\n'; //Output: 1234

// ESEMPIO COPY_IF
std :: string a = "Solo Le Lettere in Maiuscolo
Verranno Considerate";

std :: string uppers;

// back_inserter è uno speciale tipo di output iterator per
// permettere agli algoritmi che sovrascrivono gli elementi, come il
// copy, di inserire dei nuovi elementi automaticamente alla fine del
// container.
std :: copy_if(a.begin(), a.end(), std :: back_inserter( uppers ), []( auto s){ return std
:: isupper(s); } ); // una volta funzionava anche così:
std::copy_if(a.begin(), a.end(), std::back_inserter( uppers ),
std::isupper);
std :: cout << uppers << std :: endl; //Output: SLLMVC

// ESEMPIO COPY_BACKWARD
std :: vector<int> inVector;
for( int i = 0; i < 10; i++)
{
    inVector.push_back(i);
}

std :: vector<int> outVector(15);

```

```

    std::copy_backward(inVector.begin(), inVector.end()
                      (), outVector.end()));

    std::cout << "outVector contiene: ";
    for(auto v : outVector){
        std::cout << v << " ";
    }

//Output: outVector contiene: 0 0 0 0 0 1 2 3 4 5 6 7 8 9
    return 0;
}

```

## **Muovere | Move**

Queste ci permettono di spostare gli elementi da una sequenza ad un'altra: `std::move`, `std::move_backward`.

```

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

int main()
{
    std::vector<std::string> a = { "zero", "one", "two"
        , "three", "four", "five", "six", "seven",
        "eight", "nine", "ten" };
    std::vector<std::string> b = { "0", "1", "2", "3",
        "4", "&", "6", "7", "8", "9", "10" };

// Sposta i primi due parametri di move nell'inizio del suo terzo
// parametro.
    std::move( a.begin(), a.begin() + 3, b.begin());
// anche mettendo a.begin() funziona.

    std::cout << "il vettore a contiene: ";
    for(auto v : a){
        std::cout << v << " ";
    }
}

```

```
//Output: Il vettore a contiene: three four five six seven eight nine
ten

    return 0;
}
```

## *Scambiare | Swap*

Le operazioni di **swap** ci permettono di scambiare gli elementi di due sequenze, contenitore: **std::swap**, **std::swap\_ranges**, **iter\_swap**.

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

int main()
{
    // ESEMPIO SWAP
    int x = 5, int y = 12;

    std :: cout << "x prima dello swap: " << x << ", y
    prima dello swap: " << y << std :: endl; //Output:
    x prima dello swap: 5, y prima dello swap: 12

    std :: swap(x,y);

    std :: cout << "x dopo lo swap: " << x << ", y dopo
    lo swap: " << y << std :: endl; //Output: x dopo lo
    swap: 12, y dopo lo swap: 5

    //ESEMPIO SWAP_RANGES
    std :: vector<int> vec1(7, 66);
    std :: vector<int> vec2(7, 18);

    std :: swap_ranges(vec1.begin() + 1, vec1.end() - 1,
                      vec2.begin());

    std :: cout << "vec1 contiene: ";
    for(auto v : vec1){
        std :: cout << v << " ";
    }
}
```

```

//Output: vec1 contiene: 66 18 18 18 18 18 18 66

    std :: cout << '\n';

    std :: cout << "vec2 contiene: ";
    for( auto v : vec2){
        std :: cout << v << " ";
    }

//Output: vec2 contiene: 66 66 66 66 66 18 18

    std :: cout << '\n';
    return 0;
}

```

## ***Trasformare | Transform***

Applica un'operazione sugli elementi delle sequenze: `std::transform`.

**In Place** : vuol dire che il risultato viene messo nello stesso contenitore e non in un altro a parte.

```

#include <iostream>
#include <algorithm>
#include <vector>
#include <cctype>

int main()
{
    // ESEMPIO 1
    std :: vector<int> a = { 5, 7, 8, 9, 1, 2};
    std :: vector<int> b = { 3, 6, 2, 1, 0, 9};

    std :: vector<int> c;

    std :: transform( a.begin(), a.end(), b.begin(), std
        :: back_inserter(c), [] (int a, int b) { return a
        + b * b; });

    std :: cout << "c contiene: ";
    for( auto v : c)

```

```

    {
        std :: cout << v << " ";
    }

    std :: cout << '\n';

    //Output: c contiene: 14 43 12 10 1 83

    //ESEMPIO 2
    std :: string s = "Questa frase verrà trasformata"
                      ;
    std :: string out;

    std :: transform( s.begin(), s.end(), std ::
                     back_inserter(out), [] (auto o){ return std ::
                     toupper(o); });

    std :: cout << out << '\n';

    //Output: QUESTA FRASE VERRÀ TRASFORMATA
    return 0;
}

```

## *Rimpiazzare | Replace*

Queste permettono di rimpiazzare alcuni elementi della sequenza con altri:

std::replace, std::replace\_if, std::replace\_copy, std::replace\_copy\_if.

```

#include <iostream> // per std::cout
#include <algorithm> // per std::replace, replace_if,
                    replace_copy_if
#include <vector> // per std::vector
#include <array> // per std::array
#include <iterator> // per std::ostream_iterator
#include <functional> // per std::bind

int main()
{
    // ESEMPIO REPLACE
    std :: array<int, 10> arr{3, 2, 1, 7, 8, 6, 11, 9,
                                0, 33};

```

```

std::replace(arr.begin(), arr.end(), 6, 66);

for (int a : arr) {
    std::cout << a << " ";
}
std::cout << '\n';

//Output: 3 2 1 7 8 66 11 9 0 33

// ESEMPIO REPLACE_IF
// Se è minore di 3 allora lo sostituiamo con 37.
std::replace_if(arr.begin(), arr.end(),
    std::bind(std::less<int>(), std::placeholders::_1,
              3), 37);
for (int a : arr) {
    std::cout << a << " ";
}
std::cout << '\n';

//Output: 3 37 37 7 8 66 11 9 37 33

// ESEMPIO REPLACE_COPY_IF
std::vector<int> v{6, 1, 22, 66, 3, 9, 8, 1, 4, 5,
                   7, 0};
std::replace_copy_if(v.begin(), v.end(), std::
    ostream_iterator<int>(std::cout, " "), [](int n
        ) { return n > 6; }, 33);
std::cout << '\n';

//Output: 6 1 33 33 3 33 33 1 4 5 33 0
return 0;
}

```

## *Riempire | Fill*

Utilizziamo `std::fill` e `std::fill_n` per riempire una sequenza con una serie di elementi.

```

#include <iostream>
#include <algorithm>
#include <vector>

```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

int main()
{
    std::vector<int> v(10, 21);
    v.reserve(10);
    std::fill_n( std::back_inserter( v ), 10, 36);

    for( auto e : v)
    {
        std::cout << e << " ";
    }

    std::cout << '\n';

    //Output: 21 21 21 21 21 21 21 21 21 21 36 36 36 36 36 36 36 36 36
    //          36
    return 0;
}
```

## ***Generatori | Generate***

Servono per generare elementi in base ad una funzione generatrice. Questa è definita dall'utente ed è chiamata in modo successivo per assegnare gli elementi, numeri.

Queste sono: `std::generate`, `std::generate_n`.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>
#include <random>
#include <functional>

int main()
{
    // ESEMPIO 1
    std::vector<int> v;
    std::generate_n( std::back_inserter(v), 8, [ val = 0 ]() mutable {
        const auto old = val;
        val += 6;
    });
}
```

```

        return old;
    });

    for (auto e : v)
    {
        std::cout << e << " ";
    }

    std::cout << '\n';

//Output: 0 6 12 18 24 30 36 42

// ESEMPIO 2
std::vector<int> v2;

std::mt19937 rng( std::random_device{}() );
std::uniform_int_distribution<int> d(0, 20);

std::generate_n( std::back_inserter(v2), 8, std::
    bind(d, rng));

for (auto e : v2)
{
    std::cout << e << " ";
}

std::cout << '\n';

//Output: 9 6 15 19 14 3 13 8 (pseudo-casuale ogni volta)
return 0;
}

```

## *Rimozione | Remove*

Come implica dal nome, queste operazioni forniscono un modo per rimuovere elementi da una sequenza: `std::remove`, `std::remove_if`, `std::remove_copy`, `std::remove_copy_if`.

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

```

#include <string>
#include <cctype>

int main()
{
    std::string str = " Testo con degli spazi
                        bianchi";

    auto noSpace = std::remove(str.begin(), str.end(),
                               ' ');

    std::cout << str << std::endl; //Output:
                                    Testocondegli spazi bianchi bianchi

    std::string str2 = "Testo\n con\tdegli spazi
                        bianchi\n\n";
    str2.erase(std::remove_if(str2.begin(), str2.end(),
                           [](unsigned char c){return std::isspace(c);}),
               str2.end());

    std::cout << str2 << std::endl; //Output:
                                    Testocondegli spazi bianchi
    return 0;
}

```

## *Unico | Unique*

Permettono di ottenere una sequenza unica, senza elementi ripetuti.

Queste operazioni sono: `std::unique`, `std::unique_copy`.

```

#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
#include <iterator>

int main()
{
    // ESEMPIO UNIQUE

```

```

std::vector<std::string> s = { "0", "1", "2", "2",
                               "2", "3", "5", "4", "6", "9", "9", "5", "11"
};

const auto s2 = std::unique(s.begin(), s.end());
s.erase(s2, s.end());

for(auto e : s)
{
    std::cout << e << " ";
}

std::cout << '\n';
//Output: 0 1 2 3 5 4 6 9 5 11

// ESEMPIO UNIQUE_COPY
std::string str1 = "La      stringa      con molti
                     spazi bianchi!";
std::cout << "prima: " << str1 << '\n'; //Output: La
                     stringa con molti spazi bianchi!

std::string str2;
std::unique_copy(str1.begin(), str1.end(), std::
                 back_inserter(str2),
                 [] (char c1, char c2){ return c1 == ' ' && c2 == ' ';
                     });
std::cout << "dopo:   " << str2 << '\n'; //Output: La
                     stringa con molti spazi bianchi!
return 0;
}

```

## *Invertire | Reverse*

Queste operazioni consentono di invertire l'ordine delle sequenze: `std::reverse`,  
`std::reverse_copy`.

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

```

int main()
{
    // ESEMPIO REVERSE
    std::vector<int> vec{ 9, 6, 3};
    std::reverse(vec.begin(), vec.end());
    for(auto v : vec) std::cout << v;
    std::cout << '\n';
    //Output: 369

    // ESEMPIO REVERSE_COPY
    std::vector<int> vec2({2, 1, 4});

    std::vector<int> reverseVec(3);

    std::reverse_copy(std::begin(vec2), std::end(vec2)
                    , std::begin(reverseVec));
    for(auto v : reverseVec)
    {
        std::cout << v << " ";
    }

    std::cout << '\n';
    //Output: 4 1 2
    return 0;
}

```

## *Ruotare | Rotate*

Ruotano l'ordine degli elementi nella sequenza: `std::rotate`, `std::rotate_copy`.

```

#include <iostream>
#include <algorithm>
#include <vector>

int main()
{
    std::vector<int> v{3, 7, 9, 8, 2, 1, 0, 10, 4, 5,
                       12};

    // Rotazione verso sinistra
    std::rotate(v.begin(), v.begin() + 1, v.end());

```

```

for (auto e : v) std::cout << e << " ";
std::cout << '\n';

//Output: 7 9 8 2 1 0 10 4 5 12 3

// Rotazione verso destra
std::rotate(v.rbegin(), v.rbegin() + 1, v.rend());

for (auto e : v) std::cout << e << " ";
std::cout << '\n';

//Output: 3 7 9 8 2 1 0 10 4 5 12
return 0;
}

```

## *Spostare | Shift*

Servono per spostare di tot elementi le sequenze, a differenza della rotazione che si limita a ruotare la sequenza, con lo spostamento si perdono o si ottengono dati:

`std::shift_left, std::shift_right.`

```

#include <iostream>
#include <algorithm>
#include <vector>
#include <string>

int main()
{
    std::vector<std::string> g{"a", "b", "c", "d", "e",
                             "f", "g"};
    std::shift_left( begin(g), end(g), 3 );
    for (auto s : g) std::cout << s << " ";
    std::cout << '\n';

    //Output: . . . d e f g (in realtà mi dava errore std::shift_left non è
    //membro di std)

    std::shift_right( begin(g), end(g), 3 );
}

```

```

    for (auto s : g) std::cout << s << " ";
    std::cout << '\n';

    //Output: . . . d . . . (in realtà mi dava errore std::shift_right non è
    //membro di std)

    return 0;
}

```

## Mischiare | Shuffle

Serve per mischiare, mescolare gli elementi della sequenza: **std::shuffle**, **std::random\_shuffle**.

```

#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
#include <random>

int main()
{
    std::vector<std::string> a = { "zero", "one", "two",
        "three", "four", "five", "six", "seven", "eight",
        "nine", "ten" };

    std::mt19937 rng( std::random_device{}() );
    std::shuffle( a.begin(), a.end(), rng );

    for (auto s : a) std::cout << s << " ";
    std::cout << '\n';

    //Output: three zero four two ten one nine seven six eight five (è
    //sempre diversa perché è casuale)
    return 0;
}

```



Per tutte queste operazioni c'è un equivalente per *ranges* del C++20 a pag. 335

---

## *Operazioni su Partizioni*

---

**Definizione:** Le operazioni su partizioni permettono di eseguire partizioni sulle sequenze di elementi.

Queste operazioni possibili sono: `std::partition`, `std::is_partitioned`, `std::stable_partition`, `std::partition_copy`, `std::partition_point`.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <array>

int main()
{
    // ESEMPIO PARTITION
    std::vector<std::string> a = { "zero", "one", "two",
        "three", "four", "five", "six", "seven",
        "eight", "nine", "ten" };
    std::vector<int> b = {0, 1, 2, 3, 4, 5, 6, 7, 8,
        9, 10};

    auto partition1 = std::partition( a.begin(), a.end(),
        [](>const auto& s) { return std::any_of( s.
            begin(), s.end(), [](>char c){ return c == 'e';
        }) } );

    for( auto p : a)
    {
        std::cout << p << " ";
    }

    std::cout << '\n';

    //Output: zero one ten three nine five eight seven six four two

    auto partition2 = std::partition( b.begin(), b.end(),
        [](>int i) { return i % 2 == 0; } );
}
```

```

for( auto p : b)
{
    std::cout << p << " ";
}

std::cout << '\n';

//Output: 0 10 2 8 4 6 5 7 3 9 1

// ESEMPIO STABLE_PARTITION
std::vector<std::string> c = { "zero", "one", "two",
                                "three", "four", "five", "six", "seven",
                                "eight", "nine", "ten" };
std::vector<int> d = {0, 1, 2, 3, 4, 5, 6, 7, 8,
                      9, 10};

auto partition3 = std::stable_partition( a.begin(),
                                         a.end(), [](&const auto& s) { return std::any_of(
                                         s.begin(), s.end(), [](&char c){ return c ==
                                         'e';}); });

for( auto p : c)
{
    std::cout << p << " ";
}

std::cout << '\n';

//Output: zero one two three four five six seven eight nine ten

auto partition4 = std::stable_partition( b.begin(),
                                         b.end(), [](&int i) { return i % 2 == 0;});

for( auto p : d)
{
    std::cout << p << " ";
}

std::cout << '\n';

```

```
//Output: 0 1 2 3 4 5 6 7 8 9 10

//ESEMPIO IS_PARTITIONED
std::vector<int> v = { 2, 4, 8, 6, 0, 10};
bool isPartitioned = std::is_partitioned( v.begin()
() , v.end() , [] (int i){ return i % 2 == 0; });

std::cout << std::boolalpha << isPartitioned <<
std::endl; //Output: true

// ESEMPIO PARTITION_COPY
int arr [10] = {1,2,3,4,5,6,7,8,9,10};
int trueArr [5] = {0};
int falseArr [5] = {0};

std::partition_copy( std::begin(arr) , std::end(arr)
, std::begin(trueArr) , std::begin(falseArr) ,
[] (int i){return i > 5; });

for( int a : trueArr )
{
    std::cout << a << " ";
}

std::cout << '\n';

//Output: 6 7 8 9 10

for( int a : falseArr )
{
    std::cout << a << " ";
}

std::cout << '\n';

//Output: 1 2 3 4 5

// ESEMPIO PARTITION_POINT
std::array v = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

```
auto is_even = [](int i){ return i % 2 == 0; };
const auto parPoint = std::partition_point(v.
    cbegin(), v.cend(), is_even);
const auto i = std::distance(v.cbegin(), pp);
std::cout << "Partition point a: " << i << "; v["
    << i << "] = " << *pp << '\n'; //Output: Partition
    point a: 4; v[4] = 5
return 0;
}
```



Per tutte queste operazioni c'è un equivalente per *ranges* del C++20 a pag. 337

---

## Operazioni di Ordinamento

---

**Definizione:** Le operazioni di ordinamento consentono di ordinare le sequenze.  
Queste sono: `std::sort`, `std::stable_sort`, `std::partial_sort`, `std::partial_sort_copy`,  
`std::is_sorted`, `std::is_sorted_until`, `std::nth_element`.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>

int main()
{
    // ESEMPIO SORT E IS_SORTED
    std::vector<int> v{ 7, 9, 1, 0, 2, 5, 3, 4, 8, 10,
                       6};

    std::cout << std::boolalpha << std::is_sorted(v.
begin(), v.end()) << '\n'; //Output: false

    std::sort(v.begin(), v.end());

    for (auto e : v)
    {
        std::cout << e << " ";
    }

    std::cout << '\n';

    //Output: 0 1 2 3 4 5 6 7 8 9 10

    std::cout << std::boolalpha << std::is_sorted(v.
begin(), v.end()) << '\n'; //Output: true

    // ESEMPIO SORT
    // Il sort, di solito usa il minore < per fare l'ordinamento, ma in
    // questo caso noi gli passiamo il maggiore >.
```

```

std :: sort(v.begin(), v.end(), std :: greater<int>{})
;

for (auto e : v)
{
    std :: cout << e << " ";
}

std :: cout << '\n';

//Output: 10 9 8 7 6 5 4 3 2 1 0

// ESEMPIO STABLE_SORT
class Employee {
    public:
        int age;
        std :: string name; // Non partecipa nei paragoni
};

// Questo ci serve per indicare alle funzioni sort come devono fare il
// paragone per ordinare gli elementi.
bool operator<(const Employee & lhs, const
               Employee & rhs)
{
    return lhs.age < rhs.age;
}

std :: vector<Employee> employees =
{
    {22, "Gigi"}, 
    {37, "Maria"}, 
    {48, "Alessio"}, 
};

std :: stable_sort(v.begin(), v.end());

for (const Employee & e : employees)
    std :: cout << e.age << ", " << e.name << '\n';

//Output: 22, Gigi

```

```

//Output: 37, Maria
//Output: 48, Alessio

// ESEMPIO PARTIAL_SORT
std::vector<int> v2 = { 5, 7, 1, 0, 3, 5, 6, 4, 7,
                        8, 10, 2 };

// Ordina solo una parte del contenitore, non tutto.
// Può essere utile, magari per delle classifiche, dove magari servono
// soltanto i primi 10 o i primi 100 giocatori e non tutti gli altri.
std::partial_sort(v2.begin(), v2.begin() + 4, v2.end());

for (auto& v : v2)
{
    std::cout << v << " ";
}

std::cout << '\n';

//Output: 0 1 2 3 7 5 6 5 7 8 10 4 (i primi 4 elementi sono ordinati:
// 0 1 2 3, tutti gli altri son rimasti in posizioni non ordinate)

// ESEMPIO NTH_ELEMENT
std::vector<int> v3{11, 7, 9, 4, 2, 2, 6, 7, 9, 3,
                     7, 7};

auto m = v3.begin() + v3.size() / 2;
std::nth_element(v3.begin(), m, v3.end());

for (auto& e : v3)
{
    std::cout << e << " ";
}

std::cout << '\n';

//Output: 6 2 3 4 2 7 7 7 9 9 7 11

std::cout << "The medianIl mediano è: " << v3[v3.

```

```
    size () / 2] << '\n' ; //Output: Il mediano è 7  
    return 0;  
}
```



Per tutte queste operazioni c'è un equivalente per *ranges* del C++20 a pag. 339

---

## *Operazioni di Ricerca Binaria*

---

**Definizione:** Queste servono per operazioni su ricerche binarie: `std::lower_bound`, `std::upper_bound`, `std::equal_range`, `std::binary_search`.

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

int main()
{
    // ESEMPIO LOWER_BOUND
    std::vector<int> v{ 15, 5, 22, 11, 33, 7, 88, 0,
                        9, 1, 4, 26, 52 };

    // Restituisce un iteratore che punta al primo elemento che non è
    // minore del valore.
    const int lowerBound = 3;
    const auto i = std::lower_bound(v.begin(), v.end()
                                    , lowerBound);

    std::cout << *i << '\n'; //Output: 15

    // ESEMPIO UPPER_BOUND
    // Restituisce un iteratore che punta al primo elemento che è
    // maggiore del valore.
    const int upperBound = 7;
    const auto j = std::upper_bound(v.begin(), v.end()
                                    , upperBound);

    std::cout << *j << '\n'; //Output: 22

    // ESEMPIO EQUAL_RANGE
    std::vector<int> v = { 11, 10, 33, 33, 33, 111,
                          10,
                          333, 369, 77, 77, 80 };
```

```

// Dichiariamo un iteratore per il
// valore di ritorno di std::equal_range
std :: pair<std :: vector<int >:: iterator ,
std :: vector<int >:: iterator> it;

std :: sort(v.begin(), v.end());
// v diventa 10 10 11 33 33 33 77 77 80 111 333 369

// Usiamo std::equal_range e compariamo gli elementi
// con 33
it = std :: equal_range(v.begin(), v.begin() + 12,
33);

// Displaying the subrange bounds
std :: cout << "33 è presente nel vettore dall' indice: "
<< (it . first - v.begin()) << " fino a: "
<< (it . second - v.begin());

//Output: 33 è presente nel vettore dall'indice: 3 fino a: 6

// ESEMPIO BINARY_SEARCH
std :: vector<int > haystack {1, 3, 7, 5, 9};
std :: vector<int > needles {7, 2, 3};

for (auto needle : needles) {
    std :: cout << "Cerco il: " << needle << '\n';
    if (std :: binary_search(haystack.begin(),
haystack.end(), needle)) {
        std :: cout << "Trovato " << needle << '\n';
    } else {
        std :: cout << "Non trovato!\n";
    }
}

//Output: Cerco il: 7
//Output: Trovato
//Output: Cerco il: 2
//Output: Non trovato!
//Output: Cerco il: 3

