

Programmazione C++

Il C++ è un linguaggio di programmazione di tipo **multiparadigma**, che supporta diversi stili di programmazione, tra cui la *programmazione procedurale, orientata agli oggetti e generica*.

È stato sviluppato da Bjarne Stroustrup negli anni '80 come estensione del linguaggio C, ereditando molte funzionalità, ma introducendo altri concetti come le *classi, ereditarietà, e polimorfismo*.

Il linguaggio ha subito varie evoluzioni nel corso degli anni, con nuove versioni standard come:

- **C++98/03**: prima versione standardizzata
- **C++11/14**: aggiunta di funzionalità moderne come *lambda expression, multitasking, automatic type deduction*
- **C++17**: miglioramenti sintattici e nuove librerie
- **C++20**: introduzione di concept, modulo e operatore di confronto `<=>`

Pipeline di compilazione

Il processo di compilazione del C++ può essere diviso in diverse fasi che trasformano il codice sorgente in un file eseguibile.

Una **dichiarazione** informa il compilatore dell'esistenza di una variabile o funzione, ma non ne fornisce l'implementazione.

Una **definizione** fornisce al compilatore l'implementazione vera e propria della funzione o variabile. Una definizione è sempre anche una dichiarazione, ma non viceversa.

Tutto deve essere dichiarato prima del suo utilizzo. Ogni file sorgente viene compilato separatamente, producendo un file oggetto corrispondente; solo dopo che tutti i file sono stati compilati, il linker li combina nell'eseguibile finale.

Preprocessore

Prende il codice sorgente, e lo elabora per trasformarlo in un testo modificato, noto come **unità di compilazione**. Le sue principali operazioni sono gestite con le **direttive**, che sono istruzioni che iniziano con `#`. Possono essere:

- **#define/#undef**: definiscono o rimuovono macro o tag
 - Sostituisce un identificatore con un valore o del codice. Vengono utilizzate per automatizzare operazioni di "trova e sostituisci" nel codice prima della compilazione

- **#if, #ifdef, #ifndef, #else, #elif, #endif:** permettono di abilitare o disabilitare porzioni di codice in base a condizioni definite
 - Per evitare inclusioni multiple dello stesso file header, è buona norma includere le **guardie**, che evitano che un file venga incluso più volte.


```
#ifndef HEADER
#define HEADER
//content
#endif
```
- **#include:** inserisce file di codice esterno nel file corrente
 - Si usa <> per includere file standard, "" per includere file personali esterni

Compilatore

Prende l'unità di compilazione, e la converte in **codice oggetto**, che è un codice binario intermedio non eseguibile. Ogni file sorgente viene compilato separatamente, generando un corrispondente **file oggetto**. Le operazioni principali sono:

- **Analisi sintattica:** verifica la correttezza della sintassi, controllo dei tipi, e inizializzazione delle variabili
- **Analisi semantica:** identifica e verifica le chiamate a funzioni e simboli esterni
- **Generazione di codice:** crea istruzioni a basso livello, e gestisce l'allocazione della memoria
- **Codice di debug:** se abilitato, il compilatore può includere informazioni di debug nel codice oggetto per agevolare il troubleshooting

Compila un file sorgente alla volta, quindi deve identificare cosa non è nel file compilato in quel momento. Mette dei **segnaposto per il linker**, così sa che la funzione è definita in un altro file. Può anche ottimizzare il codice. I file sono **passati in ordine**, quindi bisogna dichiarare le funzioni prima di usarle.

Linker

Prende tutti i file oggetto generati dal compilatore, e li unisce in un unico file eseguibile. Il processo include:

- **Collegamento delle chiamate a funzione:** il linker associa ogni chiamata a funzione con la sua implementazione. Se non riesce a trovare una funzione, genera errori di tipo *unresolved external symbol*
- **Incorporazione delle librerie:** il linker include le librerie necessarie, e aggiunge eventuale codice extra per rendere il file eseguibile sulla piattaforma di destinazione

File sorgente e header

Nel C++, il codice è organizzato in due tipi principali:

- **File .cpp:** contengono le definizioni delle funzioni, classi e variabili
- **File .h:** contengono le **dichiarazioni** delle funzioni, classi e variabili. Sono inclusi dai .cpp per fornire l'interfaccia di ciò che sarà usata. Questo processo è cruciale perché nel C++ tutto deve essere dichiarato prima del suo utilizzo.
 - o Anche i file h possono usare #include, perché potrebbero voler usare tipi particolari che sono inclusi in altri file .h .