```

```
//Output: Trovato  
    return 0;  
}
```



Per tutte queste operazioni c'è un equivalente per *ranges* del C++20 a pag. 340

---

## *Operazioni di Fusione | Merge*

---

**Definizione:** Le operazioni di merge permettono di combinare, di fondere gli elementi di due sequenze: `std::merge`, `std::inplace_merge`.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <random>
#include <functional>

int main()
{
    // ESEMPIO MERGE
    std::random_device rd;
    std::mt19937 mt(rd());
    std::uniform_int_distribution<> distribution(0, 9)
        ;

    std::vector<int> v1(10), v2(10);
    std::generate(v1.begin(), v1.end(), std::bind(
        distribution, std::ref(mt)));
    std::generate(v2.begin(), v2.end(), std::bind(
        distribution, std::ref(mt)));

    std::cout << "Prima del sorting" << '\n';

    for (auto& v : v1)
    {
        std::cout << v << " ";
    }
    std::cout << '\n';

    for (auto& v : v2)
    {
        std::cout << v << " ";
    }
}
```

```
std :: cout << '\n';

//Output: Prima del sorting
//Output: 5 6 1 8 4 9 2 5 2 2 (numeri generati casualmente)
//Output: 6 0 6 5 7 8 8 2 8 5 (numeri generati casualmente)

std :: sort(v1.begin(), v1.end());
std :: sort(v2.begin(), v2.end());

std :: cout << "Dopo il sorting" << '\n';

for (auto& v : v1)
{
    std :: cout << v << " ";
}
std :: cout << '\n';

for (auto& v : v2)
{
    std :: cout << v << " ";
}
std :: cout << '\n';

//Output: Dopo il sorting
//Output: 1 2 2 2 4 5 5 6 8 9
//Output: 0 2 5 5 6 6 7 8 8 8

std :: vector<int> v3;
std :: merge(v1.begin(), v1.end(), v2.begin(), v2.end(), std :: back_inserter(v3));

std :: cout << "v3 contiene: ";

for (auto& v : v3)
{
    std :: cout << v << " ";
}
std :: cout << '\n';
```

```

//Output: v3 contiene: 0 1 2 2 2 2 4 5 5 5 5 6 6 6 7 8 8 8 9 (numeri
    generati casualmente)

// ESEMPIO INPLACE_MERGE
int first[] = {7, 36, 12, 24, 8};
int second[] = {57, 94, 14, 67, 80};

std::vector<int> v(10);

std::vector<int>::iterator it;

std::sort(first, first+5);
std::sort(second, second+5);

it = std::copy(first, first+5, v.begin());
std::copy(second, second+5, it);

std::inplace_merge(v.begin(), v.begin() + 5, v.end());
;

std::cout << "Il vettore risultante contiene: ";
for (it=v.begin(); it!=v.end(); ++it)
{
    std::cout << ' ' << *it;
}
std::cout << '\n';

//Output: Il vettore risultante contiene: 7 8 12 14 24 36 57 67 80 94
return 0;
}

```



Per tutte queste operazioni c'è un equivalente per *ranges* del C++20 a pag. 341

---

## *Operazioni sugli Insiemi*

---

**Definizione:** Le operazioni sugli insiemi permettono di attuare le tipiche operazioni matematiche: *unione* : std::set\_union, *intersezione* : std::set\_intersection, *differenza* : std::set\_difference, *simmetria differenza* : std::set\_symmetric\_difference.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>

// Funzione per mostrare gli elementi di un std::vector
template<typename T>
void printVec(const std::vector<T>& v)
{
    for (const auto& i : v)
    {
        std::cout << i << " ";
    }
    std::cout << '\n';
}

int main()
{
    // ESEMPIO INCLUDES
    std::vector<char> v1 = { 'a', 'b', 'c', 'd', 'f',
                            'g' };
    std::vector<char> v2 = { 'a', 'b', 'c' };
    std::vector<char> v3 = { 'a', 'c' };

    printVec(v1);
    std::cout << "include: " << '\n';

    printVec(v2);

    std::cout << std::boolalpha << std::includes(v1.
begin(), v1.end(), v2.begin(), v2.end()) << '\n'

```

```

'; //Output: true

printVec(v3);

std::cout << std::boolalpha << std::includes(v1.
begin(), v1.end(), v3.begin(), v3.end()) << '\n
'; //Output: false

std::cout << std::flush;

// ESEMPIO SET_UNION, SET_INTERSECTION,
SET_DIFFERENCE, SET_SYMMETRIC_DIFFERENCE
std::vector<int> vec1{ 1, 2, 3, 5, 7, 9 };
std::vector<int> vec2{ 6, 4, 1, 2, 3, 9, 11 };

std::sort(vec1.begin(), vec1.end());
std::sort(vec2.begin(), vec2.end());

std::vector<int> destUnion;

std::set_union(vec1.begin(), vec1.end(), vec2.
begin(), vec2.end(), std::back_inserter(
destUnion));

std::cout << "unione: ";
printVec(destUnion); //Output: unione: 1 2 3 4 5 6 7 9 11

std::vector<int> destIntersection;

std::set_intersection(vec1.begin(), vec1.end(),
vec2.begin(), vec2.end(), std::back_inserter(
destIntersection));

std::cout << "intersezione: ";
printVec(destIntersection); //Output: intersezione: 1 2 3
                           9

std::vector<int> destDifference;

```

```
    std::set_difference(vec1.begin(), vec1.end(), vec2
        .begin(), vec2.end(), std::back_inserter(
        destDifference));  
  
    std::cout << "differenza: ";
    printVec(destDifference); //Output: differenza: 5 7  
  
    std::vector<int> destSymmDifference;  
  
    std::set_symmetric_difference(vec1.begin(), vec1.
        end(), vec2.begin(), vec2.end(), std::back_inserter(
        destSymmDifference));  
  
    std::cout << "differenza simmetrica: ";
    printVec(destSymmDifference); //Output: differenza
        simmetrica: 4 5 6 7 11  
  
    return 0;
}
```



Per tutte queste operazioni c'è un equivalente per *ranges* del C++20 a pag. 342

## *Operazioni su Heap*

---

**Definizione:** L'Heap è una struttura dati basata sugli alberi che soddisfa la proprietà degli heap:

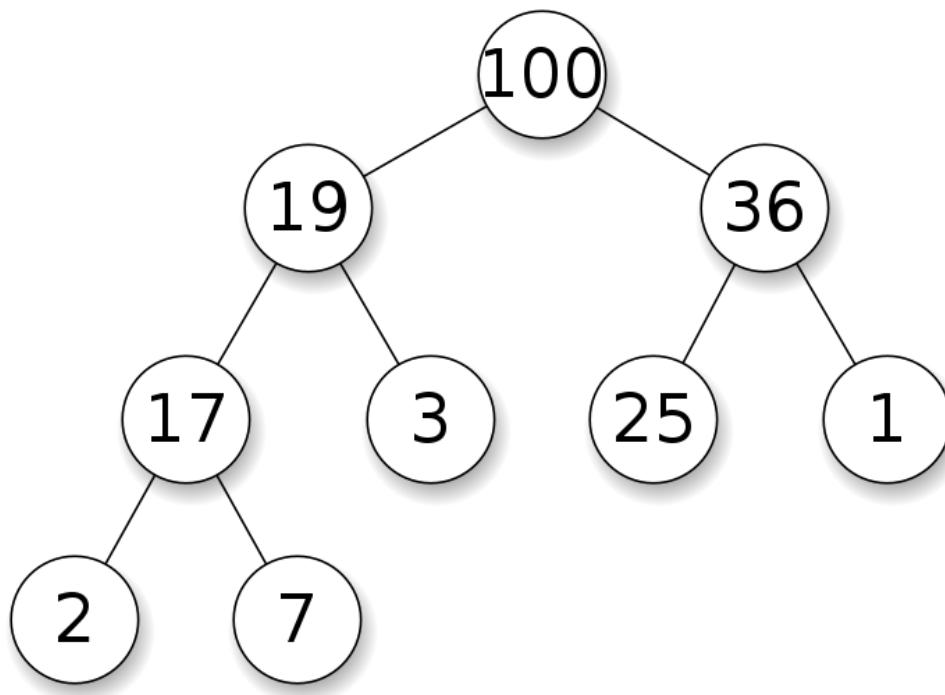
In un **max heap**:

- le chiavi (il valore) di ciascun nodo padre sono sempre maggiori o uguali a quelle dei nodi figli.
- La chiave (il valore) del valore massimo appartiene alla radice (detta *root node*).

In un **min heap**:

- Le chiavi (il valore) di ciascun nodo sono minori o uguali a quelle dei figli.
- La chiave del valore minimo appartiene alla radice.

## Tree representation



## Array representation

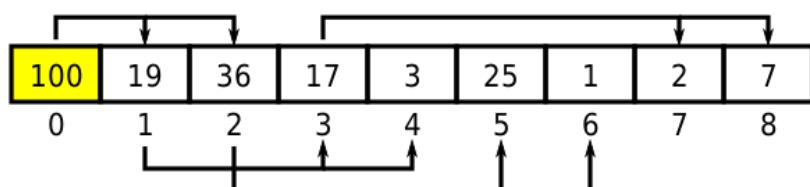


Figura 4.2: Max Heap

Le operazioni disponibili sugli heap sono: `std::push_heap`, `std::pop_heap`, `std::make_heap`, `std::sort_heap`, `std::is_heap`, `std::is_heap_until`.

```
#include <iostream>
```

```

#include <algorithm>
#include <vector>
#include <random>
#include <functional>

// Funzione per mostrare gli elementi di un std::vector
template<typename T>
void printVec(const std::vector<T>& v)
{
    for(const auto& i : v)
    {
        std::cout << i << " ";
    }
    std::cout << '\n';
}

int main()
{
    // ESEMPIO MAKE_HEAP, SORT_HEAP
    std::mt19937 rng{ std::random_device{}() };
    std::uniform_real_distribution<float> dist{0.0f,
                                                99.0f};
    auto rnd = std::bind(dist, rng);

    std::vector<int> v( 22 );
    std::generate_n( v.begin(), v.size(), rnd );

    std::make_heap( v.begin(), v.end()); // se qui avessi
                                         messo std::make_heap( v.begin(), v.end(), std::greater<int> );
                                         avremmo fatto un min-heap
    std::sort_heap( v.begin(), v.end());

    printVec(v); //Output: 2 2 8 15 23 23 30 38 52 53 64 64 64
                  66
                  68 71 73 74 77 78 87 92

    // ESEMPIO IS_HEAP, MAKE_HEAP

    std::vector<int> v2 = { 7, 12, 8, 9, 18, 36, 44,
                           52, 90, 11, 5};

```

```

printVec(v2); //Output: 7 12 8 9 18 36 44 52 90 11 5

std::cout << std::boolalpha << "is_heap: " << std
::is_heap(v2.begin(), v2.end()) << '\n'; //
Output: is_heap: false

if (!std::is_heap(v2.begin(), v2.end()))
{
    std::make_heap(v2.begin(), v2.end(), std::
        greater{}); // Col std::greater faccio un min heap.
}

printVec(v2); //Output: 5 7 8 9 11 36 44 52 90 12 18

std::cout << std::boolalpha << "is_heap: " << std
::is_heap(v2.begin(), v2.end()) << '\n'; //
Output: is_heap: true

// ESEMPIO PUSH_HEAP, POP_HEAP

std::vector<int> v3 = { 8, 10, 4, 7, 5, 6, 3, 1,
0, 2};

std::make_heap(v3.begin(), v3.end());

v3.push_back(9);

std::cout << "Prima del push_heap: ";
printVec(v3); //Output: Prima del push_heap: 10 8 6 7 5 4 3 1
               0 2 9

std::push_heap(v3.begin(), v3.end());

std::cout << "Dopo il push_heap: ";
printVec(v3); //Output: Dopo il push_heap: 10 9 6 7 8 4 3 1 0
               2 5

std::pop_heap(v3.begin(), v3.end());
std::cout << "Dopo la pop_heap: ";
printVec(v3); //Output: Dopo la pop_heap: 9 8 6 7 5 4 3 1 0 2

```

10 (abbiamo rimosso l'ultimo elemento)

```
// ESEMPIO IS_HEAP_UNTIL
std :: vector<int> v4{2,6,9,3,8,4,5,1,7};

std :: sort(v4.begin(),v4.end());
std :: reverse(v4.begin(),v4.end());

auto last = std :: is_heap_until (v4.begin(),v4.end
());

std :: cout << "I primi " << (last - v4.begin()) <<
    " elementi sono un heap valido: ";
printVec(foo); //Output: I primi 9 elementi sono un heap
valido: 9 8 7 6 5 4 3 2 1

return 0;
}
```

---

## *Operazioni di Min/Max*

---

**Definizione:** Le operazioni di min/max servono per ottenere il minimo o il massimo da una data espressione: `std::max`, `std::min`, `std::minmax`, `std::min_element`, `std::max_element`, `std::minmax_element`, `std::clamp`.

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

int main()
{
    // ESEMPIO MIN, MAX
    const int a = 25;
    const int b = 82;

    std::cout << "Elemento minore: " << std::min(a, b)
        << std::endl; //Output: 25
    std::cout << "Elemento maggiore: " << std::min(a,
        b) << std::endl; //Output: 82

    // ESEMPIO MINMAX
    int x = 18;
    int y = 22;

    auto mm = std::minmax( x, y );

    std::cout << mm.first << std::endl; //Output: 18
    std::cout << mm.second << std::endl; //Output: 22

    // Oppure si poteva fare anche:

    auto [min, max] = std::minmax(x, y); // Se usiamo la
        // reference, auto&[min, max] a delle temporanee ci porterà ad un
        // undefined behaviour.

    std::cout << min << std::endl; //Output: 18
```

```

std :: cout << max << std :: endl; //Output: 22

// ESEMPIO MIN_ELEMENT, MAX_ELEMENT,
// MINMAX_ELEMENT
std :: vector<int> v = { 8, 77, 42, 54, 3};

auto minElement = std :: min_element( v.begin(), v.
end() );

std :: cout << "Elemento minore: " << minElement <<
std :: endl; //Output: 3

auto maxElement = std :: max_element( v.begin(), v.
end() );

std :: cout << "Elemento maggiore: " << maxElement
<< std :: endl; //Output: 77

auto minMaxElement = std :: minmax_element(v.begin()
, v.end());

std :: cout << "Elemento minore con minmax: " <<
minMaxElement.first << std :: endl; //Output: 3

std :: cout << "Elemento maggiore con minmax: " <<
minMaxElement.second << std :: endl; //Output: 77

// ESEMPIO CLAMP
// Blocca, fissa il valore dentro i boundaries
// In questo caso il valore è 1.5f ed i boundaries sono tra 0.0f e 1.0f
// 1.5f supera il boundary sopra lo 1.0 quindi il valore della
// variabile sarà 1.0f
auto clamped = std :: clamp(1.5f, 0.0f, 1.0f);

std :: cout << clamped << std :: endl; //Output: 1

clamped = std :: clamp(-2.0f, 0.0f, 1.0f);

std :: cout << clamped << std :: endl; //Output: 0

```

```
    return 0;  
}
```



Per tutte queste operazioni c'è un equivalente per *ranges* del C++20 a pag. 347

---

## *Operazioni di Comparazione*

---

**Definizione:** Queste operazioni ci permettono di comparare elementi delle sequenze: `std::equal`, `std::lexicographical_compare`, `std::lexicographical_compare_three_way`.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
#include <string_view>

// ESEMPIO EQUAL
// Palindromo: sequenza di caratteri, che letta al contrario, rimane
// invariata.
// std::string_view serve per evitare di copiare dati che sono già posseduti
// da qualche altra parte.
bool isPalindrome( const std :: string_view& s )
{
    return std :: equal(s . begin () , s . begin () + s . size ()
        /2 , s . rbegin ());
}

void check( const std :: string_view& s )
{
    std :: cout << s << (isPalindrome(s) ? " è " : " non
    è ")
    << " un palindromo \n";
}

int main()
{
    check( "ANNA" ); //Output: ANNA è un palindromo

    // ESEMPIO LEXICOGRAPHICAL_COMPARE
    std :: vector<std :: string> a = { 7, 9, 0, 8, 5 };
    std :: vector<std :: string> b = { 7, 9, 8, 0, 5 };
}
```

```

std::cout << std::lexicographical_compare(a.begin()
() , a.end() , b.begin() , b.end()) << std::endl;
//Output: false

// ESEMPIO LEXICOGRAPHICAL_COMPARE 2
std::vector<std::string> v1 = {"One" , "Two" , "
Three"};
std::vector<std::string> v2 = {"one" , "two" , "
three"};

bool compare = std::lexicographical_compare(v1.
begin() , v1.end() , v2.begin() , v2.end());

std::cout << (compare ? "v1 è minore di v2" : "v1
non è minore di v2") << std::endl; //Output: v1 è
minore di v2

v1[0] = "two";

compare = std::lexicographical_compare(v1.begin() ,
v1.end() , v2.begin() , v2.end());

std::cout << (compare ? "v1 è minore di v2" : "v1
non è minore di v2") << std::endl; //Output: v1
non è minore di v2

return 0;
}

```



Per tutte queste operazioni c'è un equivalente per *ranges* del C++20 a pag. 348

---

## *Operazioni su Permutazioni*

---

**Definizione:** Queste servono per usufruire di operazioni su permutazioni:

`std::is_permutation, std::next_permutation, std::prev_permutation.`

Una **permutazione** è un modo di ordinare in successione degli oggetti distinti.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>

int main()
{
    // ESEMPIO IS_PERMUTATION
    std::string s1 = "abcd";
    std::string s2 = "bdac";
    std::string s3 = "bdbc";

    std::cout << std::boolalpha << std::is_permutation
        (s1.begin(), s1.end(), s2.begin()) << std::endl
        ; //Output: true

    std::cout << std::boolalpha << std::is_permutation
        (s1.begin(), s1.end(), s3.begin()) << std::endl
        ; //Output: false

    // ESEMPIO PREV_PERMUTATION, NEXT_PERMUTATION
    std::cout << s1 << std::endl; //Output:

    std::prev_permutation(s1.begin(), s1.end());

    std::cout << "prev_permutation: " << s1 << std::endl; //Output: dcba

    std::next_permutation(s1.begin(), s1.end());
}
```

```
    std::cout << "next_permutation: " << s1 << std::endl; //Output: abcd  
  
    return 0;  
}
```



Per tutte queste operazioni c'è un equivalente per *ranges* del C++20 a pag. 349

---

## *Operazioni Numeriche*

---

**Definizione:** Le **operazioni numeriche** mettono a disposizione una serie di comuni funzioni matematiche su delle sequenze numeriche ed anche altre.

Questi operazioni rispetto alle altre, anche se appartenenti alla *Libreria degli Algoritmi* sono definite nell'header <numeric>.

Queste operazioni sono: std::accumulate, std::reduce, std::inner\_product, std::adjacent\_difference, std::partial\_sum, std::exclusive\_scan, std::inclusive\_scan, std::transform\_reduce, std::transform\_exclusive\_scan, std::transform\_inclusive\_scan, std::iota.

```
#include <iostream>
#include <numeric>
#include <vector>
#include <iterator>

int main()
{
    // ESEMPIO ACCUMULATE, REDUCE
    std :: vector<int> v = { 24, 77, -6, 14, 22, -90,
                           12, 3};
    // Lo 0 nel std::accumulate è il valore di partenza, in questo caso
    // parte da un valore di partenza di zero e somma tutti gli elementi
    // del vettore v.
    std :: cout << std :: accumulate(v.begin(), v.end(),
                                      0) << std :: endl; //Output: 56

    std :: cout << std :: reduce(v.begin(), v.end(), 0) <<
    std :: endl; //Output: 56

    // ESEMPIO ACCUMULATE, REDUCE 2
    std :: vector<std :: string> s = {"ciao", " a ", "
                                         tutti"};
    std :: cout << std :: accumulate(s.begin(), s.end(),
                                      std :: string(" ")); //Output: ciao a tutti
```

```

std::cout << std::reduce(s.begin(), s.end(), std::
    string("")); //Output: ciao a tutti

// ESEMPIO INNER_PRODUCT
std::vector<int> a = { 7, 3, 2 };
std::vector<int> b = { 5, 8, 6 };

std::cout << std::inner_product( a.begin(), a.end
    (), b.begin(), 0) << std::endl; //Output: 71

std::cout << std::inner_product( a.begin(), a.end
    (), b.begin(), 0, std::minus<int>{}, std::
    multiplies<int>{}) << std::endl; //Output: -71

// ESEMPIO ADJACENT_DIFFERENCE, PARTIAL_SUM
std::vector<int> vec = { 13, 2, 5, 4, 7, 9};
std::vector<int> vec2;
std::vector<int> vec3;

// Fa una differenza tra il secondo e il primo di ogni coppia di
// elementi della sequenza.
std::adjacent_difference( vec.begin(), vec.end(),
    std::back_inserter( vec2 ));

for( const auto& i : vec2)
{
    std::cout << i << " ";
}

std::cout << '\n';

//Output: 13 -11 3 -1 3 2

std::partial_sum( vec.begin(), vec.end(), std::
    back_inserter( vec3 ));

for( const auto& i : vec3)
{
    std::cout << i << " ";
}

```

```

std :: cout << '\n';

//Output: 13 15 20 24 31 40

// ESEMPIO INCLUSIVE_SCAN, EXCLUSIVE_SCAN
std :: vector<int> v1 = { 12, 3, 7, 8, 44, 9};
std :: vector<int> v2;
std :: vector<int> v3;

// Esegue un' operazione di somma di prefisso inclusiva con
// binary_op.
std :: inclusive_scan( v1.begin(), v1.end(), std :: back_inserter( v2 ) );

for( const auto& i : v2 )
{
    std :: cout << i << " ";
}

std :: cout << '\n';

//Output: 12 15 22 30 74 83

// Esegue un' operazione di somma di prefisso esclusiva con
// binary_op.
std :: exclusive_scan( v1.begin(), v1.end(), std :: back_inserter( v3 ), 0 );

for( const auto& i : v3 )
{
    std :: cout << i << " ";
}

std :: cout << '\n';

//Output: 0 12 15 22 30 74

// ESEMPIO TRANSFORM_REDUCE
std :: vector<double> xvalues(567, 1.0), yvalues

```

```
(567, 1.0);

double result = std::transform_reduce(xvalues.
    begin(), xvalues.end(), yvalues.begin(), 0.0);
std::cout << result << '\n'; //Output: 567

return 0;
}
```

### Differenza tra `std::accumulate` vs `std::reduce`

accumulate	reduce
Ordina da sinistra a destra.	Non garantisce l'ordine.
Non ha le <i>execution policies</i> .	Ha le <i>execution policies</i> .



Per tutte queste operazioni c'è un equivalente per *ranges* del C++20 a pag. 350

### *Operazioni su memoria non inizializzata*

**Definizione:** Queste operazioni servono per eseguire operazioni su memorie non inizializzate: `std::uninitialized_copy`, `std::uninitialized_copy_n`, `std::uninitialized_fill`, `std::uninitialized_fill_n`, `std::uninitialized_move`, `std::uninitialized_move_n`, `std::uninitialized_default_construct`, `std::uninitialized_default_construct_n`, `std::uninitialized_value_construct`, `std::uninitialized_value_construct_n`, `std::construct`, `std::destroy`, `std::destroy_at`, `std::construct_at`, `std::destroy_n`. Per quanto riguarda `std::get_temporary_buffer` e `std::return_temporary_buffer` sono state rimosse col C++20.

Non fanno parte della *Libreria degli Algoritmi*, ma sono comunque molto utili legate ad essa.

Sono definite nell'header <memory>.

```
}

// std::launder performa il Memory Laundering, questo previene il
// compilatore dal tracciare da dove hai preso questo oggetto, così
// evitando ottimizzazioni che potrebbero non essere più
// applicabili.
auto ptr = std::launder( reinterpret_cast<A*>(
    buffer));

std::destroy(ptr, ptr + 8);

// Al posto di std::destroy si potrebbe usare std::destroy_n:
std::destroy_n(ptr, 8);

// Al posto di std::destroy_n si potrebbe usare std::destroy_at:
for (int i = 0; i < 8; ++i)
{
    std::destroy_at(ptr + i);
}

//Output: 0 distrutto
//Output: 1 distrutto
//Output: 2 distrutto
//Output: 3 distrutto
//Output: 4 distrutto
//Output: 5 distrutto
//Output: 6 distrutto
//Output: 7 distrutto

// ESEMPIO UNINITIALIZED_DEFAULT_CONSTRUCT,
// UNINITIALIZED_DEFAULT_CONSTRUCT_N

class S {
public:
    std::string m{ "Valore di default" };
};

constexpr int n {3};
// alignof restituisce l'allineamento del dato passato come
// parametro.
```

```

alignas(alignof(S)) unsigned char mem[n * sizeof(S
)];

try
{
    auto first {reinterpret_cast<S*>(mem)};
    auto last {first + n};

    std::uninitialized_default_construct(first,
                                         last);

    for (auto it {first}; it != last; ++it) {
        std::cout << it->m << '\n';
    }

    std::destroy(first, last);
}
catch (...) // In questo modo con ... (ellissi) prendiamo
            // qualsiasi eccezione.
{
    std::cout << "Eccezione!\n";
}

// Da notare che per "i tipi triviali"
// il uninitialized_default_construct generalmente non riempie di
// zero la data area di memoria non inizializzata.
int v[] { 1, 2, 3, 4 };
const int original[] { 1, 2, 3, 4 };
std::uninitialized_default_construct(std::begin(v)
, std::end(v));

// Oppure al posto di std::uninitialized_default_construct si poteva
// usare:
std::uninitialized_default_construct_n(std::begin(
v), std::size(v));

std::cout << (std::memcmp(v, original, sizeof(v))
== 0 ? "Non modificato\n" : "Modificato\n");
// Il risultato non è specificato.

```

```
//Output: Valore di default
//Output: Valore di default
//Output: Valore di default
//Output: Non modificato

// ESEMPIO UNINITIALIZED_VALUE_CONSTRUCT,
// UNINITIALIZED_VALUE_CONSTRUCT_N
class Z {
public:
    std::string m{ "Valore di default" };
};

constexpr int n {3};
alignas(alignof(Z)) unsigned char mem[n * sizeof(Z)];

try
{
    auto first {reinterpret_cast<Z*>(mem)};
    auto last {first + n};

    std::uninitialized_value_construct(first, last);

    for (auto it {first}; it != last; ++it) {
        std::cout << it->m << '\n';
    }

    std::destroy(first, last);
}
catch (...) {
    std::cout << "Eccezione!\n";
}

// Da notare che per "i tipi triviali" il uninitialized_value_construct
// non riempie di zero la data area di memoria non inizializzata.
int v[] { 1, 2, 3, 4 };
for (const int i : v) { std::cout << i << ' '; }
std::cout << '\n';
```

```
std::uninitialized_value_construct(std::begin(v),  
        std::end(v));  
  
// Oppure al posto std::uninitialized_value_construct si può usare:  
std::uninitialized_value_construct_n(std::begin(v)  
        , std::size(v));  
  
for (const int i : v) { std::cout << i << ' '; }  
std::cout << '\n';  
  
//Output: Valore di default  
//Output: Valore di default  
//Output: Valore di default  
//Output: 1 2 3 4  
//Output: 0 0 0 0  
}
```



Per tutte queste operazioni c'è un equivalente per *ranges* del C++20 a pag. 351

---

## *Optionals*

---

**Definizione:** Gli **optionals** permettono di definire valori opzionali che per dunque possono esistere come no.

Questi non fanno parte della *Libreria degli Algoritmi*, ma possono comunque essere utili in correlazione ad essa.

Per ottenere il valore del `std::optional` utilizziamo l'operatore `*`.

Sono definiti nel file di intestazione: `<optional>`.

```
#include <iostream>
#include <optional>
#include <string>
#include <fstream>

// Al posto di std::nullopt si poteva anche mettere {}.
// Quelli nell'uguale sono i valori di default se i parametri non vengono
// passati quando viene chiamata la funzione.
int foo(int x, std::optional<int> y = std::nullopt,
        int z = 0)
{
    return x * y.value_or( 17 ) + z;
}

// Funzione che ritorna std::pair con std::optional
std::pair<bool, std::string> loadFile()
{
    std::ifstream file{ "test.txt" };
    if( file )
    {
        return { true, std::string{
                    std::istreambuf_iterator{ file },
                    std::istreambuf_iterator<char>{}}
                };
    }
    return { false, {} };
}
```

```

int main()
{
    // ESEMPIO OPTIONAL
    struct Point {
        float x, y;
        Point(float x, float y) : x(x), y(y) {}
    };

    // Questi sono tutti tipi di optionals empty (vuoti).
    std::optional<int> oEmpty;
    std::optional<int> oEmpty2 {};
    std::optional<int> oEmpty3 = {};
    std::optional<float> oEmpty4 = std::nullopt;

    // Qui assegniamo un valore direttamente
    std::optional<int> oInt = 34;
    std::optional<int> oInt2(9);
    std::optional oP( Point{ 7.2f, 4.1f } );

    // make_optional
    auto oDouble = std::make_optional( 3.69 );
    auto oPoint = std::make_optional<Point>( 3.2f, 1.4
                                              f );

    // emplacement
    std::optional<Point> oEmpty5;
    oEmpty5.emplace( 4.5f, 1.7f );

    // in_place
    std::optional<Point> oEmpty6{ std::in_place, 2.2f,
                                 4.4f };

    std::cout << *oInt << std::endl; //Output: 34

    // Si può evitare di mettere il * per fare i paragoni oppure lo si può
    // anche mettere.
    if(oInt > oInt2)
    {
        std::cout << *oInt << " è maggiore di " << *
        oInt2 << std::endl;
    }
}

```

```
}

//Output: 34 è maggiore di 9

// Nelle funzioni

std::cout << foo(5, 23, 8) << std::endl; //Output:
123

std::cout << foo(5, {}, 8) << std::endl; //Output:
93

std::cout << foo(5, {}) << std::endl; //Output: 85

std::cout << foo(5, 23) << std::endl; //Output: 115

return 0;
}
```

---

## ***Execution Policies***

---

**Definizione:** Le **execution policies** servono per l'esecuzione parallelizzata di un programma.

La *parallelizzazione* è quando molteplici calcoli vengono eseguiti simultaneamente da più *threads*.

Queste non fanno parte della *Libreria degli Algoritmi*, ma possono essere utili soprattutto se si eseguono programmi con le threads.

Alcuni degli algoritmi della *Libreria degli Algoritmi* usufruiscono delle **execution policies**, quindi può essere utile fare chiarezza su quest'ultime.

Le **execution policies** sono definite nell'header `<execution>`.

Utilizzeremo questa classe *Timer* per tenere traccia di quanto tempo è stato necessario per eseguire una porzione di codice.

```
// Nell'header file: Timer.h
#pragma once
#include <chrono>

class Timer {
public:
    Timer() noexcept;

    float mark() noexcept;

    float peek() const noexcept;

private:
    std::chrono::steady_clock::time_point last;
};

// Nel file di implementazione: Timer.cpp
#include "Timer.h"

Timer::Timer() noexcept
{
    last = std::chrono::steady_clock::now();
}
```

```

}

float Timer::mark() noexcept
{
    const auto old = last;
    last = std::chrono::steady_clock::now();
    const std::chrono::duration<float> frameTime =
        last - old;
    return frameTime.count();
}

float Timer::peek() const noexcept
{
    return std::chrono::duration<float>(std::chrono::
        steady_clock::now() - last).count();
}

```

Un primo esempio dell'efficienza delle **execution policies**:

```

#include <iostream>
#include <execution>
#include <vector>
#include <random>
#include <algorithm>
#include <chrono>
#include "Timer.h"

void test()
{
    std::mt19937 rne{ 72 };
    std::uniform_real_distribution<float> d(-10., 10.
        );
    auto rng = [&rne, &d](){ return d( rne ); };

    constexpr size_t n = 60000000;
    std::vector<float> v( n );
    std::generate( v.begin(), v.end(), rng );

    Timer timer;
    timer.mark();
    std::sort( v.begin(), v.end() );
    auto t = timer.peek();

```

```

        std :: cout << t << std :: endl;
    }

void testWithExecPolicies()
{
    std :: mt19937 rne{ 72 };
    std :: uniform_real_distribution<float> d(-10., 10.
        );
    auto rng = [&rne, &d](){ return d( rne );};

    constexpr size_t n = 60000000;
    std :: vector<float> v( n );
    std :: generate( v.begin(), v.end(), rng );

    Timer timer;
    timer.mark();
    std :: sort( std :: execution :: par, v.begin(), v.end()
        );
    auto t = timer.peek();

    std :: cout << t << std :: endl;
}

int main()
{
    test(); //Output: 6sec tempo diverso per tutti ma maggiore
            //rispetto all'altro test. (l'output è sempre diverso, in base al tuo
            //computer, alla sequenza, ecc.)

    testWithExecPolicies(); //Output: 1.3sec tempo diverso
                           //per tutti, ma minore rispetto all'altro test. (l'output è sempre
                           //diverso, in base al tuo computer, alla sequenza, ecc.)

    return 0;
}

```

Con *par* eseguiamo la parte di codice indicata in modo *parallelo*.

```

#include <iostream>
#include <execution>
#include <vector>

```

```

#include <algorithm>
#include "Timer.h"

void test()
{
    std::mt19937 rne{ 54 };
    std::uniform_real_distribution<double> d( 1., 10.
        ); // I valori negativi potrebbero causare dei problemi.
    auto rng = [&rne, &d](){ { return d( rne ); } };

    constexpr size_t n = 30000000;
    std::vector<double> v( n );
    std::generate( v.begin(), v.end(), rng );

    Timer timer;
    timer.mark();

    std::transform( std::execution::par, v.begin(), v.
        end(), v.begin(), [] (double a)
    {
        const auto b = std::acos( 1.0 / a );
        return std::pow(b, b);
    });
    auto t = timer.peek();

    std::cout << t << std::endl;
}

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

```

Un terzo esempio, in cui però senza la *mutua esclusione* (mutex) per evitare la *race condition* il programma non darebbe lo stesso risultato.

La *race condition* è quando due *threads* (flussi di istruzioni che possono essere eseguiti in modo concorrente) arrivano entrambe alla stessa variabile ed entrambe cambiano il valore di essa nello stesso momento. Quindi una stessa variabile di valore 9 potrebbe essere aumentata di 1 da due *threads* portando il valore di suddetta variabile a 10, al posto di 11.

```

#include <iostream>
#include <execution>
#include <vector>
#include <thread>
#include <mutex>
#include <algorithm>
#include "Timer.h"

void testWithoutMutex()
{
    std::mt19937 rne{ 72 };
    std::uniform_real_distribution<double> d( 1., 10.
        );
    auto rng = [&rne, &d](){ { return d( rne ); } ;

    constexpr size_t n = 30000000;
    std::vector<double> v( n );
    std::vector<double> q( n );
    std::generate( v.begin(), v.end(), rng );

    Timer timer;
    timer.mark();

    std::for_each( v.begin(), v.end(), [&rng](double&
        a) )
    {
        a = rng();
    );
    auto t1 = timer.peek();

    rne.seed( 72 );

    timer.mark();
    std::for_each( std::execution::par, v.begin(), v.
        end(), [&rng](double& a) )
    {
        a = rng();
    );
    auto t2 = timer.peek();
}

```

```

    std::sort( std::execution::par , v.begin() , v.end()
               );
    std::sort( std::execution::par , q.begin() , q.end()
               );

    std::cout << std::boolalpha << (v == q) << std::
        endl; //Output: viene false se non li ordini (col sort), ma
              senza la mutex verrebbe comunque false
}

void testWithMutex()
{
    std::mt19937 rne{ 72 };
    std::uniform_real_distribution<double> d( 1., 10.
        );
    auto rng = [&rne, &d]() { return d( rne ); };

    constexpr size_t n = 30000000;
    std::vector<double> v( n );
    std::vector<double> q( n );
    std::generate( v.begin() , v.end() , rng );

    Timer timer;
    timer.mark();

    std::for_each( v.begin() , v.end() , [&rng]( double&
        a ) )
    {
        a = rng();
    });
    auto t1 = timer.peek();

    rne.seed( 72 );
    std::mutex mutex;

    timer.mark();
    std::for_each( std::execution::par , v.begin() , v.
        end() , [&rng, &mutex]( double& a ) )
    {
        // In questo modo con la mutex, solo una thread alla volta ha

```

accesso alla variabile e può modificarla, poi quando ha finito rilascia la risorsa e anche le altre threads possono lockare la variabile ed utilizzarla.

```

    std :: lock_guard<std :: mutex> lock ( mutex ) ;
    a = rng () ;
} );
auto t2 = timer . peek () ;

std :: cout << std :: boolalpha << ( v == q ) << std :: endl ; //Output: false (non sono la stessa sequenza, ma sono gli stessi numeri è per questo che riordinandoli ci da true poi)

std :: sort ( std :: execution :: par , v . begin () , v . end () );
std :: sort ( std :: execution :: par , q . begin () , q . end () );

std :: cout << std :: boolalpha << ( v == q ) << std :: endl ; //Output: true
}

int main ()
{
    testWithoutMutex ();

    testWithMutex ();
    return 0;
}

```

Per la questione del *Multithreading* servirebbe un intero capitolo a parte per spiegare, approfondire il funzionamento, cosa sono i *threads*, cos'è la *mutex*, cosa sono le *condition variables*, i *semaphore*, eccetera.



Per tutte queste operazioni c'è un equivalente per *ranges* del C++20 a pag. 333

## C++20

**Definizione:** Il C++20 ha introdotto nel linguaggio diverse funzionalità utili agli algoritmi, formando così la versione *Constrained Algorithms* della maggior parte degli algoritmi dando vita al namespace `std::ranges`.

Questa versione del linguaggio ha introdotto una serie di funzionalità, alcune di queste sono: **Ranges**, **Concepts**, **Modules**, **Coroutine** e altro ancora.

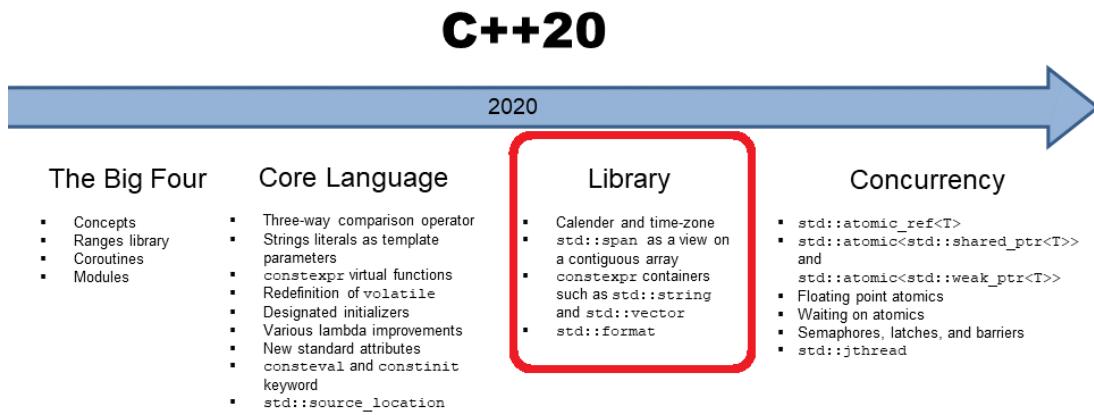


Figura 4.3: C++20 Timeline

## Ranges

**Definizione:** I **ranges** sono, concettualmente, un paio di iteratori, all'inizio ed alla fine della sequenza ( o una *sentinel*, sentinella in certi casi).

I **ranges** velocizzano la chiamata alle funzioni della *Libreria degli Algoritmi* in cui al posto di dover passare un iteratore (`.begin()`, `.end()`) con i **ranges** possiamo semplicemente passare l'oggetto, la sequenza stessa.

## Tipi di Ranges

I ranges possono essere *input ranges* (possono essere letti), *output ranges* (ci si può scrivere) o entrambi.

Gli *input ranges*:

Concetto	Descrizione
<code>std::ranges::input_range</code>	può essere iterato dall'inizio alla fine almeno una volta.
<code>std::ranges::forward_range</code>	può essere iterato dall'inizio alla fine molteplici volte.
<code>std::ranges::bidirectional_range</code>	può muoversi all'indietro con <code>- -</code> .
<code>std::ranges::random_access_range</code>	si possono saltare gli elementi con <code>[ ]</code> .
<code>std::ranges::contiguous_range</code>	gli elementi vengono memorizzati consecutivamente in memoria.

Questi concetti sono derivati direttamente dai rispettivi concetti negli iteratori.

Ci sono anche dei **ranges** che sono indipendenti dall'*input* o *output*, per esempio `std::ranges::sized_range` che richiede che lo spazio di un range sia recuperabile da `std::ranges::size()`.

Le `ranges::views` sono dei **ranges** che trasformano il **range** attraverso degli algoritmi o delle operazioni. Le **views** non possiedono alcuna data (nel senso dati, informazioni) eccetto il loro algoritmo e il tempo di costruzione; distruzione o copiatura non dovrebbe dipendere dal numero di elementi che rappresentano.

I **ranges** sono definiti nel file di intestazione: `<ranges>`.

Di seguito, alcuni esempi degli algoritmi già visti, ma con l'utilizzo dei **ranges**.

### *Operazioni su sequenze non modificabili*

```
#include <iostream>
#include <algorithm>
#include <string_view>
#include <ranges>

int main()
{
    // ESEMPIO RANGES::STARTS_WITH
    std :: cout << std :: boolalpha << std :: ranges :: starts_with( "luce" , "lucertola" ) << '\n' ; // Output: true
    std :: cout << std :: boolalpha << std :: ranges :: starts_with( "const" , "cast" ) << '\n' ; //Output: false
```

```

// ESEMPIO RANGES::ENDS_WITH
std::cout << std::boolalpha << std::ranges::
    ends_with("ter", "computer") << '\n'; //Output:
    true
std::cout << std::boolalpha << std::ranges::
    ends_with("asse", "casse") << '\n'; //Output: true
std::cout << std::boolalpha << std::ranges::
    ends_with("ada", "scheda") << '\n'; //Output:
    false
return 0;
}

```

### *Operazioni su sequenze modificabili*

```

#include <iostream>
#include <algorithm>
#include <vector>
#include <ranges>
#include <string>
#include <random>
#include <array>

void printContainer(const auto& container)
{
    for(const auto& o : container)
    {
        std::cout << o << ' ';
    }
    std::cout << '\n';
}

int main()
{
    // ESEMPIO RANGES::VIEWS::TRANSFORM
    namespace rn = std::ranges;
    namespace vi = std::ranges::views;

    std::vector<int> numbers = { 7, 8, 1, 2, 5, 3 };

```

```

auto result = numbers | vi::filter([](int n) {
    return n % 2 == 0; })
| vi::transform([](int n) { return n * 2; });

for(const int& n : result)
{
    std::cout << n << ' ';
}

//Output: 16 4

// ESEMPIO RANGES::SAMPLE
constexpr int NUM_OF LETTERS = 6;
std::string in = "gjhcioz", out;
std::sample(in.begin(), in.end(), std::
    back_inserter(out),
NUM_OF LETTERS, std::mt19937{std::random_device
    {}()});
std::cout << NUM_OF LETTERS << " lettere casuali
da: " << in << " : " << out << '\n';

//Output: 6 lettere casuali da: gjhcioz : gjcioz

// ESEMPIO RANGES::REPLACE, RANGES::REPLACE_IF
std::array<int, 10> arr{5, 7, 4, 2, 8, 6, 1, 9, 0,
3};

std::replace(arr.begin(), arr.end(), 8, 88);

printContainer(arr);

std::replace_if(arr.begin(), arr.end(),
std::bind(std::less<int>(), std::placeholders::_1,
3), 33);
printContainer(arr);

//Output: 5 7 4 2 88 6 1 9 0 3
//Output: 5 7 4 33 88 6 33 9 33 3
return 0;
}

```

## *Operazioni su Partizioni*

```

#include <iostream>
#include <algorithm>
#include <vector>
#include <ranges>
#include <cctype>
#include <iterator>
#include <array> // per std::array
#include <utility> // per std::as_const

void printContainer(const auto& container)
{
    for(const auto& o : container)
    {
        std::cout << o << ' ';
    }
    std::cout << '\n';
}

int main()
{
    // ESEMPIO RANGES::PARTITION_COPY
    const auto in = {'N', '0', 'U', 'M', '4', 'B', '9',
                    'E', '1', '0', 'R', '6'};

    std::vector<int> v1(size(in)), v2(size(in));

    auto pred = [](&char c){ return std::isalpha(c); };

    auto result = std::ranges::partition_copy(in, v1.
                                              begin(), v2.begin(), pred);

    std::ostream_iterator<char> cout {std::cout, " "};
    std::cout << "in = ";
    std::ranges::copy(in, cout);
    std::cout << "\n";
    std::copy(v1.begin(), result.out1, cout);
    std::cout << "\n";
    std::copy(v2.begin(), result.out2, cout);
}

```

```

std :: cout << '\n';

//Output: in = N 0 U M 4 B 9 E 1 0 R 6
//Output: v1 = N U M B E R
//Output: v2 = 0 4 9 1 0 6

// ESEMPIO RANGES::PARTITION_POINT
std :: array v = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

auto is_even = [] (int i) { return i % 2 == 0; };

std :: ranges :: partition (v, is_even);

std :: cout << "Dopo la partizione: ";
printContainer (v);

const auto pp = std :: ranges :: partition_point (v,
    is_even);
const auto i = std :: ranges :: distance (v.cbegin (), 
    pp);
std :: cout << "Partition point è a: " << i << " ; v["
" << i << "] = " << *pp << '\n';

std :: cout << "Prima partizione (tutti numeri pari)
: ";
printContainer (v);

std :: cout << "Seconda partizione (tutti numeri
dispari): ";
printContainer (v);

//Output: Dopo la partizione: 8 2 6 4 5 3 7 1 9
//Output: Partition point è a: 4; v[4] = 5
//Output: Prima partizione (tutti numeri pari): 8 2 6 4 5 3 7 1 9
//Output: Seconda partizione (tutti numeri dispari): 8 2 6 4 5 3 7 1 9

// ESEMPIO RANGES::IS_PARTITIONED, RANGES::PARTITION,
std :: array<int , 9> v;

auto isEven = [] (int i) { return i % 2 == 0; };

```

```

auto print = [&](bool o) {
    for (int x : v) std::cout << x << ' ';
    std::cout << (o ? "=> " : "> non ") << "
        partizionato\n";
};

std::iota(v.begin(), v.end(), 1);
print(std::ranges::is_partitioned(v, isEven));

std::ranges::partition(v, isEven);
print(std::ranges::is_partitioned(std::as_const(v)
    , isEven));

std::ranges::reverse(v);
print(std::ranges::is_partitioned(v.cbegin(), v.
    cend(), isEven));
print(std::ranges::is_partitioned(v.cbegin(), v.
    rend(), isEven));

//Output: 1 2 3 4 5 6 7 8 9 => non partizionato
//Output: 8 2 6 4 5 3 7 1 9 => partizionato
//Output: 9 1 7 3 5 4 6 2 8 => non partizionato
//Output: 9 1 7 3 5 4 6 2 8 => partizionato

return 0;
}

```

### *Operazioni di Ordinamento*

```

#include <iostream>
#include <algorithm>
#include <vector>
#include <ranges>
#include <array>

int main()
{
    // ESEMPIO RANGES::SORT, RANGES::IS_SORTED
    namespace ranges = std::ranges;

```

```

    std::array arr {7, 8, 4, 2, 5, 3};

    ranges::copy(arr, std::ostream_iterator<int>(std::
        cout, " "));
    std::cout << ": is_sorted: " << std::boolalpha
    << ranges::is_sorted(arr) << '\n';

    ranges::sort(arr);

    ranges::copy(arr, std::ostream_iterator<int>(std::
        cout, " "));
    std::cout << ": is_sorted: "
    << ranges::is_sorted(ranges::begin(arr),
        ranges::end(arr)) << '\n';
    return 0;
}

```

### *Operazioni su ricerca binaria*

```

#include <iostream>
#include <algorithm>
#include <vector>
#include <ranges>

int main()
{
    namespace ranges = std::ranges;

    // ESEMPIO RANGES::LOWER_BOUND,
    // RANGES::UPPER_BOUND
    std::vector<int> vec = { 4, 1, 7, 3, 2, 3, 3, 8,
        4, 4, 5, 8, 6 };

    {
        auto lower = ranges::lower_bound(vec.begin(),
            vec.end(), 4);
        auto upper = ranges::upper_bound(vec.begin(),
            vec.end(), 4);
    }
}

```

```

        ranges::copy(lower, upper, std::
                      ostream_iterator<int>(std::cout, " "));
        std::cout << '\n'; //Output: 8 4 4

    }

{
    auto lower = ranges::lower_bound(vec, 3);
    auto upper = ranges::upper_bound(vec, 3);

    ranges::copy(lower, upper, std::
                  ostream_iterator<int>(std::cout, " "));
    std::cout << '\n'; //Output: 7 3 2 3 3
}

// ESEMPIO RANGES::BINARY_SEARCH
constexpr static auto haystack = {1, 3, 4, 8, 9,
                                 5};
constexpr static auto needles = {1, 2, 3};

for (const int needle : needles) {
    std::cout << "Cercando il: " << needle << ": "
    ;
    std::ranges::binary_search(haystack, needle) ?
        std::cout << "trovato\n" : std::cout << "non trovato!\n";
}
//Output: Cercando il: 1: trovato
//Output: Cercando il: 2: non trovato!
//Output: Cercando il: 3: trovato
return 0;
}

```

### *Operazioni di Fusione | Merge*

```

#include <iostream>
#include <algorithm>
#include <vector>
#include <ranges>

```

```

void printContainer(const auto& container)
{
    for(const auto& o : container)
    {
        std::cout << o << ' ';
    }
    std::cout << '\n';
}

int main()
{
    // ESEMPIO RANGES::MERGE
    std::vector<int> in1, in2, out;

    in1 = {1, 2, 3, 4, 5};
    in2 = {            3, 4, 5, 6, 7};
    out.resize(in1.size() + in2.size());
    const auto ret = std::ranges::merge(in1, in2, out.
        begin()));

    printContainer(in1);
    std::cout << "+" << '\n';
    printContainer(in2);
    std::cout << "=" << '\n';
    printContainer(out);

    //Output: 1 2 3 4 5
    //Output: +
    //Output: 3 4 5 6 7
    //Output: =
    //Output: 1 2 3 3 4 4 5 5 6 7
    return 0;
}

```

### *Operazioni sugli Insiemi*

```

#include <iostream>
#include <algorithm>
#include <vector>
#include <ranges>

```

```

void printContainer(const auto& container)
{
    for(const auto& o : container)
    {
        std::cout << o << ' ';
    }
    std::cout << '\n';
}

int main()
{
    // ESEMPIO RANGES::SET_INTERSECTION
    std::vector<int> v1 = {8, 2, 7, 9, 4, 5, 6};
    std::vector<int> v2 = {2, 3, 4, 3, 5, 7};
    std::vector<int> v3;

    std::ranges::sort(v1.begin(), v1.end());
    std::ranges::sort(v2.begin(), v2.end());

    std::ranges::set_intersection(v1, v2, std::
        back_inserter(v3));

    std::cout << "v1 contiene: ";
    printContainer(v1);

    //Output: v1 contiene: 2 4 5 6 7 8 9

    std::cout << "v2 contiene: ";
    printContainer(v2);

    //Output: v2 contiene: 2 3 3 4 5 7

    std::cout << "l'intersezione v3 contiene: ";
    printContainer(v3);

    //Output: l'intersezione v3 contiene: 2 4 5 7

    // ESEMPIO RANGES::SET_SYMMETRIC_DIFFERENCE
    const auto in1 = {1, 3, 4,      6, 7, 9};

```

```

const auto in2 = {1,      4, 5, 6,      9};

std :: vector<int> out;

std :: ranges :: set_symmetric_difference(in1, in2,
    std :: back_inserter(out));

std :: cout << "in1 contiene: ";
printContainer(in1);

//Output: in1 contiene: 1 3 4 6 7 9

std :: cout << "in2 contiene: ";
printContainer(in2);

//Output: in2 contiene: 1 4 5 6 9

std :: cout << "la differenza simmetrica out
    contiene: ";
printContainer(out);

//Output: la differenza simmetrica out contiene: 3 5 7

// ESEMPIO RANGES::SET_UNION
std :: vector<int> vec1, vec2, vec3;

vec1 = {1, 2, 3, 4, 5};
vec2 = {            3, 4, 5, 6, 7};
vec3.resize(vec1.size() + vec2.size());
std :: ranges :: set_union(vec1, vec2, vec3.begin());

std :: cout << "vec1 contiene: ";
printContainer(vec1);

//Output: vec1 contiene: 1 2 3 4 5

std :: cout << "vec2 contiene: ";
printContainer(vec2);

//Output: vec2 contiene: 3 4 5 6 7

```

```

    std::cout << "vec3 contiene: ";
    printContainer(vec3);

    //Output: vec3 contiene: 1 2 3 4 5 6 7 0 0 0

    return 0;
}

```

### *Operazioni su Heap*

```

#include <iostream>
#include <algorithm>
#include <vector>
#include <ranges>
#include <array>

void printContainer(const auto& container)
{
    for(const auto& o : container)
    {
        std::cout << o << ' ';
    }
    std::cout << '\n';
}

int main()
{
    // ESEMPIO RANGES::MAKE_HEAP, RANGES::POP_HEAP
    std::array v { 4, 7, 5, 1, 3, 9, 8, 6, 5, 2 };
    std::cout << "Inizialmente: ";
    printContainer(v);

    //Output: Inizialmente: 4 7 5 1 3 9 8 6 5 2

    std::ranges::make_heap(v);
    std::cout << "make_heap: ";
    printContainer(v);

    //Output: make_heap: 9 7 8 6 3 5 4 1 5 2
}

```

```

print("convertendo l'heap in un array ordinato:");
for (auto n {std::ssize(v)}; n >= 0; --n) {
    std::ranges::pop_heap(v.begin(), v.begin() + n);
    print("[ ", v.cbegin(), v.cbegin() + n, " ]   ")
    ;
    print("[ ", v.cbegin() + n, v.cend(), " ]\n");
}

//Output:
[ 8 7 5 6 3 2 4 1 5 9 ]   [ ]
[ 7 6 5 5 3 2 4 1 8 ]   [ 9 ]
[ 6 5 5 1 3 2 4 7 ]   [ 8 9 ]
[ 5 5 4 1 3 2 6 ]   [ 7 8 9 ]
[ 5 3 4 1 2 5 ]   [ 6 7 8 9 ]
[ 4 3 2 1 5 ]   [ 5 6 7 8 9 ]
[ 3 1 2 4 ]   [ 5 5 6 7 8 9 ]
[ 2 1 3 ]   [ 4 5 5 6 7 8 9 ]
[ 1 2 ]   [ 3 4 5 5 6 7 8 9 ]
[ 1 ]   [ 2 3 4 5 5 6 7 8 9 ]
[ ]   [ 1 2 3 4 5 5 6 7 8 9 ]

// ESEMPIO RANGES::SORT_HEAP
std::array v {4, 9, 4, 7, 6, 12};
std::cout << "array originale: ";
printContainer(v);

//Output: array originale: 4 9 4 7 6 12

std::ranges::make_heap(v);
std::cout << "dopo il make_heap: ";
printContainer(v);

//Output: dopo il make_heap: 12 9 4 7 6 4

std::ranges::sort_heap(v);
std::cout << "dopo il sort_heap: ";
printContainer(v);

```

```
//Output: dopo il sort_heap: 4 4 6 7 9 12
return 0;
}
```

## *Operazioni su Min/Max*

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <ranges>

int main()
{
    // ESEMPIO RANGES::MIN_ELEMENT, RANGES::MAX_ELEMENT
    std::vector<int> v{ 2, 6, -17, 2, 1, 9 };

    namespace ranges = std::ranges;

    auto result = ranges::max_element(v.begin(), v.end());
    std::cout << "elemento massimo a: " << ranges::
        distance(v.begin(), result) << '\n'; //Output:
    elemento massimo a: 5

    auto absCompare = [] (int a, int b) { return (std::
        abs(a) < std::abs(b)); };
    result = ranges::max_element(v, absCompare);
    std::cout << "elemento massimo (assoluto) a: " <<
        ranges::distance(v.begin(), result) << '\n'; //Output:
    elemento massimo (assoluto) a: 2

    auto result2 = ranges::min_element(v.begin(), v.
        end());
    std::cout << "elemento minimo a: " << ranges::
        distance(v.begin(), result) << '\n'; //Output:
    elemento minimo a: 2

    result = ranges::min_element(v, absCompare);
```

```

    std :: cout << "elemento minimo (assoluto) a: " <<
        ranges :: distance(v.begin(), result2) << '\n';
        //Output: elemento minimo (assoluto) a: 2

    // ESEMPIO RANGES::MINMAX
    const auto v = { 5, 1, 3, 4, 2, 7, 5, 9 };
    const auto [min, max] = ranges :: minmax_element(v);
    std :: cout << "minimo: " << *min << ", all'indice:
        [ " << ranges :: distance(v.begin(), min) << "] \n"
    << "massimo: " << *max << ", all'indice: [ " <<
        ranges :: distance(v.begin(), max) << "] \n";

    //Output: minimo: 1, all'indice: [1]
    //Output: massimo: 9, all'indice: [7]

    return 0;
}

```

### *Operazioni di Comparazione*

```

#include <iostream>
#include <algorithm>
#include <vector>
#include <ranges>
#include <string_view>

// Palindromo: sequenza di caratteri che letta al contrario rimane
// invariata.
constexpr bool isPalindrome(const std :: string_view s)
{
    namespace views = std :: views;
    auto forward = s | views :: take(s.size() / 2);
    auto backward = s | views :: reverse | views :: take(s
        .size() / 2);
    return std :: ranges :: equal(forward, backward);
}

void test(const std :: string_view s)
{

```

```
    std::cout << s << (isPalindrome(s) ? " è" : " non  
    è") << " un palindromo\n";  
}  
  
int main()  
{  
    // ESEMPIO RANGES::EQUAL  
    test("radar"); //Output: radar è un palindromo  
    test("ciao"); //Output: ciao non è un palindromo  
    return 0;  
}
```

## *Operazioni su Permutazioni*

```

    std :: cout << std :: boolalpha << "is_permutation( "
    << r1 << ", " << r2 << " ): " << std :: ranges ::
    is_permutation(r1, r2) << '\n' << "
    is_permutation( "
    << r1 << ", " << r3 << " ): " << std :: ranges ::
    is_permutation(r1, r3) << '\n';

    //Output: is_permutation( 1 2 3 4 5 , 3 5 4 1 2 ): true
    //Output: is_permutation( 1 2 3 4 5 , 3 5 4 1 1 ): false
    return 0;
}

```

### *Operazioni numeriche*

```

#include <iostream>
#include <algorithm>
#include <vector>
#include <ranges>
#include <numeric>
#include <list>
#include <random>

int main()
{
    // ESEMPIO RANGES::IOTA
    std :: list<int> l(10);
    std :: ranges :: iota(l.begin(), l.end(), -4);

    std :: vector<std :: list <int>::iterator> v(l.size());
    std :: ranges :: iota(v, l.begin());

    std :: ranges :: shuffle(v, std :: mt19937{std ::
        random_device {}()});
}

std :: cout << "Contenuti della lista: ";
for(auto n: l) std :: cout << n << ' ';
std :: cout << '\n';

//Output: Contenuti della lista: -4 -3 -2 -1 0 1 2 3 4 5

```

```

    std::cout << "Contenuti della lista, mischiati: ";
    for (auto i: v) std::cout << *i << ' ';
    std::cout << '\n';

//Output: Contenuti della lista, mischiati: 0 -1 3 4 -4 1 -2 -3 2 5

    return 0;
}

```

### *Operazioni su memoria non inizializzata*

```

#include <iostream>
#include <algorithm>
#include <vector>
#include <ranges>
#include <memory>
#include <string>

int main()
{
    // ESEMPIO RANGES::UNINITIALIZED_FILL_N,
    // RANGES::DESTROY
    constexpr int n {3};
    alignas(alignof(std::string)) char out[n * sizeof(
        std::string) ];

    try
    {
        auto first {reinterpret_cast<std::string*>(out
            )};
        auto last = std::ranges::uninitialized_fill_n(
            first, n, "C++");

        for (auto it {first}; it != last; ++it) {
            std::cout << *it << '\n';
        }

        std::ranges::destroy(first, last);
    }
}

```

```

catch (...)
{
    std :: cout << "Exception!\n";
}

//Output: C++
//Output: C++
//Output: C++
return 0;
}

```

### *Tipi di Ritorno*

```

#include <iostream>
#include <algorithm>
#include <vector>
#include <ranges>

int main()
{
    // ESEMPIO RANGES::IN_FUN_RESULT
    int v[ ] { 1, 2, 3 };

    const auto [last, negate] = std :: ranges ::  

        for_each_n(v, std :: size(v),  

        [](int& x) { return x = -x; });

    const auto result = std :: ranges :: for_each(std ::  

        cbegin(v), last,  

        [](int x) { std :: cout << x << ' '; });
    std :: cout << "| ";

    std :: ranges :: for_each(v, negate);
    std :: ranges :: for_each(v, result.fun());
    std :: cout << '\n';

    //Output: -1 -2 -3 | 1 2 3

    // ESEMPIO RANGES::MIN_MAX_RESULT
    constexpr static auto v2 = {3, 1, 4, 1, 5, 9, 2};
}

```

```
{  
    constexpr auto result2 = std::ranges::minmax(  
        v2);  
    static_assert(1 == result2.min && 9 == result2  
        .max);  
}  
  
{  
    constexpr auto result2 = std::ranges::  
        minmax_element(v2);  
    static_assert(1 == *result2.min && 9 == *  
        result2.max);  
}  
  
//Output: [nessun output, non stiamo stampando niente]  
  
// ESEMPIO RANGES::IN_FOUND_RESULT  
int v3[] { 1, 2, 3 };  
  
const auto result3 = std::ranges::next_permutation  
(v3);  
  
std::ranges::for_each(std::cbegin(v3), result3.in,  
    [](int e){ std::cout << e << ' '});  
  
std::cout << std::boolalpha << "\n" "result3.found  
: " << result3.found << '\n';  
  
//Output: 1 3 2  
//Output: result3.found: true  
return 0;  
}
```

## *Concepts*

**Definizione:** I Concepts sono un nuovo approccio rivoluzionario del C++20 per trattare i templates. Permettono di porre dei vincoli (*constraints*) ai parametri dei template per migliorare la leggibilità del codice, la velocità al momento della compilazione e fornire dei migliori messaggi d'errore.

Il C++20 fornisce due nuove keywords: **concept** e **requires** ed un insieme di **concepts** predefiniti dalla *Libreria Standard*.

Per poter usufruire di queste funzionalità sarà necessario includere `<concepts>`.

### *concept*

```
#include <iostream>
#include <concepts>
#include <type_traits>
#include <string>

// template<typename T>
// concept概念名 = concept表达式;

concept integral = std::is_integral_v<T>; // std::is_integral_v da <type_traits> restituisce true o false se il parametro passato è di tipo intero.

template<integral T> // Questo al posto del tipico typename T
class MyClass {
public:
    MyClass(T x)
    {
        this->x = x;
    }

private:
    T x;
};

template<typename T>
class MyClass2 {
public:
```

```

    MyClass(T x)
    {
        this->x = x;
    }

private:
    T x;
};

int main()
{
    std::string s = "ciao";

    int x = 5;

    MyClass<int> instance(s); // Errore template argument
                                deduction/substitution failed. (passiamo una stringa, quando
                                invece c'è il vincolo che i parametri T siano interi)

    MyClass<double> instance2(7.12); // Errore requires
                                      integral<T> (il tipo della classe è double quando noi invece
                                      vogliamo un tipo intero)

    MyClass instance3(x); // Questo va bene. (soddisfa il
                          vincolo che sia intero)

    MyClass<int> instance4(x); // Questo va bene. (soddisfa il
                                vincolo che sia intero)

    MyClass2<double> myClass2Instance(5.55); // Non da
                                                errore perché stiamo usando la classe senza il vincolo.

    MyClass2 myClass2Instance2(s); // Non da errore perché
                                  stiamo usando la classe senza il vincolo.

    return 0;
}

```

In questo esempio, abbiamo usato la keyword **concept** per definire una classe template con un vincolo, in questo caso il vincolo che la classe sia di tipo integrale, ovvero sia un numero intero e quindi non andavano bene né le istanze double né string né qualsiasi

altri tipo diverso dagli interi.

```
#include <iostream>
#include <concepts>

template<typename T>
concept hasStringDataMember = requires(T v) {
    { v.name } -> std::convertible_to<std::string>;
};

class Person {
public:
    int age { 0 };
    std::string name;
};

class Box {
public:
    double weight { 0.0 };
    double volume { 0.0 };
};

int main()
{
    static_assert(hasStringDataMember<Person>);
    static_assert(!hasStringDataMember<Box>); // Qui
        non da errore perché abbiamo messo ! altrimenti lo darebbe
        perché Box non ha un membro di tipo std::string chiamato name.
    return 0;
}
```

### *requires*

```
#include <iostream>
#include <concepts>
#include <string>
#include <numeric> // per std::accumulate
#include <type_traits>

// ESEMPIO REQUIRES 1
template <typename T>
```

```

requires std::convertible_to<T, std::string>
void func(const T& x) {
    std::string s = x;
}

// ESEMPIO REQUIRES 2
template<typename T>
requires std::integral<T> || std::floating_point<T> // 
    o numeri interi o con la virgola (std::integral e std::floating_point
    vengono dal'header <concepts>)
constexpr double average(const std::vector<T> &vec) {
    const double sum = std::accumulate(vec.cbegin(),
        vec.cend(), (T)0);
    return sum / vec.size();
}

int main()
{
    // ESEMPIO REQUIRES 1
    const std::string s = "ciao";
    const int x = 7;

    func(x); // Errore: perché x è di tipo intero e invece noi abbiamo
              // bisogno di un tipo convertibile a std::string.

    func(s); // Questo va bene.

    // ESEMPIO REQUIRES 2
    std::vector<int> v = { 7, 8, 12, 5};

    std::cout << average(v) << std::endl; //Output: 8
    //funziona perché è intero)

    std::vector<float> v2 = { 5.7, 12.4, 8.8, 9.1 };

    std::cout << average(v2) << std::endl; //Output: 9
    //funziona perché è un numero con la virgola) (funziona anche il
    //double, non solo il float)

    std::vector<bool> v3 = { true, false, true, true,

```

```

    false  };

    std::cout << average(v3) << std::endl; //Output: 0.2
    (funziona perché i booleani sono dei numeri, ovvero true = 1;
     false = 0)

    std::vector<std::string> v4 = { "ciao", "a", "
        tutti" };

    std::cout << average(v4) << std::endl; //Output:
    Errore constraints not satisfied (vincoli non soddisfatti perché
    non è né un intero né un numero con la virgola)
    return 0;
}

```

In questo esempio abbiamo visto come **requires** ci permette di validare che una data variabile sia di una determinata tipologia.

```

#include <iostream>
#include <concepts>

// ESEMPIO REQUIRES 1
template <typename T>
concept addable = requires (T a, T b) {
    a+b;
};

void function(addable auto x) {
    std::cout << x << " è sommabile!" << std::endl;
}

// ESEMPIO REQUIRES 2
template <typename T>
requires std::integral<T> ||
(std::invocable<T> &&
std::integral<typename std::invoke_result<T>::type>)
void function2(const T& x) {
    if constexpr (std::invocable<T>) {
        std::cout << "Il risultato della chiamata è: "
        << x() << "\n";
    } else {
        std::cout << "Il valore è: " << x << "\n";
    }
}

```

```

        }
    }

int main()
{
    // ESEMPIO REQUIRES 1
    function(3); //Output: 3 è sommabile!

    function("Hello World!"); //Output: Errore

    // ESEMPIO REQUIRES 2
    function2(6); //Output: Il valore è: 6 (va bene perché è un
                   intero, quindi va nell'else)
    function2([]() { return 3; }); //Output: (va bene
                                   perché è invocabile, restituisce un intero, va nell'if)

    function2(3.3); //Output: Errore perchè non è un intero né
                    un invocabile
    function2([]() { return 3.3; }); //Output: Errore
                                   perchè è un invocabile, ma restituisce un valore non intero.
    return 0;
}

```

La Libreria Standard fornisce vari **concepts** predefiniti nell'header `<concepts>`, come quelli visti `std::integral` e `std::floating_point` e tanti altri.

### ***Concepts fixano, sposano e rimpiazzano SFINAE***

**SFINAE** (*Substitution Failure Is Not An Error*), concetto trattato nel capitolo Concetti Avanzati a pag.256 viene praticamente sostituito dai **Concepts** che possono fare la stessa cosa, ma molto meglio.

## *Modules*

**Definizione:** I **Modules** permettono di velocizzare un programma attraverso l'importazione di **moduli**, porzioni di codice, invece di **includere un header**. Quindi, in un certo senso i **modules** servono in parte, ma non del tutto sostituiscono gli **headers**.

I **Modules** forniscono due nuove keywords: **export** e **import**.

Negli **headers** il *preprocessore* essenzialmente copierà ed incollerà i contenuti dell'**header** al posto dell'**include** e questo per ogni file di implementazione nel programma.

I **Modules** sono come gli **headers**, ma sono anche una **unità di traduzione** (*translation unit*), proprio per questo viene compilata separatamente e solo una volta. La keyword **import** ci permette di ottenere accesso alle dichiarazioni della libreria.

Il vantaggio principale è che i **modules** vengono *parsati* a livello di **abstract syntax tree** (*ABT*) dal compilatore.

Questo ci permette una serie di vantaggi:

1. **Isolamento**: perché un'unità di modulo è un'unità di traduzione separata: possiede la sua serie di macros, dichiarazioni, direttive che non influenzano chi lo *importa*.
2. **Controllo dell'Interfaccia**: perché può dichiarare entità con un link interno, con l'**export**.
3. **Deduplicazione**: perché in molti casi non sarà più necessario fornire una dichiarazione in un header file.
4. **One Definition Rule violation avoidance (ODR)**: un modulo può definire un'entità solo una volta e fornire quella definizione ai clienti.
5. **Ordine di inizializzazione delle variabili non-locali**: perché l'**import** stabilisce un ordine di dipendenza tra le *unità di traduzione* che contengono (uniche) definizioni di variabili.
6. **Dichiarazioni di moduli privati**: le entità dichiarate in un modulo che non sono esportate e non hanno un link interno sono usabili dalle *unità di traduzione* stesse nel modulo.
7. **Stabilità ABI**: le regole dell'**inline** sono state aggiustate per supportare una strategia d'implementazione dove le funzioni *non-inline* possono servire come frontiera *ABI* per gli aggiornamenti delle librerie condivise.

8. **Velocità di compilazione** : perché i contenuti dei moduli non devono essere *riparsati*, in molti casi questo velocizza i tempi di compilazione.
9. **Tooling (Utensili, Strumentazione, Attrezzi)** : le "dichiarazioni strutturali" che riguardano import e module hanno delle restrizioni nel loro uso per renderle leggibili e efficienti da parte degli strumenti che hanno bisogno di capire il grafo delle dipendenze di un progetto. Le restrizioni permettono anche molti, se non tutti, i casi d'uso di queste parole comuni come identificatori.

```
// Mathematics.cpp
export module Mathematics;
export auto plus(auto x, auto y) -> decltype(x+y) {
    return x + y;
}
export namespace Mathematics {
    auto minus(auto x, auto y) -> decltype(x-y) {
        return x - y;
}
void this_function_will_not_be_exported() {}

// In un altro file: main.cpp
import <iostream>;
import Mathematics;
int main() {
    std::cout << "1+2 = " << plus(1,2) << "\n"; // Output: 3
    std::cout << "3-2 = " << Mathematics::minus(3,2)
    << "\n"; // Output: 1
    return 0;
}
```

Abbiamo importato il modulo *Mathematics* e abbiamo usufruito delle sue funzioni nel file *main.cpp*. La funzione *plus* l'abbiamo chiamata direttamente, mentre la *minus* l'abbiamo chiamata con *Mathematics::* visto che si trovava all'interno del *namespace*.

```
export module helloworld;

export const char* getHelloWorld()
{
    return "Hello World!";
}
```

```
// In un altro file: main.cpp
#import helloworld;

#include <iostream>

int main()
{
    std::cout << getHelloWorld() << std::endl; // Output: Hello World!
    return 0;
}
```

In questo esempio, abbiamo il nostro modulo *helloworld* con una singola funzione che restituisce un *const char\** e lo abbiamo importato in un file con la keyword **import** ed abbiamo usato la funzione al suo interno.

### ***Modules Partitions***

Le **modules partitions** servono per suddividere in più file un **module**. Le partizioni sono accessibili solo dal modulo principale, devono essere una singola unità (un singolo file), ma possono essere delle unità di interfaccia (gli *export*) o una solo un'unità. Ma se l'unità è un'unità di interfaccia allora deve essere riesportata nel modulo principale *export import :partition;*. Il modulo principale avrà accesso a tutte le partizioni, anche se non sono segnate come *export*.

```
// helloworld.cpp
export module helloworld;

export import :english;
export import :spanish;
export import :italian;

// helloworld_english.cpp
export module helloworld:english;

export const char* getHelloWorldEng()
{
    return "Hello World!";
}

// helloworld_spanish.cpp
```

```
export module helloworld:spanish;

export const char* getHelloWorldEs()
{
    return "¡Hola Mundo!";
}

// helloworld_italian.cpp
export module helloworld:italian;

export const char* getHelloWorldIta()
{
    return "Ciao Mondo!";
}

// In un altro file: main.cpp
import helloworld;

#include <iostream>
#include <cstdlib>

int main()
{
    int scelta;
    std::cout << "0. Inglese" << '\n';
    std::cout << "1. Spagnolo" << '\n';
    std::cout << "2. Italiano" << '\n';
    std::cout << "Scelta: ";
    std::cin >> scelta;

    switch(scelta)
    {
        case 1:
            std::cout << getHelloWorldEs() << std::
                endl;
            break;
        case 2:
            std::cout << getHelloWorldIta() << std::
                endl;
            break;
    }
}
```

```

        case 0:
    default:
        std::cout << getHelloWorldEng() << std::
            endl;
        break;
    }
    return 0;
}

```

Qui abbiamo creato un modulo chiamato *helloworld* con tre partizioni: *english*, *spanish*, *italian*.

La sintassi *export module <nome-modulo>:<nome-partizione>* dichiara che il dato modulo è una partizione d'interfaccia appartenente a *<nome-modulo>* con la partizione chiamata *<nome-partizione>*.

La sintassi *import :<nome-partizione>* importa la partizione chiamata *<nome-partizione>*.

La sintassi *export import :<nome-partizione>* rende l'entità esportata nel modulo partizione visibile come parte del modulo interfaccia.

### ***Module Implementation***

È un modulo che non ha la keyword *export* prima della keyword *module* nel suo modulo. Le entità dichiarate in una unità d'implementazione sono visibili solo nel modulo in cui fanno parte.

```

// helloworld.cpp
export module helloworld;

import :english;
import :spanish;
import :italian;

export const char* getHelloWorldEng();
export const char* getHelloWorldEs();
export const char* getHelloWorldIta();

// helloworld_english.cpp
module helloworld:english;

const char* getHelloWorldEng()
{
    return "Hello world!";
}

```

```
}

// helloworld_spanish.cpp
module helloworld:spanish;

const char* getHelloWorldEs()
{
    return "¡Hola Mundo!";
}

// helloworld_italian.cpp
module helloworld:italian;

const char* getHelloWorldIta()
{
    return "Ciao Mondo";
}
```

Questo era il nostro precedente esempio se usiamo le *unità d'implementazione*. Abbiamo visto qui solo uno spunto dell'importanza e della potenza dei **Modules**, ma è possibile approfondire molto di più l'argomento.

## Coroutines

**Definizione:** Le **coroutines** sono delle funzioni che sospendono l'esecuzione per poi riprenderla più tardi.

Sono una struttura generale di controllo dove il flusso di controllo è passato tra due differenti routine senza ritorno.

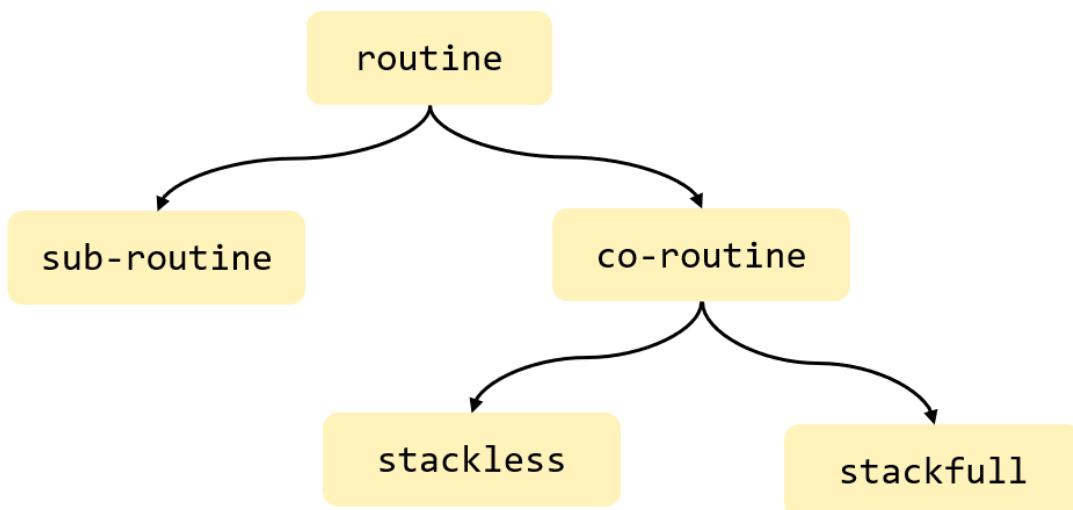


Figura 4.4: C++20 Coroutines in general

- Una **coroutine** è una funzione che:
  1. Contiene le keywords: `co_await`, `co_yield` e/o `co_return`.
  2. Usano un tipo di ritorno per specificare una *promessa*.
- Più in generale, le **coroutine** del **C++20** consistono in:
  - Promise (Promessa)
    - \* Definisce il comportamento della coroutine nel complesso.
    - \* Agisce come comunicatore tra il chiamante e la coroutine chiamata.
  - Awaiter
    - \* Controlla il comportamento di sospensione e di ripresa.
  - Coroutine Handle
    - \* Controlla il comportamento di esecuzione.

### A cosa servono le coroutines?

Le **coroutines** forniscono un alto livello di *concorrenza* con molto poco *overhead*. Una **Coroutine** si può sospendere in un predeterminato punto, quindi si può evitare di *lockare* (bloccare) le strutture dati condivise.

Con i *threads* ogni *thread* necessita di un proprio stack, quindi l'utilizzo di memoria cresce linearmente, mentre con le **coroutines**, il numero di esse che si ha non ha una diretta relazione con l'utilizzo di memoria. Per la maggior parte dei casi, i **coroutines** sono una scelta ottimale visto che son più veloci rispetto ai *threads*.

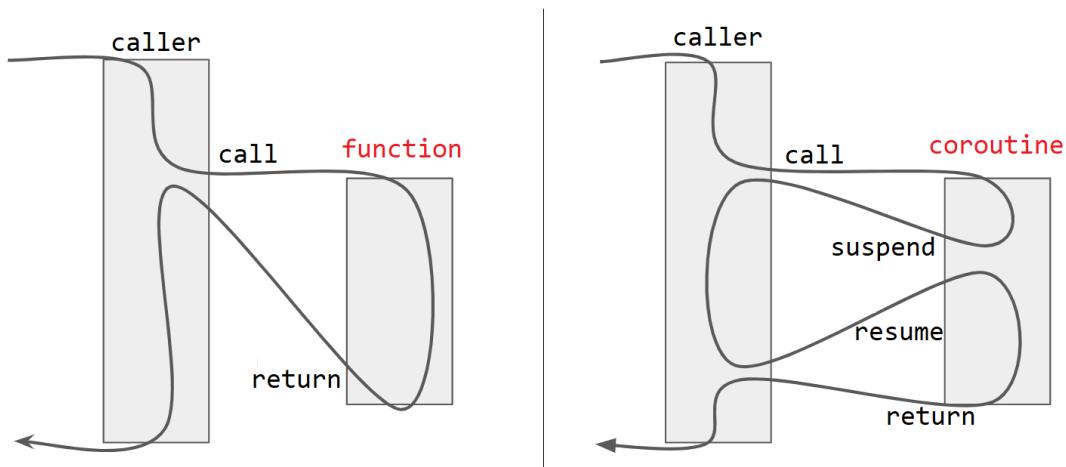


Figura 4.5: C++20 Coroutines

### Terminologia delle Coroutine

#### Awaitable

- È un tipo che supporta l'operatore `co_await`.

```
class dummy { // Awaitable
public:
    std::suspend_always operator co_await() {
        return {};
    }
};
```

```
HelloWorldCoro print_hello_world()
{
    std::cout << "Hello ";
    co_await dummy{};
```

```
    std::cout << "World!" << std::endl;
}
```

### Awaiter

- È un tipo che implementa tre speciali funzioni che sono chiamate come parte dell'espressione `co_await`: `await_ready()`, `await_suspend()`, `await_resume()`. Per esempio, la *Libreria Standard* ha definito degli `awaiters`: `std::suspend_always` e `std::suspend_never`.

- Un tipo può essere sia di tipo `Awaitable` che `Awaiter`.

```
class MyAwaiter {
public:
    bool await_ready() { return false; }
    void await_suspend(std::coroutine_handle<>) {}
    void await_resume() {}
};

HelloWorldCoro print_hello_world()
{
    std::cout << "Hello ";
    co_await MyAwaiter{};
    std::cout << "World!" << std::endl;
}

co_await:
```

- è un operatore unario che sospende la `coroutine` e restituisce il controllo al chiamante. Il suo operando è una espressione il cui l'operatore deve definire `co_await` o `Awaitable`.

```
class dummy {} // Awaitable

class HelloWorldCoro {
public:
    class promise_type {
public:
    // ...
    auto await_transform(const dummy&)
    {
        return std::suspend_always{};
    }
};
```

```

        }
    };
// ...
};

HelloWorldCoro print_hello_world() {
    std::cout << "Hello ";
    co_await dummy{};
    std::cout << "World!" << std::endl;
}

```

**Promise:**

- Un tipo strettamente chiamato *promise\_type*.
- È usato dentro al **coroutine**. La **coroutine** sottomette il proprio risultato o eccezione attraverso questo oggetto.
- La tipologia del *promise type* è determinato dal compilatore dal tipo di ritorno del **coroutine** usando `std::coroutine_traits`.

**Croutine Handle:**

- È usato per riprendere l'esecuzione della **coroutine** o per distruggerla. Indica anche lo stato della **coroutine** usando `std::coroutine_handle::done()`.

**Croutine State:**

- È un oggetto generato dal compilatore, memorizzato sull'heap che contiene:
  - L'oggetto *promise* (promessa).
  - I parametri (tutti copiati per valore).
  - Le variabili locali.
  - La rappresentazione del punto di sospensione attuale, così che la *resume* sappia da dove continuare e la *destroy* sa quali variabili si trovano nello scope (nella portata).

***Esempi di coroutines***

Per usufruire delle **coroutines** è necessario includere `<coroutine>`.

**Sospendere una coroutine:**

```
// Flags compilatore: -std=c++20 -fcoroutines -fno-exceptions
#include <iostream>
#include <coroutine>

class HelloWorldCoro {
public:
    class promise_type { // compiler looks for 'promise_type'
public:
    HelloWorldCoro get_return_object() { return
        this; }
    std::suspend_always initial_suspend() { return
        {}; }
    std::suspend_always final_suspend() { return
        {}; }
};

HelloWorldCoro(promise_type* p) : m_handle(std::
    coroutine_handle<promise_type>::from_promised(*p
)) {}
~HelloWorldCoro() { m_handle.destroy(); }

    std::coroutine_handle<promise_type> m_handle;
};

HelloWorldCoro print_hello_world() {
    std::cout << "Hello ";
    co_await std::suspend_always {};
    std::cout << "World!" << std::endl;
}

int main() {
    HelloWorldCoro mycoro = print_hello_world();

    mycoro.m_handle.resume();
    mycoro.m_handle(); // Equal to mycoro.m_handle.resume();
    return EXIT_SUCCESS;
}
```

Una **coroutine** può essere ripresa da una funzione membro del `std::coroutine_handle` o invocando la funzione operatore dell'oggetto `std::coroutine_handle`.

Come detto in precedenza, la **coroutine** consiste in:

- La *promessa*, nel nostro esempio `promise_type`.
- *Awaiter*, nel nostro esempio `std::suspend_always`.
- *Croutine Handle*, nel nostro esempio `std::coroutine_handle`.

Restituire un valore dalla `coroutine`:

```
// Flags compilatore: -std=c++20 -fcoroutines -fno-exceptions
#include <iostream>
#include <coroutine>
#include <cassert>

class HelloWorldCoro {
public:
    class promise_type {
public:
        int m_value;

        HelloWorldCoro get_return_object() { return
            *this; }
        std::suspend_always initial_suspend() { return
            {}; }
        std::suspend_always final_suspend() { return
            {}; }

        void return_value(int val) { m_value = val; }
    };

    HelloWorldCoro(promise_type* p) : m_handle(std::
        coroutine_handle<promise_type>::from_promise(*p
    )) {}
    ~HelloWorldCoro() { m_handle.destroy(); }

    std::coroutine_handle<promise_type> m_handle;
};

HelloWorldCoro print_hello_world() {
    std::cout << "Hello ";
    co_await std::suspend_always{ };
    std::cout << "World!" << std::endl;
```

```

        co_return -1;
    }

int main() {
    HelloWorldCoro mycoro = print_hello_world();

    mycoro.m_handle.resume();
    mycoro.m_handle.resume();
    assert(mycoro.m_handle.promise().m_value == -1);

    //Output: Hello World!

    return EXIT_SUCCESS;
}

```

Per restituire un valore dalla **coroutine**, c'è bisogno di fornire la funzione *return\_value* al *promise type*.

Rendere (*Yielding*) un valore dalle **coroutines**:

```

// Flags compilatore: -std=c++20 -fcoroutines -fno-exceptions
#include <iostream>
#include <coroutine>
#include <cassert>

class HelloWorldCoro {
public:
    class promise_type {
public:
    int m_val;

    HelloWorldCoro get_return_object() { return
        this; }
    std::suspend_always initial_suspend() { return
        {}; }
    std::suspend_always final_suspend() { return
        {}; }

    std::suspend_always yield_value(int val) {
        m_val = val;
        return {};
    }
}

```

```

};

HelloWorldCoro(promise_type* p) : m_handle(std::
    coroutine_handle<promise_type>::from_promise(*p
)) {}

~HelloWorldCoro() { m_handle.destroy(); }

std::coroutine_handle<promise_type> m_handle;
};

HelloWorldCoro print_hello_world() {
    std::cout << "Hello ";
    co_yield 1;
    std::cout << "World!" << std::endl;
}

int main() {
    HelloWorldCoro mycoro = print_hello_world();

    mycoro.m_handle.resume();
    assert(mycoro.m_handle.promise().m_val == 1);
    mycoro.m_handle.resume();

    //Output: Hello World!
    return EXIT_SUCCESS;
}

```

Per poter rendere, produrre (*yielding*) un valore dalla **coroutine** è necessario fornire la funzione *yield\_value()* al *promise\_type* che restituisce il *awaitable\_type*.  
Generando una sequenza di numeri interi con le **coroutine**:

```

// Flags compilatore: -std=c++20 -fcoroutines -fno-exceptions
#include <iostream>
#include <coroutine>

class Generator {
public:
    class promise_type {
        public:
            int m_val;

        Generator get_return_object() { return this; }
    };
}
```

```

    std::suspend_never initial_suspend() { return
        {}; }
    std::suspend_always final_suspend() { return
        {}; }

    std::suspend_always yield_value(int val) {
        m_val = val;
        return {};
    }

    class iterator {
        public:
        bool operator!=(const iterator& rhs) { return
            not m_h_ptr->done(); }
        iterator& operator++() {
            m_h_ptr->resume();
            return *this;
        }
        int operator*() { return m_h_ptr->promise().
            m_val; }

        std::coroutine_handle<promise_type> *m_h_ptr;
    };

    iterator begin() { return iterator{&m_handle}; }
    iterator end() { return iterator{nullptr}; }

    Generator(promise_type* p) : m_handle(std::
        coroutine_handle<promise_type>::from_promise(*p
        )) {}
    ~Generator() { m_handle.destroy(); }

    std::coroutine_handle<promise_type> m_handle;
};

Generator range(uint32_t start, uint32_t end) {
    while(start != end)
        co_yield start++;
}

```

```

    }

int main() {
    for (auto &&no : range(0, 10)) {
        std::cout << no << std::endl;
    }
    //Output: 0
    //Output: 1
    //Output: 2
    //Output: 3
    //Output: 4
    //Output: 5
    //Output: 6
    //Output: 7
    //Output: 8
    //Output: 9

    return EXIT_SUCCESS;
}

```

Questo è in breve cosa sono e a cosa servono le **coroutines** del C++20, anche qui si potrebbe estendere e approfondire l'argomento, ma per il momento vediamo soltanto un piccolo assaggio.

### ***Stackfull & Stackless Coroutines***

Cosa sono le **stackfull** e le **stackless coroutines**?

- **Stackfull** : Le **stackfull coroutines** hanno bisogno di uno *stack* separato per essere eseguite.
- **Stackless** : Le **stackless coroutines** usano lo stesso *stack* del chiamante.

### ***Differenza tra threads vs coroutines***

- Le **coroutines** riguardano il *programming model* (modello di programmazione) mentre le **threads** riguardano l'*execution model* (modello d'esecuzione).
- Le **coroutines** possono ancora fare concorrenza senza l'*overhead* (sovraffico) dello scheduler, semplicemente gestisce lo *context-switching* stesso.
- Un altro beneficio delle **coroutine** è il loro minore uso di memoria. Con i **thread**, ognuno di essi ha bisogno di allocare il proprio *stack* e così la memoria cresce

linearmente, mentre il numero di **coroutines** non ha una relazione diretta con l'utilizzo di memoria.

Queste sono le *4 Big features* del **C++20** che cambieranno il modo di scrivere C++.

Le abbiamo soltanto trattate in modo breve e conciso, ma sono 4 argomenti fondamentali del **C++20**.

# Linee guida del Linguaggio

5

---

## *Introduzione*

---

In questo capitolo tratterò delle **CppCoreGuidelines** che sono delle linee guida del linguaggio, create dall'inventore del C++ *Bjarne Stroustrup* e da *Herb Sutter*.

Il **CppCoreGuidelines** non è uno standard ISO ufficiale.

Tratterò qui alcune di queste linee guida, ma sarebbe molto difficile trattarle tutte, visto che queste sono veramente tante. (credo più di 500 se non molto di più)

È difficile per me, trattare queste linee guida nello stesso modo esaustivo e con la stessa profondità delle **CppCoreGuidelines**, a volte elencherò solo i punti principali di una linea guida, comunque se volete controllare una specifica linea guida che ho trattato potete trovarla qui: [CppCoreGuidelines](#).

Ovviamente, ce ne sono tante altre buone che però non riuscirò a trattare.

## *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 i commenti che i commenti dovrebbero essere in inglese a differenza di come ho fatto io in questa guida.

Tratterò in primis le linee guida sui nomi e sul layout, presenti nella sezione di supporto: [NL | Naming and Layout rules](#) delle CppCoreGuidelines.

### *NL | Naming and Layout suggestions*

#### *NL.1: Non dire nei commenti quello che può essere semplicemente affermato nel codice*

I compilatori non leggono i commenti. I commenti sono meno precisi del codice. I commenti non vengono sempre aggiornati in modo costante come il codice.

#### *NL.2: Indica l'intento nei commenti*

Il codice dice ciò che è stato fatto, non quello che dovrebbe essere fatto. Spesso l'intento può essere indicato più chiaramente e concisamente che l'implementazione. Se i commenti e il codice sono in disaccordo, entrambi sono probabilmente errati.

#### *NL.3: Mantieni i commenti nitidi*

La verbosità rallenta la comprensione e rende il codice più difficile da leggere diffondendolo nei sorgenti.

**Nota:** Usa l'inglese. Anche se sei madre lingua in un'altra lingua, usa l'inglese in modo che tutti i programmatore possano capire.

#### *NL.4: Mantieni un'indentazione costante*

Per via della leggibilità. Per evitare "sciocchi errori".

**Nota:** Sempre indentare dopo gli if, for, while è una buona idea.

### **NL.5: Evita la codifica delle informazioni sui tipi dei nomi**

Se i nomi riflettono i tipi rispetto alle funzionalità, diventerà difficile cambiare i tipi usati per fornire quella funzionalità. Inoltre, se il tipo di una variabile viene cambiato, il codice che la usa dovrà essere anch'esso modificato. Minimizza le conversioni non intenzionali.

Il seguente è un esempio **ERRATO**:

```
void print_int(int i);
void print_double(double d);
void print_string(const char *);

print_int(3);
print_double(3.69);
print_string("abc");
```

Il seguente è un esempio **CORRETTO**:

```
void print(int i);
void print(double d);
void print(string_view);

print(3);
print(3.69);
print("abc");
```

**Nota:** I nomi con i tipi codificati al loro interno sono verbosi e criptici.

```
printS // print a std::string
prints // print a C-style string
printi // print an int
```

Le tecniche come la *notazione Ungherese* sono state usate nei linguaggi *untyped* (senza tipi), ma è generalmente non necessario e persino dannoso in un linguaggio fortemente staticamente-tipizzato come il C++, perché le notazioni non vengono aggiornate e interferiscono con il buon uso del linguaggio.

### **NL.7: Fai sì che la lunghezza di un nome sia proporzionale alla lunghezza del suo scope (raggio)**

Più grande è lo scope (raggio), più grande è la possibilità di confusione e di "scontri" tra nomi.

### **NL.8: Usa uno stile dei nomi in modo costante**

La costanza nei nomi e nel loro stile aumenta la leggibilità.

**Nota:** Ci sono molti stili, non puoi seguirli tutti, sceglie uno, ma lascia lo stile delle librerie importate con il loro stile originale.

**Esempio:** Lo Standard ISO, usa le lettere minuscole e le cifre, separando le parole con gli underscores (trattini bassi):

- int
- vector
- my\_map

Evita i doppi trattini bassi: \_\_

**Esempio:** Lo Stroustrup: ISO Standard, ma con le lettere maiuscole usate per i tipi e i concetti definiti dall'utente.

- int
- vector
- My\_map

**Esempio:** CamelCase, capitalizza ogni parola in un identificatore multi-parola:

- int
- vector
- MyMap
- myMap

Alcune convenzioni capitalizzano la prima lettera, altre no.

### **NL.9: Usa ALL\_CAPS (tutte maiuscole) solo per i nomi delle macro**

Per evitare di confondere le macro con i nomi che obbediscono allo scope (raggio) e regole del tipo.

**Nota:** Questa regola si applica anche alle costanti non-macro.

Applicazioni per assicurarci che venga rispettata:

- Segnalare tutte le macro con le lettere minuscole.
- Segnala tutte le macro che non sono nella forma ALL\_CAPS.

### **NL.10: Preferire nomi con lo stile\_underscore (con i trattini bassi)**

L'utilizzo degli underscores (trattini bassi) per separare parti del nome è originale dello stile del C e del C++ nella *Libreria Standard del C++*.

**Nota:** Questa regola è un default soltanto se si ha la possibilità di scelta. Spesso, non si ha la possibilità di scelta e si deve seguire uno stile stabilito per coerenza. Il bisogno di costanza sconfigge il gusto personale.

**Esempio Stroustrup:** ISO Standard, ma con le lettere maiuscole per i propri tipi e per i concetti:

- int
- vector
- My\_Map

### **NL.11: Rendi i letterali leggibili**

Per la leggibilità del codice.

**Esempio** Usa i separatori nelle cifre per evitare lunghe stringhe di cifre.

```
auto c = 299'792'458; // m/s2
auto q2 = 0b0000'1111'0000'0000;
auto ss_number = 123'456'7890;
```

**Esempio** Usa suffissi letterali quando necessario per chiarificare.

```
auto hello = "Hello!"s; // a std::string
auto world = "world"; // a C-style string
auto interval = 100ms; // using <chrono>
```

**Nota:** I letterali (literals) non dovrebbero essere sparsi per l'intero codice come "magic constants", ma è comunque una buona idea per renderli chiari quando sono definiti. È semplice fare un typo in una lunga stringa di interi.

Il seguente è un esempio **ERRATO**:

```
#include < chrono >

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

Il seguente è un esempio **CORRETTO**:

```
#include <chrono>

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

Applicazioni per assicurare il rispetto della regola:

- Segnalare le lunghe sequenze di cifre. Il problema è definire "lunghe"; forse 7.

### **NL.15: Usa gli spazi con moderazione**

Troppi spazi rendono il testo largo e distraente.

**Nota:** Gli spazi bianchi ben piazzati sono un aiuto significativo per la leggibilità. Basta non esagerare.

### **NL.16: Usa una convenzionale ordine di dichiarazione della classe**

Un ordine convenzionale di membri migliora la leggibilità.

Quando dichiari una classe usa il seguente ordine:

- tipi: classi, enums, aliases (using)
- costruttori, assegnamenti, distruttore
- funzioni
- dati

Usa l'ordine: **public** prima di **protected** prima di **private**.

```
class X {
    public:
        // interface
    protected:
        // unchecked function for use by derived class implementations
    private:
        // implementation details
};
```

**Esempio** A volte, l'ordine di default dei membri va in conflitto con l'intenzione di separare l'interfaccia pubblica dai dettagli implementativi. In questi casi, i tipi privati e le funzioni possono essere piazzati con i dati privati.

**Esempio ERRATO:** Evita di dichiarare molteplici blocchi di accesso dispersi tra blocchi di dichiarazioni con differenti accessi. Il seguente esempio è **ERRATO**:

```
class X {    // bad
public:
void f();
public:
int g();
// ...
};
```

L'utilizzo di macro per dichiarare gruppi di membri spesso porta a violazioni nelle regole di ordinamento. Comunque, usare le macro oscura quello che è stato espresso.

Applicazioni per il rispetto della regola:

- Segnala le partenze dall'ordine suggerito.

### **NL.17: Usa il layout di K&R**

Questo è il layout originale del C e C++. Preserva bene lo spazio verticale. Distingue bene i diversi costrutti del linguaggi (come le funzioni e le classi).

Questo può essere consultato al seguente link: [PPP Style Guide](#)

**Nota:** Nel contesto del C++, questo stile è spesso chiamato **Stroustrup**.

#### **Esempio**

```
struct Cable {
    int x;
    // ...
};

double foo(int x)
{
    if (0 < x) {
        // ...
    }

    switch (x) {
        case 0:
        // ...
    }
}
```

```

        break;
    case amazing:
    // ...
    break;
    default:
    // ...
    break;
}

if (0 < x)
++x;

if (x < 0)
something();
else
something_else();

return some_value;
}

```

Da notare lo spazio tra `if` e `(`

**Nota:** Usa linee separate per ogni statement, le parentesi di un `if` e il corpo di un `for`.

**Nota:** La `{` per una `class` e per una `struct` non è su una linea separata, ma la `{` per una funzione sì.

**Nota:** Capitalizza il nome dei tipi definiti dall'utente per distinguerli dai tipi della libreria standard.

**Nota:** Non capitalizzare i nomi delle funzioni.

Per applicare questa regola, usa un IDE per la formattazione.

### NL.18: Usa il layout dei dichiaratori del C

Lo stile di layout del C enfatizza l'uso nelle espressioni e nella grammatica, mentre lo stile del C++ enfatizza i tipi. Questo non vale per le references.

#### Esempio

```

T& operator[](size_t);      // OK
T &operator[](size_t);      // semplicemente strano
T & operator[](size_t);      // indeciso

```

### NL.19: Evita nomi che possono essere facilmente fraintesi

Per la leggibilità. Può capitare di confondere parole simili o errate.

**Esempio**

```
int oO01lL = 6; // male

int splunk = 7;
int splonk = 8; // male: splunk e splonk possono essere facilmente
confusi
```

***NL.20: Non piazzare due dichiarazioni sulla stessa riga***

Per via della leggibilità. È davvero semplice lasciarsi sfuggire una dichiarazione quando ce ne sono più di una in una linea.

**Esempio**

```
int x = 8; std::string s = "hello"; // non lo fare
int x = 8; f(x); ++x; // non lo fare
```

***NL.21: Dichiara un nome (solo) per dichiarazione***

Per la leggibilità. Per minimizzare la confusione con la sintassi.

**Nota:** Per i dettagli, guardare ES.10 a pag.397.

***NL.25: Non usare void come argomento***

È verboso, prolioso ed è necessario solo per questioni riguardanti la compatibilità col C.

**Esempio**

```
void f(void); // male

void f(); // meglio
```

**Nota:** Persino Dennis Ritchie (creatore del C) considerava void f(void) un abominio. Lo si poteva fare in C e se lo si fosse vietato avrebbe causato grossi problemi, ma non lo si può usare nel 21º secolo e in C++.

***NL.26: Usa la convenzionale notazione const***

La notazione convenzionale è più familiare alla maggior parte dei programmatore. È importante per la coerenza nei grandi progetti, community

**Esempio**

```
const int x = 7; // OK
int const y = 9; // male
```

```
const int *const p = nullptr;    // OK, constant pointer to
                                constant int
int const *const p = nullptr;    // male, constant pointer to
                                constant int
```

**Nota:** Gli esempi marcati "male" potrebbero sembrare più logici, ma confondono anche più persone, specialmente i principianti.

Come sempre, ricorda che lo scopo di queste regole di layout è la coerenza, la costanza e che l'estetica varia immensamente.

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



Figura 5.1: CppCoreGuidelines

Comunque ne elencherò qualche d'una.

## *Organizzazione*

RELEASE 0.8

Le linee guide sono così suddivise:  
Sezioni Maggiori:

In. Introduction	Per. Performance
P. Philosophy	CP. Concurrency and Parallelism
I. Interfaces	E. Error handling
F. Functions	Con. Constants and immutability
C. Classes and class hierarchies	T. Templates and generic programming
Enum. Enumerations	CPL. C-style programming
R. Resource Management	SF. Source Files
ES. Expressions and Statements	SL. The Standard Library

## *Sezioni Maggiori*

Come detto, riuscirò a trattare solo alcune delle linee guide. Son veramente troppe!

## *In | Introduction*

Lo scopo delle **CppCoreGuidelines** è quello di aiutare i programmati a scrivere codice più semplice, più efficiente, più mantenibile.

Cercare di applicare tutte queste regole non è semplice.

Ma cerca di considerare queste linee guida, questi ideali quando scrivi codice e cerca di approssimarli il più possibile.

## *P | Philosophy*

Questa è una sezione molto generale che riguarda la **Philosophy** (Filosofia) delle linee guida.

Queste sono delle regole non meccaniche, ma regole individuali che riflettono temi filosofici. Senza una base filosofica, i concetti concreti, specifici mancherebbero di razionalità.

### **P.1: Esprimi le idee direttamente nel codice**

La motivazione principale è che i compilatori non leggono i commenti (o i documenti di design correlati) ovviamente e nemmeno tanti programmati. Ciò che è espresso nel codice ha delle semantiche e può essere controllato dai compilatori e altri strumenti. Mi raccomando usare **const** ove necessario. Rende chiaro l'intento che quella variabile, oggetto non deve essere modificato. Se non lo si mette, è il programmatore a dover indovinare se questo serve o no.

### **P.2: Scrivi in ISO Standard C++**

Questo è un insieme di linee guida per scrivere in *ISO Standard C++*.

**Nota:** Usarle non garantisce portabilità.

### **P.3: Esprimi l'intento**

A meno che l'intento del codice è chiaro (con i nomi o i commenti), è impossibile dire se il codice fa quello che dovrebbe.

**Nota:** Di' quello che dovrebbe essere fatto, piuttosto che come dovrebbe essere fatto.

### **P.4: Idealmente, un programma dovrebbe essere staticamente type safe**

Idealmente, un programma dovrebbe essere completamente staticamente (a compile-time) type safe.

Purtroppo, questo non è possibile. Per via di:

- unions
- casts
- array decay
- errori di range
- narrowing conversions (conversioni strette, limitate)

Alcune alternative per queste aree problematiche, per esempio:

- unions | usa **variant** (C++17)
- casts | minimizza il loro utilizzo; i templates possono essere d'aiuto per questo.
- array decay | usa **span** (dal GLSL)
- errori di range | usa **span**
- narrowing conversions | minimizza il loro utilizzo e usa **narrow** e **narrow\_cast**.

### **P.5: Preferire i controlli a compile-time rispetto che a run-time**

La ragione è per via della chiarezza e delle performance. Non hai bisogno di scrivere *error handlers* per errori a compile time.

**Formulazione Alternativa:** Non posticipare a run time quello che può essere ben fatto a compile time.

**Applicazioni:**

- Guarda agli argomenti dei puntatori.
- Guarda ai controlli a run-time per le violazioni di range.

### **P.6: Quello che non può essere controllato a compile time dovrebbe essere controllato a run time**

Lasciare degli errori difficili da scovare in un programma è chiedere per crashes e pessimi risultati.

**Nota:** Idealmente, cerchiamo di prendere tutti gli errori (che non sono nella logica del programmatore) o a compile time o a run time.

### **P.7: Prendere gli errori a run-time in anticipo**

Evita crashes "misteriosi". Evita errori che porterebbero (possibilmente non identificati) a risultati errati.

**Applicazioni:**

- Controlla i puntatori e gli arrays. Controlla i range prima e non ripetutamente.
- Controlla le conversioni: Elimina o segna le *narrowing conversions* (conversioni strette).
- Controlla i valori non controllati che arrivano dall'input.
- Controlla i dati strutturati (oggetti di classi invarianti) che vengono convertiti in stringhe.

### P.8: Non *leakere* nessuna risorsa

Anche solo una piccola crescita nelle necessità di risorse, a lungo andare, esauriscono le risorse disponibili. Molto importante per programmi di vecchia data nel tempo, ma è un comportamento responsabile di ogni sviluppatore.

**Applicazioni:**

- Guarda i puntatori: Classificali in non-proprietari (di default) e proprietari. Quando possibile, rimpiazza quelli proprietari con gli *handles* della libreria standard.
- Guarda per **new** e **delete**.
- Guarda alle funzioni che allocano risorse e restituiscono raw pointers (come fopen, malloc e strdup).

### P.9: Non perdere né tempo né spazio

Tempo e spazio che spendi per ottenere il risultato non è sprecato.  
 “Un altro beneficio di impegnarsi per l’efficienza è che il processo ti forza a capire il problema in più profondità”. Alex Stepanov.

**Applicazioni:**

- Segnala il valore di ritorno inutilizzato da un operatore++ o operatore– in una funzione definita dall’utente e non di default. Preferire l’utilizzo della forma prefissa.

### P.10: Preferire i dati immutabili a quelli mutabili

Perché è più semplice pensare in termini di costanti che di variabili. Qualcosa di immutabile non può cambiare inaspettatamente. A volte gli immutabili permettono una migliore ottimizzazione. Non si può avere *data race* (race condition) su una costante.

**P.11: Incapsula i costruttori disordinati, al posto di diffonderli nel codice**

Il codice disordinato, confusionario è più probabile che nasconde bugs ed è più difficile da scrivere. Una buona interfaccia è più semplice e sicura da usare. Codice disordinato e di basso livello genera più codice di questo tipo.

Applicazioni:

- Cerca del "codice disordinato" come una complessa manipolazione di un puntatore e dei casts fuori dalle implementazioni delle astrazioni.

**P.12: Usa gli strumenti di supporto come è appropriato**

Ci sono molte cose cose sono fatte meglio "dalle macchine". I computers non si stancano e non si annoiano delle tasks ripetitive.

Per esempio:

- Static analysis tools
- Concurrency tools
- Testing tools

**Nota:** Cerca di non diventare dipendente sui tools (strumenti) troppo elaborati o specializzati. Questi potrebbero rendere il tuo codice non portabile.

**P.13: Usa le librerie supportate in modo appropriato**

Usare le librerie ben progettate, ben documentate, e ben supportate salva tempo e impegno; Una libreria largamente diffusa è più probabile che verrà aggiornata e portata a nuovi sistemi.

Le conoscenze di una libreria molto usata possono farti risparmiare tempo sui tuoi futuri progetti. Quindi, se una libreria adatta esiste per la tua applicazione/ per il tuo dominio applicativo, allora usala.

**Nota:** Di default usa:

- ISO C++ Standard Library
- Guidelines Support Library

**Nota:** Se una libreria ben progettata, ben documentata e ben supportata non esiste per il dominio che ti serve, allora dovresti progettarne una e implementarla e poi usarla.

## *I | Interfaces*

Un'interfaccia è un contratto tra due parti di un programma. Di cui, uno di questi è il *supplier* (fornitore) del servizio e l'altro è l'utente del servizio. Affermando con precisione qual è il servizio che si prevede di fornire è essenziale.

Avere delle buone interfacce è probabilmente il singolo aspetto più importante dell'intera organizzazione del codice.

### **I.1: Fai delle interfacce esplicite**

Correttezza. Le assunzioni non preciseate in un'interfaccia vengono semplicemente trascurate, tralasciate e difficili da testare.

### **I.2: Evita variabili globali non costanti**

Le variabili globali non costanti nascondono delle dipendenze e rendono le dipendenze soggette a cambiamenti imprevedibili.

### **I.3: Evita i singletons**

I singleton sono praticamente dei complicati oggetti globali celati

### **I.4: Rendi le interfacce precise e fortemente typed**

I tipi sono la documentazione migliore e più semplice, migliorano la leggibilità per via del loro significato ben definito e vengono controllati al tempo di compilazione. Inoltre, a volte, il codice precisamente "tipato" è spesso meglio ottimizzato.

### **I.5: Precondizioni di stato (se ce ne sono)**

Gli argomenti hanno significati che potrebbero limitare il loro proprio utilizzo nel chiamante.

### **I.6: Preferisci `Expect()` per esprimere le precondizioni**

Per rendere chiaro che la condizione è una precondizione e per consentire l'uso dello strumento.

### **I.7: Postcondizioni di stato**

Per rilevare malintesi riguardo al risultato e possibilmente individuare implementazioni errate.

**I.8: Preferire Ensures() per esprimere postcondizioni**

Per rendere chiaro che la condizione è una postcondizione e per consentire l'uso dello strumento.

**I.9: Se un'interfaccia è un template, allora documenta i suoi parametri usando i concepts**

Rendi l'interfaccia precisamente specificata e controllata a compile-time (non così distante) nel futuro.

**I.10: Usa le eccezioni per segnalare un fallimento nel performare una attività necessaria**

Non dovrebbe essere possibile ignorare un errore perché questo potrebbe lasciare il sistema o la computazione in uno stato indefinito. Questa è una delle principali fonti di errori.

**I.11: Non trasferire mai l'ownership (la proprietà) di un raw pointer ( $T^*$ ) o di una referenza ( $T\&$ )**

Se si è in dubbio se un chiamante o un chiamato possiedono un oggetto, allora questo potrebbe portare a dei leaks o a una prematura distruzione dell'oggetto.

**I.12: Dichiara un puntatore che *non* deve essere `null` come `not_null`**

Per evitare di dereferenzare gli errori a `nullptr`. Per migliorare le performance evitando degli inutili controlli sul `nullptr`.

**I.13: Non passare un array come un singolo puntatore**

Le interfacce stile puntatori sono prone agli errori. Inoltre, un raw pointer ad un array deve dipendere su qualche convenzione per permettere al chiamato di determinare il size.

**I.23: Mantieni il numero di argomenti in una funzione basso**

Avere molti argomenti apre le porte alla possibilità di confusione. Passare molti argomenti è spesso costoso rispetto ad altre alternative.

## *F | Functions*

### **E.21: Per restituire molteplici "out" values, preferire il ritorno di una struttura o di una tupla**

Un valore di ritorno è auto esplicativo, si auto documenta come un valore soltanto di output. Si possono ritornare molteplici valori attraverso le **tuple** (incluso i **pair**) con la convenienza dell'usare **tie**.

Applicazioni:

- Gli "output parameters" dovrebbero essere rimpiazzati dai valori di ritorno. Un "output parameter" è uno a cui la funzione scrive, invoca un membro non costante o passa un non costante.

### **E.51: Se c'è possibilità di scelta, preferisci gli argomenti di default rispetto all'overloading**

Gli argomenti di default forniscono interfacce alternative ad una singola implementazione. Non c'è alcuna garanzia che un insieme di funzioni "overloadate" implementano tutte le stesse semantiche.

L'utilizzo degli argomenti di default può evitare la replicazione del codice.

## *C | Classes and class hierarchies*

### **C.45: Non definire un costruttore di default solo per inizializzare dei dati membri; usa gli inizializzatori della classe piuttosto**

Usare gli inizializzatori della classe permette al compilatore di generare la funzione per te. Le funzioni generate dal compilatore potrebbero essere più efficienti.

**Applicazioni:**

- Un default constructor dovrebbe fare di più che semplicemente inizializzare le variabili membro con delle costanti.

### **C.47: Definisci e inizializza le variabili membro nell'ordine della loro dichiarazione**

Per minimizzare la confusione e gli errori. Nell'ordine dell'inizializzazione succede.

**Applicazioni:**

- La lista di inizializzazione dovrebbe menzionare i membri nello stesso ordine in cui sono stati dichiarati.

### **C.48: Preferisci gli inizializzatori nella classe al posto degli inizializzatori ai membri nei costruttori per gli inizializzatori costanti**

Rende esplicito che lo stesso valore è da usare in tutti i costruttori. Evita la ripetizione.

Evita i problemi di manutenzione. Porta a codice più corto ed efficiente.

**Applicazioni:**

- Ogni costruttore dovrebbe inizializzare ogni variabile membro (o esplicitamente attraverso una chiamata a costruttore o attraverso un costruttore di default).
- Gli argomenti di default in un costruttore suggeriscono che un inizializzatore potrebbe essere più appropriato.

## *Enum | Enumerations*

### **Enum.3: Preferire le classi enum al posto degli enum "scoperti"**

Per ridurre le sorprese: gli enum tradizionali si convertono a int troppo facilmente.

**Applicazioni:**

- Avvertire (Warn, porre un warning) su qualsiasi definizione di un enum che non è in una classe.

## R | *Resource Management*

### R.10: Evita la `malloc()` e la `free()`

Perché non supportano la costruzione e distruzione e non si mescolano bene con `new` e `delete`.

Applicazione per rispettare la regola:

- Segnala l'uso esplicito della `malloc` e della `free`.

### R.11: Evita di chiamare `new` e `delete` in modo esplicito

Il puntatore ritornato dalla `new` dovrebbe appartenere al gestore della risorsa (che può chiamare la `delete`). Se il puntatore ritornato dalla `new` viene assegnato ad un puntatore, l'oggetto può essere "leakato".

Applicazione per rispettare la regola:

- Poni un warning sull'uso esplicito della `new` e della `delete`. Suggerisci l'uso di `make_unique` piuttosto.

## *ES | Expressions and Statements*

### ES.10: Dichiara solo un nome per dichiarazione

Una singola dichiarazione per linee migliora la leggibilità ed evita errori correlati alla grammatica del C/C++. Lascia anche spazio per un più descrittivo commento a fine linea.

Il seguente è un esempio **ERRATO**:

```
char *p, c, q[7], *pp[7], **qq[10]; // male!
```

**Eccezione** structured binding (C++17) è specificamente progettato per introdurre diverse variabili:

```
auto [iter, inserted] = m.insert_or_assign(k, val);
if (inserted) { /* new entry was inserted */ }
```

**Applicazioni per il rispetto della regola:**

- Contrassegna le variabili e le costanti con più dichiaratori (esempio: int\* p, q;).

### ES.46: Evita le conversioni aritmetiche che perdono dati (narrowing, truncating)

Una conversione stretta (narrowing) distrugge le informazioni, spesso in modo inaspettato.

**Applicazioni:**

Un buon analizzatore riesce a rilevare tutte le conversioni strette (narrowing), però segnalarle tutte potrebbe portare a dei falsi positivi. Consigli:

- Segnala tutte le conversioni da numeri con la virgola (floating-point) a numeri interi (forse solo **float → char** e **double → int**).
- Segnala **long → char** (anche **int → char**).
- Considera le conversioni strette per gli argomenti delle funzioni molto sospette.

### ES.50: Non togliere il const col cast

Rende il **const** una menzogna. Se la variabile è dichiarata **const**, modificarla risulterebbe in un *undefined behaviour* (comportamento indefinito).

Il seguente è un esempio **ERRATO**:

```

void f(const int& x)
{
    const_cast<int&>(x) = 42; // MALE
}

static int i = 0;
static const int j = 0;

f(i); // effetto collaterale silenzioso
f(j); // undefined behavior

```

Quando sei tentato a ricorrere al `const_cast` per evitare la duplicazione del codice, [...] preferisci mettere il codice in una funzione e rendila un template così che possa dedurre il `const`. Questo non usa alcun `const_cast`:

```

class Foo {
public:                                     // buono
    Bar& get_bar()           { return get_bar_impl(*this); }
    const Bar& get_bar() const { return get_bar_impl(*
        this); }
private:
    Bar my_bar;

template<class T>                         // buono, deduce se T è const
    o non-const
    static auto& get_bar_impl(T& t)
    { /* the complex logic around getting a possibly-
       const reference to my_bar */ }
};

```

Non fare un grosso lavoro all'interno di un template, potrebbe portare ad un *code bloat* (riconfiamento, codice troppo lungo).

**Eccezione** Potresti voler rimuovere il `const` quando chiami funzioni incorrectamente `const`.

Al posto, preferisci wrappare le funzioni inline `const` per encapsulare il cast in un unico posto.

Applicazioni:

- Segnala i `const_cast`.

## *Per | Performance*

### **Per.2: Non ottimizzare prematuramente**

Codice altamente elaborato per l'ottimizzazione è, di solito, più grande e difficile da modificare piuttosto che del codice non ottimizzato.

### **Per.3: Non ottimizzare qualcosa che non è critico a livello di performance**

Ottimizzare una parte che non è critica per le prestazioni di un programma non ha effetti sulle prestazioni del sistema.

**Nota:** Se il tuo programma spende il 4% del suo tempo di esecuzione nella computazione A e il 40% del tempo sulla computazione B, un miglioramento del 50% sulla A non avrà un grosso impatto come un 5% di miglioramento sulla B.

### **Per.4: Non presumere che del codice più complicato sia necessariamente più veloce**

Il codice semplice può essere molto veloce. Gli ottimizzatori (optimizers) fanno cose meravigliose con del semplice codice.

Il seguente esempio è **ERRATO**:

```
// intended to be faster, but is often slower

vector<uint8_t> v(100000);

for (size_t i = 0; i < v.size(); i += sizeof(uint64_t))
{
    uint64_t& quad_word = *reinterpret_cast<uint64_t*>(&v[i]);
    quad_word = ~quad_word;
}
```

Il seguente esempio è **CORRETTO**:

```
// clear expression of intent, fast execution

vector<uint8_t> v(100000);

for (auto& c : v)
    c = ~c;
```

**Per.5: Non supporre che il codice a basso livello sia necessariamente più veloce di quello ad alto livello**

Il codice di basso livello può, a volte, inibire le ottimizzazioni. Gli *optimizers* qualche volta fanno meraviglie con il codice di alto livello.

**Per.7: Progetta per permettere l'ottimizzazione**

Perché spesso abbiamo bisogno di ottimizzare il design iniziale. Perché un design che ignora la possibilità per un miglioramento futuro è difficile da cambiare.

**Nota:** L'ottimizzazione prematura, si dice, sia la radice di ogni male (Per.2 a pag.399), ma questo non è un motivo per disprezzare le performance.

Non è mai prematuro considerare ciò che rende un progetto suscettibile al miglioramento, e il miglioramento delle prestazioni è un miglioramento comunemente desiderato.

In particolare, quando scrivi una funzione ampia, tieni in considerazione:

- **Il passaggio delle informazioni (Information passing):** Preferisci le interfacce chiare che hanno sufficienti informazioni per un successivo miglioramento delle implementazioni.
- **Compattezza dei dati:** Di default, usa dati compatti, come gli `std::vector` e accedi in un modo sistematico. (Per.19 a pagina 401). Se ti serve una struttura linkata, prova a crearne un'interfaccia.
- **Passaggio degli argomenti e il return:** Fai una distinzione tra dati mutabili e non-mutabili.
- **Astrazione:** Non generalizzare troppo; un design che cerca di soddisfare ogni possibile uso (e abuso) e rinvia ogni decisione di progettazione per dopo (utilizzando indirezioni in fase di compilazione (compile-time) o di esecuzione (run-time)) è di solito un progetto complicato, gonfio e difficile da capire. Generalizza da esempi concreti, preservando le performance mentre generalizziamo. Non generalizzare basandoti sulla mera speculazione di possibili futuri bisogni. L'ideale è zero overhead procurato dalla generalizzazione.
- **Librerie:** Usa librerie con delle buone interfacce. Se non ce ne sono, prova a creartene una da solo e imita lo stile delle interfacce di una buona libreria. Per esempio la *Libreria Standard*.
- **Isolamento:** Isola il tuo codice confusionario/vecchio fornendo un'interfaccia di tua scelta (questo è a volte chiamato *providing a wrapper*). Non permettere a del pessimo design di infiltrarsi, "di far sanguinare" il tuo codice.

**Nota:** Non creare un'interfaccia per poi fermarsi alla prima implementazione e al primo caso d'uso a cui avevi pensato. Una volta completata la prima implementazione, ricontrollala; una volta distribuita, sarà più difficile rimediare agli errori.

**Nota:** La necessità per l'efficienza non implica un bisogno per del codice di basso livello.

Il codice di alto livello non implica lentezza o rigonfiamento.

**Nota:** Le cose hanno dei costi. Non essere paranoico sui costi (i computer moderni sono davvero molto veloci), ma abbi un'idea approssimativa dell'ordine di grandezza del costo di ciò che usi.

**Nota:** Se riesci a pensare ad una sola implementazione, allora probabilmente non hai qualcosa per cui puoi escogitare un'interfaccia stabile.

**Nota:** Questa regola non contraddice la regola Per.2 *Non ottimizzare prematuramente* a pag.399.

**Applicazioni per il rispetto della regola:**

- Forse cercare gli **void\*** negli argomenti delle funzioni potrebbe portare a trovare interfacce che impediscono future ottimizzazioni.

### **Per.11: Sposta la computazione dal run time al compile time**

Per ridurre lo spazio occupato dal codice e il tempo di esecuzione. Per evitare *data races* attraverso l'utilizzo delle costanti. Per "acchiappare" (catch) errori a compile time (e quindi eliminando il bisogno di codice per gestire gli errori).

**Nota:** L'ideale è di non eseguire tutto a compile time. Ovvio che certe computazioni dipendono dagli input e quindi non possono essere spostate a compile time, ma a parte quello è che le complesse computazioni a compile time possono seriamente incrementare i tempi di compilazione e complicare il debugging. È anche possibile rallentare il codice attraverso le computazioni a compile time. Questo è certamente raro, ma fattorizzando un calcolo generale in separati sotto-calcoli ottimali è possibile rendere la cache delle istruzioni meno efficace.

**Applicazioni per il rispetto della regola:**

- Guarda per delle funzioni che potrebbero essere `constexpr`, ma che non lo sono.
- Guarda per funzioni chiamate con tutti argomenti costanti.
- Guarda alle macro che potrebbero essere `constexpr`.

### **Per.19: Accedi alla memoria in modo prevedibile**

Le prestazioni sono molto sensibili alle performance della cache e gli algoritmi della cache favoriscono semplice (di solito lineari) accessi a dati adiacenti.

```
int matrix[rows][cols];  
  
// male  
for (int c = 0; c < cols; ++c)  
for (int r = 0; r < rows; ++r)  
sum += matrix[r][c];  
  
// bene  
for (int r = 0; r < rows; ++r)  
for (int c = 0; c < cols; ++c)  
sum += matrix[r][c];
```

## ***CP | Concurrency and Parallelism***

### **CP.2: Evita le *data races***

A meno che tu lo faccia, non c'è garanzia che funzioni e i sottili errori persisteranno.

**Nota:** In sintesi, se due threads possono accedere allo stesso oggetto contemporaneamente (concurrently, in modo concorrente) (senza sincronizzazione) e almeno uno è uno scrittore (che esegue operazioni non-const) allora avrai una *data race*.

**Applicazioni per il rispetto della regola:**

È possibile usare dei tools oppure è possibile mitigare le possibilità di *data races* così:

- Evita dati globali
- Evita variabili **statiche**
- Un uso più concreto di tipi sullo stack (e non passare troppo in giro i puntatori)
- Usa di più i dati immutabili (literals, constexpr, e const)

### **CP.3: Minimizza la condivisione esplicita di dati scrivibili**

Se non condividi dati scrivibili, non puoi avere *data race*. Meno condivisione (sharing) fai meno possibilità ci sono di dimenticarti di sincronizzare gli accessi (e avere delle *data races*). Meno condivisione fai, meno possibilità c'è di dover aspettare su una lock (quindi le prestazioni possono essere migliorate).

**Nota:** I dati immutabili possono essere condivisi in modo efficiente e sicuro. Non c'è bisogno di locking. Non si può avere una *data race* su una costante. Guarda anche

**CP.mess: Message passing** e la CP.31.

### **CP.4: Pensa in termini di tasks, al posto che in threads**

Un **thread** è un concetto di implementazione, un modo di pensare della macchina. Una **task** è una nozione applicativa, qualcosa che vorresti fare, preferibilmente in concomitanza (in modo concorrente) con altre tasks (attività). È più facile ragionare in termini di concetti applicativi.

#### **Esempio**

```
void some_fun(const std::string& msg)
{
    std::thread publisher([=] { std::cout << msg; });
    // male: meno espressivo
    // e più prone ad errori.
```

```
auto pubtask = std::async( [=] { std::cout << msg;
}) ; // OK
// ...
publisher.join();
}
```

**Nota:** Ad eccezione di `async()`, le funzionalità della libreria standard sono di basso livello, orientate alla macchina, a livello di thread e lock. Questa è una fondazione necessaria, ma dobbiamo cercare di alzare il livello di astrazione: per la produttività, per l'affidabilità e per le prestazioni. Questo è un argomento potente per l'utilizzo di librerie di alto livello e orientate più alle applicazioni (se possibile, costruite sopra la libreria standard).

### CP.8: Non provate ad usare `volatile` per la sincronizzazione

Nel C++, a differenza di altri linguaggi, `volatile` non fornisce atomicità, non sincronizza tra le threads, e non previene il riordino delle istruzioni (nè compilatore nè hardware).

Non ha niente a che vedere con la concorrenza.

**Alternativa:** Usa i tipi `atomic` dove avresti usato `volatile` in altri linguaggi. Usa la `mutex` per gli esempi più complicati.

Guarda la CP.200: Use `volatile` only to talk to non-C++ memory.

## **E | Error Handling**

### **E.1: Sviluppa una strategia per occuparti degli errori il prima possibile in un design**

Una strategia coerente e completa per la gestione degli errori e delle perdite di risorse è difficile da ammodernare in un sistema.

### **E.6: Usa RAII per prevenire i leaks**

I *leaks* sono, di solito, inaccettabili. Il rilascio/liberazione di risorse in modo manuale è prone agli errori. RAII (*Resource Acquisition Is Initialization*) è il modo più semplice e sistematico per prevenire i *leaks*.

**Nota:** Ma cosa facciamo se scriviamo un programma dove le eccezioni non possono essere usate? Ci sono molti miti contro le eccezioni. Qui di seguito solo un paio di buoni motivi:

- Siamo su un sistema così piccolo che il supporto delle eccezioni consumerebbe la maggior parte della nostra memoria 2K.
- Siamo in un sistema hard-real-time e non abbiamo strumenti che ci garantiscano che un'eccezione venga gestita entro il tempo richiesto.
- Siamo in un sistema con tonnellate di codice legacy (vecchio codice) che utilizza molti puntatori in modi difficili da capire (in particolare senza una strategia di proprietà riconoscibile) in modo che le eccezioni possano causare leaks.
- La nostra implementazione dei meccanismi di eccezione del C++ è irragionevolmente scarsa (lenta, consuma memoria, non funziona correttamente per le librerie collegate dinamicamente, ecc.) Presenta reclamo al proprio fornitore dell'implementazione; se nessun utente si lamenta, non si verificherà alcun miglioramento.
- Verremo licenziati se sfidiamo l'antica saggezza del nostro manager.

Solo la prima di queste ragioni è fondamentale, quindi quando possibile, usa le eccezioni per implementare RAII o progetta i tuoi oggetti RAII in modo che non falliscano mai.

Riguardati anche l'utilizzo del `noexcept`.

**E.12: Usa *noexcept* quando esci da una funzione perché una *throw* è impossibile o inaccettabile**

Per rendere la gestione degli errori sistematica, robusta ed efficiente.

**Nota:** Non usare le tradizionali *exception-specifications*, guarda E.30: **Don't use exception specification.**

## *Con | Constants and immutability*

### Con.1: Di default, poni gli oggetti immutabili

È più semplice ragionare in termini di oggetti immutabili, quindi poni gli oggetti a **non-const** solo quando c'è bisogno di cambiare i loro valori. Previeni cambiamenti ai valori in modo accidentale o difficili da notare.

#### Esempio

```
for (const int i : c) cout << i << '\n'; // serve solo per leggere i valori: const
```

```
for (int i : c) cout << i << '\n'; // MALE: serve solo per leggere i valori
```

**Eccezione:** I parametri passati per valore raramente mutano, ma anche raramente vengono passati **const**. Non forzare questa regola per i parametri delle funzioni, per evitare confusione e falsi positivi.

#### Applicazioni per il rispetto della regola:

- Segnala le variabili **non-const** che non vengono modificate.

### Con.5: Usa **constexpr** per i valori che possono essere calcolati a tempo di compilazione (compile-time)

Per avere migliori prestazioni, migliori controlli a compile-time, per avere una valutazione garantita a compile-time, per evitare le possibilità di *race conditions*.

#### Esempio

```
double x = f(2);           // possibile valutazione a run-time
const double y = f(2);     // possibile valutazione a run-time
constexpr double z = f(2); // errore a meno che f(2) possa essere valutata a compile time
```

**Nota:** Guarda la regola F.4: If a function might have to be evaluated at compile time, declare it **constexpr**.

#### Applicazioni per il rispetto della regola:

- Contrassegna le definizioni **const** con inizializzatori di espressioni costanti.

## *T | Templates and generic programming*

### T.1: Usa i *templates* per aumentare il livello di astrazione del codice

Per via della generalità, della riusabilità, dell'efficienza. Incoraggia definizioni coerenti di tipi definiti dagli utenti.

**Esempio errato** Concettualmente, i seguenti requisiti sono sbagliati perché ciò che vogliamo da T è più dei concetti di basso livello: "può essere incrementato" o "può essere aggiunto":

```
template<typename T>
requires Incrementable<T>
T sum1(vector<T>& v, T s)
{
    for (auto x : v) s += x;
    return s;
}

template<typename T>
requires Simple_number<T>
T sum2(vector<T>& v, T s)
{
    for (auto x : v) s = s + x;
    return s;
}
```

Assumendo che Incrementable non supporti l'operazione + e Simple\_number non supporti l'operazione +=, abbiamo implementatori sovralimitati di sum1 e sum2. E, in questo caso, abbiamo perso un'opportunità per generalizzare ulteriormente.

```
template<typename T>
requires Arithmetic<T>
T sum(vector<T>& v, T s)
{
    for (auto x : v) s += x;
    return s;
}
```

Supponendo che *Arithmetic* richieda sia + che +=, abbiamo vincolato l'utente di sum a fornire un tipo aritmetico completo. Questo non è un requisito minimo, ma dà all'implementatore di algoritmi la libertà tanto necessaria e garantisce che qualsiasi tipo aritmetico possa essere utilizzato per un'ampia varietà di algoritmi. Per una maggiore generalità e riusabilità, potremmo anche usare un concetto più generale di contenitore o range invece di impegnarci in un solo contenitore, vettore.

**Nota:** Se definiamo un template per richiedere esattamente le operazioni richieste per una singola implementazione di un singolo algoritmo (ad esempio, richiedendo solo `+=` piuttosto che `=` e `+`) e solo quelle, abbiamo manutentori sovra limitati. Miriamo a ridurre al minimo i requisiti sugli argomenti del template, ma i requisiti assolutamente minimi di un'implementazione sono raramente un concetto significativo.

#### Applicazioni per il rispetto della regola:

- Segnala gli algoritmi che usano requisiti "eccessivamente semplici", come l'uso specifico di certi operatori senza un *concept*.
- Non segnalare la definizione dei concetti "eccessivamente semplici" di per se; potrebbero essere semplicemente elementi costitutivi per concetti più utili.

### T.5: Combina le tecniche OO (Object-Oriented) e quelle generiche per amplificare le loro forze, non i loro costi

Le tecniche OO (Object Oriented, Orientate agli Oggetti) e quelle generiche sono complementari.

**Esempio** Static aiuta la dinamica: usa il polimorfismo statico per implementare interfacce dinamicamente polimorfiche.

```
class Command {
    // pure virtual functions
};

// implementations
template</* ... */>
class ConcreteCommand : public Command {
    // implement virtuals
};
```

**Esempio** La dinamica aiuta la statica: offri un'interfaccia generica, confortevole e legata staticamente, ma invia internamente in modo dinamico, in modo da offrire un layout di oggetti uniformi. Gli esempi includono *type erasure* come con il deleter di `std::shared_ptr` (ma non abusare del *type erasure*).

```
#include <memory>

class Object {
public:
    template<typename T>
    Object(T&& obj)
```

```

: concept_(std::make_shared<ConcreteCommand<T>>(
    std::forward<T>(obj))) {}

int get_id() const { return concept_->get_id(); }

private:
struct Command {
    virtual ~Command() {}
    virtual int get_id() const = 0;
};

template<typename T>
struct ConcreteCommand final : Command {
    ConcreteCommand(T&& obj) noexcept : object_(
        std::forward<T>(obj)) {}
    int get_id() const final { return object_.
        get_id(); }

    private:
    T object_;
};

std::shared_ptr<Command> concept_;

class Bar {
    public:
    int get_id() const { return 1; }
};

struct Foo {
    public:
    int get_id() const { return 2; }
};

Object o(Bar());
Object o2(Foo());

```

**Nota:** In una classe template, le funzioni non virtuali vengono istanziate solo se vengono utilizzate, ma le funzioni virtuali vengono istanziate ogni volta. Ciò può aumentare le dimensioni del codice e potrebbe limitare eccessivamente un tipo

generico creando un'istanza di funzionalità che non è mai utilizzata. Evita questo, anche se la libreria standard ha nel proprio codice questi errori.

### T.120: Usa la metaprogrammazione solo quando ti serve veramente

La template metaprogramming è difficile da ottenere in modo corretto, rallenta la compilazione ed è spesso molto difficile da mantenere. Tuttavia, ci sono esempi reali in cui la metaprogrammazione dei template offre prestazioni migliori rispetto al codice assembly a livello di esperti. Inoltre, ci sono anche esempi reali in cui il *template metaprogramming* esprime le idee fondamentali meglio del codice a runtime. Ad esempio, se si ha bisogno della manipolazione AST in fase di compilazione (ad esempio, per delle operazioni su matrice di piegatura (folding)) potrebbe non esserci altro modo in C++.

**Alternativa:** Se il risultato è un valore, al posto di un tipo, usa una **funzione constexpr**.

**Nota:** Se senti il bisogno di nascondere il tuo *template metaprogramming* in macros, allora sei andato troppo in là.

## CPL | C-style programming

### CPL.1: Preferire il C++ al C

Il C++ fornisce più controllo sui tipi e più notazione. Fornisce più supporto per la programmazione ad alto livello e spesso genera codice più veloce.

#### Esempio

```
char ch = 7;  
void* pv = &ch;  
int* pi = pv;    // non è C++  
*pi = 999;       // sovrascrive sizeof(int) bytes vicino &ch
```

Le regole per il casting implicito da e verso void\* in C sono sottili e non applicate. In particolare, questo esempio viola una regola contro la conversione di un tipo con allineamento più stretto.

#### Applicazioni per il rispetto della regola:

- Bisogna usare un compilatore C++

## SF | *Source Files*

### SF.1: Usa il suffisso .cpp per i sorgenti e .h per le interfacce se il tuo progetto non richiede già un'altra convenzione

È una convenzione di lunga data. Ma la coerenza, costanza è più importante, quindi se il tuo progetto usa un'altra convenzione, segui quella.

**Nota:** Questa convenzione riflette un modello di utilizzo comune: le intestazioni sono più spesso condivise con il C per essere compilate sia come C++ che come C, che in genere utilizza .h, ed è più semplice nominare tutte le intestazioni .h invece di avere estensioni diverse solo per quelle intestazioni che devono essere condivise con il C. D'altra parte, i file di implementazione sono raramente condivisi con il C e quindi in genere dovrebbero essere distinti dai file .c, quindi è, di solito, meglio nominare tutti i file di implementazione C++ qualcos'altro (come .cpp). I nomi specifici .h e .cpp non sono richiesti (solo consigliati come impostazione predefinita) e altri nomi sono diffusi.

Esempi sono .hh, .C e .cxx. Si possono utilizzare quei nomi in modo equivalente. In questo documento, ci riferiamo a .h e .cpp come abbreviazione per i file di intestazione e di implementazione, anche se l'estensione effettiva potrebbe essere diversa. Il tuo IDE (se ne usi uno) potrebbe avere opinioni forti sui suffissi.

#### Esempio

```
// foo.h:  
extern int a; // una dichiarazione  
extern void foo();  
  
// foo.cpp:  
int a; // una definizione  
void foo() { ++a; }
```

foo.h fornisce un'interfaccia a foo.cpp. È meglio evitare l'utilizzo di variabili globali.

#### Esempio errato

```
// foo.h:  
int a; // una definizione  
void foo() { ++a; }
```

#include <foo.h> due volte in un programma lancerà un errore di linker per via di una doppia definizione.

#### Applicazioni per il rispetto della regola:

- Contrassegna i nomi non convenzionali dei files
- Assicurati che .h e .cpp (ed equivalenti) seguano il resto delle regole (della sezione SF).

**SF.4: Includi i files .h nel file prima di altre dichiarazione**

Minimizza le dipendenze e incrementa la leggibilità.

**Esempio**

```
#include <vector>
#include <algorithm>
#include <string>

// ... il codice qui ...
```

**Esempio errato**

```
#include <vector>

// ... il codice qui ...

#include <algorithm>
#include <string>
```

**Nota:** Questo vale sia per i files .h che per i .cpp.

**Nota:** Esiste una discussione per isolare il codice dalle dichiarazioni e dalle macro nei file di intestazione, includendo le intestazioni dopo il codice che vogliamo proteggere (come nell'esempio etichettato "errato"). Tuttavia ciò funziona solo per un file (a un livello): usa quella tecnica in un header incluso con altri headers e la vulnerabilità riappare. Un namespace (uno "namespace di implementazione") può proteggere da molte dipendenze. Protezione completa e flessibilità richiedono i moduli (modules).

**SF.5: Un file .cpp deve includere il file(s) .h che definiscono la sua interfaccia**

Questo permette al compilatore di fare controlli di costanza in anticipo.

**Esempio errato**

```
// foo.h:
void foo(int);
int bar(long);
int foobar(int);

// foo.cpp:
void foo(int) { /* ... */ }
int bar(double) { /* ... */ }
double foobar(int);
```

Gli errori non verranno rilevati fino al tempo di linking per un programma che chiama bar o foobar.

#### Esempio

```
// foo.h:
void foo(int);
int bar(long);
int foobar(int);

// foo.cpp:
#include <foo.h>

void foo(int) { /* ... */ }
int bar(double) { /* ... */ }
double foobar(int); // errore: tipo di ritorno errato
```

L'errore del tipo di ritorno per foobar ora viene rilevato immediatamente quando viene compilato il foo.cpp. L'errore di tipo dell'argomento per bar non può essere rilevato fino al tempo di linking a causa della possibilità di overloading, ma l'uso sistematico dei files .h aumenta la probabilità che venga rilevato in precedenza da parte del programmatore.

#### SF.6: Usa le direttive namespaces per transizione, per librerie fondamentali (come la std) o in uno scope locale (soltanto)

*using namespace* può portare a dei clash (scontri) di nomi, quindi dovrebbe essere usato di rado, con moderazione.

Tuttavia, non è sempre possibile qualificare ogni nome da un namespace nel codice utente (ad esempio, durante la transizione) e talvolta un namespace è così fondamentale e prevalente in un codice, che una qualifica coerente sarebbe prolixa e distraente.

**Esempio** L'uso di *using namespace*; potrebbe portare a scontri con i nomi della libreria standard.

```
#include <cmath>
using namespace std;

int g(int x)
{
    int sqrt = 7;
    // ...
    return sqrt(x); // errore
}
```

**Nota:** Guardare anche la linea guida SF.7 a pagina 416.

**Applicazioni per il rispetto della regola:**

- Contrassegna le molteplici direttive *using namespaces* per diversi namespaces in un singolo file.

### SF.7: Non scrivere *using namespace* in uno scope globale in un header file

Farlo porterebbe via l'abilità di disambiguare efficacemente e di usare alternative.

Rende inoltre le intestazioni #included (incluse) dipendenti dall'ordine in quanto potrebbero avere un significato diverso se incluse in ordini diversi.

**Esempio**

```
// bad.h
#include <iostream>
using namespace std; // male

// user.cpp
#include "bad.h"

bool copy /* ... alcuni parametri ... */; // una
funzione che, guarda caso, si chiama copy

int main()
{
    copy /* ... */; // ora overloads local ::copy and std::copy,
    // potrebbe essere ambiguo
}
```

**Nota:** Un'eccezione è l'utilizzo del namespace std::literals;. Questo è necessario per utilizzare i valori letterali (literals) delle stringhe nei file di intestazione e date le regole - gli utenti sono tenuti a nominare il proprio operatore UDLs ""\_x - così che non collideranno con la libreria standard.

**Applicazioni per il rispetto della regola:**

- Segnala *using namespace* in uno scope globale in un file di intestazione. (header file)

### SF.8: Usa le #include guards per tutti i files .h

Per evitare che i file .h vengano inclusi molteplici volte. Per evitare collisioni dalle #include guards, non limitarti a denominare la guardia con il nome del file. Assicurati di

includere anche una chiave e un buon elemento di differenziazione, ad esempio il nome della libreria o del componente di cui fa parte il file di intestazione.

#### Esempio

```
// file foobar.h:  
#ifndef LIBRARY_FOOBAR_H  
#define LIBRARY_FOOBAR_H  
// ... dichiarazioni ...  
#endif // LIBRARY_FOOBAR_H
```

**Nota:** L'estensione `#pragma once` come tipo di `#include guards` non è standard e non è portabile. Guardarsi la linea guida P.2 a pag.388.

#### Applicazioni per il rispetto della regola:

- Contrassegna i files .h senza le #include guards.

### SE.9: Evita le dipendenze cicliche tra i file sorgenti

I cicli complicano la comprensione e rallentano la compilazione. Complicano anche la conversione per usare i *modules* supportati dalla lingua.

**Nota:** Elimina i cicli. non rimuoverli soltanto usando le `#include guards`.

Il seguente è un esempio **ERRATO**:

```
// file1.h:  
#include "file2.h"  
  
// file2.h:  
#include "file3.h"  
  
// file3.h:  
#include "file1.h"
```

#### Applicazioni per il rispetto della regola:

- Contrassegna tutti i cicli.

### SE.11: Gli header files dovrebbero essere indipendenti (self-contained)

Usabilità, le intestazioni (headers) dovrebbero essere semplici da usare e funzionare quando incluse da per se. Le intestazioni devono encapsulare le funzionalità che forniscono. Evitare che i client di un'header debbano gestire le dipendenze di tale header.

#### Esempio

```
#include "helpers.h"
// helpers.h dipende su std::string e include <string>
```

**Nota:** Non riuscire a seguire questo si traduce in errori difficili da diagnosticare per i clienti di un'intestazione.

**Nota:** Un'header deve includere tutte le sue dipendenze. Prestare attenzione all'utilizzo di percorsi relativi perché le implementazioni C++ divergono sul loro significato.

#### Applicazioni per il rispetto della regola:

- Un test dovrebbe verificare che il file di intestazione stesso venga compilato o che venga compilato un file cpp che include solo quel file di intestazione.

### SE12: Preferire gli #include con le virgolette per i files relativi e i simboli <> per il resto

Lo standard offre flessibilità per i compilatori di implementare le due forme di #include selezionate utilizzando la sintassi angolare (<>) o tra virgolette (""). I fornitori (Vendors) ne approfittano e utilizzano diversi algoritmi e metodi di ricerca per specificare il percorso di inclusione. Tuttavia, la guida consiste nell'utilizzare la forma con le virgolette "" per includere i file esistenti in un percorso relativo al file contenente l'istruzione #include (all'interno dello stesso componente o progetto) e di utilizzare la forma con parentesi angolare <> ovunque, ove possibile. Ciò incoraggia a essere chiari sulla località del file rispetto ai file che lo includono o agli scenari in cui è richiesto il diverso algoritmo di ricerca. Rende facile capire a colpo d'occhio se un'intestazione viene inclusa da un file relativo locale rispetto a un'header della libreria standard o un'header dal percorso di ricerca alternativo (ad esempio un'header da un'altra libreria o un insieme comune di includes).

#### Esempio

```
// foo.cpp:
#include <string> // Dalla libreria standard, richiede la forma con le <>
#include <some_library/common.h> // Un file che non è locale, incluso da un'altra libreria; usa la forma <>.
#include "foo.h" // Un file localmente relativo a foo.cpp nello stesso progetto, usa la forma con le virgolette "".
#include "foo_utils/utils.h" // Un file localmente relativo a foo.cpp nello stesso progetto, usa la forma con le virgolette "".
#include <component_b/bar.h> // Un file nello stesso progetto trovato attraverso il percorso di ricerca, usa la forma <>.
```

**Nota:** Non riuscire a seguire questo si traduce in errori difficili da diagnosticare per via di aver incluso il file sbagliato. Ad esempio, in un caso tipico in cui l'algoritmo di ricerca

#include "" potrebbe cercare prima un file esistente in un percorso relativo locale, poi l'utilizzo di questa forma per fare riferimento a un file che non è localmente relativo, potrebbe significare che se un file viene creato nel percorso relativo locale (ad esempio il file incluso viene spostato in una nuova posizione), ora si troverà prima del precedente file di inclusione e il set di includes sarà stato modificato in modo imprevisto. I creatori di librerie dovrebbero inserire le loro intestazioni in una cartella e fare in modo che i clienti includano tali file utilizzando il percorso relativo #include <some\_library/common.h>.

**Applicazioni per il rispetto della regola:**

- Un test dovrebbe identificare se le intestazioni a cui si fa riferimento tramite "" potrebbero essere referenziate con i <>.

## ***SL | The Standard Library***

### **SL.1: Usa le librerie quando possibile**

Perché risparmia tempo. Non reinventare la ruota. Non replicare il lavoro di altri. Prendi vantaggio dal lavoro degli altri quando apportano dei miglioramenti. Aiuta gli altri quando apporti dei miglioramenti.

### **SL.2: Preferire la libreria standard alle altre librerie**

Più persone conoscono la libreria standard. È più probabile che sia stabile, ben mantenuta e ampiamente disponibile rispetto al proprio codice o a maggior parte delle altre librerie.

### **SL.3: Non aggiungere entità non-standard al namespace std**

Aggiungere al `std` potrebbe cambiare il significato di codice che altrimenti sarebbe conforme agli standard. Le aggiunte al `std` potrebbero scontrarsi con le versioni future dello standard.

### **SL.4: Usa la libreria standard in modo type-safe (sicuro per i tipi)**

Perché, ovviamente, rompere questa regola può portare a dell'*undefined behaviour* (comportamento indefinito), corruzione della memoria, e ogni sorta di brutti errori.

**Nota:** Questa è una meta-regola semi-filosofica, che ha bisogno di molte regole concrete di supporto. Ne abbiamo bisogno come di un ombrello per le norme più specifiche. Sintesi delle norme più specifiche:

**C++ Standard Library sommario:**

- SL.con: Containers
- SL.str: String
- SL.io: istream
- SL.regex: Regex
- SL.chrono: Time
- SLC: The C Standard Library

**SL.con.1: Preferire l'utilizzo di STL *array* o *vector* al posto degli array del C**

Gli array del C sono meno sicuri, e non hanno vantaggi rispetto agli *array* e ai *vector* (della STL). Per gli array a lunghezza fissa, usa *std::array* che non degenerano in un puntatore quando passati ad una funzione e conoscono il loro size. Inoltre, un *std::array* allocato sullo stack, mantiene i suoi elementi sullo stack. Per un array a lunghezza variabile, usa *std::vector*, può, per di più, cambiare il proprio size e gestire l'allocazione di memoria.

**Esempio**

```
int v[SIZE];                                // MALE
```

```
std::array<int, SIZE> w;                    // ok
```

**Esempio**

```
int* v = new int[initial_size];           // MALE, si possiede un
                                         raw pointer
delete[] v;                            // MALE, delete
                                         manuale
```

```
std::vector<int> w(initial_size);        // ok
```

**Nota:** Utilizzare *gsl::span* per le referenze non proprietarie in un container.

**Nota:** Confrontare le prestazioni di un array di dimensioni fisse allocato sullo stack con un *vector* con i suoi elementi nello spazio libero è falso. Si potrebbe anche confrontare un *std::array* sullo stack con il risultato di una *malloc()* a cui si accede tramite un puntatore. Per la maggior parte del codice, anche la differenza tra l'allocazione sullo stack e l'allocazione sullo spazio libero non ha importanza, ma la comodità e la sicurezza del *vector* sì. Le persone che lavorano con codice per il quale questa differenza è importante sono in grado di scegliere tra *array* e *vector*.

**Applicazioni per il rispetto della regola:**

- Segnala la dichiarazione di un array del C all'interno di una funzione o classe che dichiara anche un contenitore STL (per evitare warnings sul codice non STL legacy). Per correggere: modificare almeno la l'array del C in un *std::array*.

**SL.str.1: Usare *std::string* per possedere le sequenze di caratteri**

*string* gestisce correttamente l'allocazione, la proprietà, la copiatura, la graduale espansione e offre una varietà di utili operazioni.

**Nota:** Non dare per scontato che *string* sia più lenta delle tecniche di basso livello senza una misurazione e ricorda che non tutto il codice è critico per le prestazioni.

Ricorda la regola **Per.2: Non ottimizzare prematuramente** a pagina 399.

**SL.str.2:** Usa `std::string_view` o `gsl::span<char>` per riferirti a sequenze di caratteri

`std::string_view` o `gsl::span<char>` forniscono semplice e (potenzialmente) accesso sicuro alle sequenze di caratteri indipendentemente da come queste sequenze sono allocate e memorizzate.

Nota: `std::string_view` (C++17) è read-only (solo per la lettura).

**SL.str.4:** Usa *char\** per riferirti ad un singolo carattere

La varietà di usi di *char*\* nel codice corrente è una maggiore fonte di errori.

Il seguente è un esempio **ERRATO**:

```
char arr[] = {'a', 'b', 'c'};  
  
void print(const char* p)  
{  
    cout << p << '\n';  
}  
  
void use()  
{  
    print(arr); // errore a run-time; potenzialmente molto male  
}
```

L'array `arr` non è una stringa array stile C perché non termina con lo zero terminatore (nul terminator).

Alternative: Vedi anche `zstring`, `string` e `string_view`.

#### Applicazioni per il rispetto della regola:

- Segnala l'uso di [ ] in un *char\**.

**SL.str.12:** Usa il suffisso `s` per le stringhe letterali (string literals) intesi come stringhe della libreria standard

L'espressione diretta di un'idea riduce al minimo gli errori.

## Esempio

```
auto pp1 = make_pair("Tokyo", 9.00); // C-style  
string,double intended?  
pair<string, double> pp2 = {"Tokyo", 9.00}; // un poco  
verboso, prolioso
```

```
auto pp3 = make_pair( "Tokyo"s , 9.00);           //  
    std::string,double // C++14  
pair pp4 = { "Tokyo"s , 9.00};                   //  
    std::string,double // C++17
```

### SL.io.3: Preferire *iostream* per l'I/O (Input/Output)

Le *iostreams* sono sicure, flessibili ed estendibili.

Esempio:

```
// scrivi un numero complesso:  
complex<double> z{ 3, 4 };  
cout << z << '\n';
```

*complex* è un tipo definito dall'utente e il suo I/O è definito senza modificare la libreria *iostream*.

```
// leggi un file di numeri complessi:  
for (complex<double> z; cin >> z; )  
v.push_back(z);
```

### SL.io.50: Evita l'*endl*

Il manipolatore *endl* è più che altro equivalente a '\n' e "\n"; come più comunemente usato, rallenta semplicemente l'output eseguendo un *flush()* ridondante. Questo rallentamento può essere significativo rispetto all'output in stile *printf*.

```
cout << "Hello , World!" << endl;      // due operazioni di  
    output e un flush  
cout << "Hello , World!\n";            // una operazione di  
    output e nessun flush
```

**Nota:** Per l'interazione *cin/cout* (ed equivalente), non c'è motivo di fare una *flush*; questo viene fatto automaticamente. Per scrivere su un file, raramente è necessario fare una *flush*.

**Nota:** A parte la questione (a volte importante) delle prestazioni, la scelta tra '\n' ed *endl* è quasi completamente estetica.

### SL.regex: Regex

`<regex>` è la libreria di espressioni regolari standard del C++. Supporta una varietà di convenzioni di pattern di espressioni regolari.

### SL.chrono: Time

`<chrono>` (definito nel namespace `std::chrono`) fornisce le nozioni di `time_point` e durata insieme a funzioni per l'output del tempo in varie unità. Fornisce orologi per la registrazione di `time_points`.

#### SL.C.1: Non usare `setjmp|longjmp`

Un `longjmp` ignora i distruttori, invalidando così tutte le strategie di gestione delle risorse basate su `RAll`.

**Applicazioni per il rispetto della regola:**

- Contrassegna tutte le occorrenze di `longjmp` e `setjmp`.



Tutto quanto citato sopra fa riferimento alla RELEASE 0.8 delle [CppCoreGuidelines](#). L'ultimo commit, al momento in cui sto scrivendo, nel *master branch* è del 8 Maggio 2022.

# Conclusioni

---

## *Conclusioni*

---

E con questo si conclude questa semplice e breve guida sul C++.  
Ringrazio il lettore per la lettura ed invito a consigliarmi ulteriori argomenti o correzioni  
da aggiungere.

Altri possibili capitoli che potrebbero essere aggiunti:

- Multithreading
- Performance | Clean Code | Competitive Programming
- Design Patterns
- C++23

Alla prossima guida!

Luca