Entry Point

L'entry point di un programma è una funzione globale denominata *main*; è necessaria, e deve essere presente una sola volta all'interno di un qualunque file sorgente con estensione .cpp.

Può essere scritta in diverse forme, ma è essenziale che **ritorni sempre un valore di tipo int**.

Il valore restituito rappresenta lo stato di esecuzione del programma; **0** indica che il programma è terminato con successo, **1** indica che si è verificato qualche forma di errore. Se si vogliono specificare altri tipi di errori, è necessario far ritornare un numero negativo.

Inoltre, può avere due argomenti:

- **int argc:** numero di argomenti passati al programma
- **char *argv[]:** array di puntatori a char con gli argomenti
 - o **argv[0]** è il path dell'eseguibile

Librerie

C++ fornisce una serie di librerie standard che offrono un ampio insieme di funzionalità. Possono essere suddivise in tre categorie principali.

Librerie C standard

Sono ereditate dal linguaggio C, e includono file header come `<math.h>`, `<string.h>`, e altre simili. Gli elementi definiti in queste librerie si trovano nel *global namespace*, il che significa che non è necessario di usare alcun prefisso.

Librerie C++ equivalenti a C

C++ fornisce delle versioni aggiornate delle librerie C, come `<cmath>` e `<cstring>`. Questi header contengono gli stessi elementi definiti sia nel *global namespace* che nel *namespace std*. In questo modo, le funzioni possono essere usate sia con che senza il prefisso `std::`.

Librerie esclusive di C++

Sono pensate specificatamente per C++, e i loro elementi sono definiti esclusivamente all'interno del namespace `std`. Esempi sono `<iostream>`, `<string>`, e altre. Gli oggetti e le funzioni che appartengono a queste devono sempre essere prefissati da `std::`, a meno che non si usi `using namespace std;`.

Standard Template Library

La STL è una componente fondamentale di C++, che offre una serie di strumenti avanzati per la gestione dei dati. Include container come *vector* e *list*, algoritmi generici applicabili a questi container, e iteratori per percorrere ed elaborare gli elementi al loro interno.

Gli header della STL, come `<vector>` e `<algorithm>`, rendono disponibile una vasta gamma di strumenti per la gestione efficiente dei dati, permettendo di implementare soluzioni flessibili ed ad alte prestazioni.

Includono **container**, che sono strutture dati come *vector* e *list*, che facilitano la gestione di collezioni di oggetti; **algoritmi**, che sono funzioni generiche che possono essere applicate ai container (ordinamento, ricerca e manipolazione); **iteratori**, oggetti che permettono di attraversare gli elementi dei container in modo uniforme.

Input e output

Sono gestiti tramite oggetti predefiniti che fanno parte del namespace standard `std`; questi oggetti permettono di comunicare con l'utente attraverso la console, sia per inviare messaggi di testo che per raccogliere input. Per accedere a tali funzionalità, è necessario **includere la libreria `<iostream>`**.

Output

L'invio di informazioni verso lo standard output viene gestito tramite l'oggetto predefinito `std::cout`. Questo oggetto permette di scrivere caratteri e stringhe in output, usando `<<`. Ogni volta che un dato viene inviato, esso viene visualizzato sulla console, facilitando l'interazione tra il programma e l'utente.

```
std::cout << "Message to show" << std::endl;
```

Input

C++ fornisce l'oggetto predefinito `std::cin`, che legge l'input dallo standard input. Viene ottenuto tramite l'operatore di estrazione `>>`, che permette di **acquisire e salvare il valore inserito dall'utente in variabili definite nel programma**. È importante assicurarsi che i dati inseriti dall'utente corrispondano al tipo di variabile in cui si desidera memorizzarli.

Errori

Si utilizza l'oggetto `std::cerr`, che invia messaggi allo standard error stream. Funziona in modo simile a `std::cout`, con la differenza che i messaggi inviati tramite `std::cerr` sono generalmente utilizzati per **segnalare errori**, rendendo più semplice la separazione dei messaggi di output da quelli di errore.

Namespace

Per semplificare il codice, è possibile usare la direttiva `using namespace std;`.

Ogni volta, piuttosto che ripetere continuamente il prefisso `std::`, si può direttamente usare questo.

Files

La gestione dell'input e dell'output su file avviene tramite le classi `ifstream` e `ofstream`, parte della libreria `<fstream>`.

Input in file

`ifstream` (input file streaming) viene utilizzata per **aprire un file in modalità di lettura**, permettendo di estrarre dati da esso. Si scrive all'interno dell'`ifstream` come si scriverebbe all'interno di uno standard input/output. Dobbiamo seguire qualche passaggio particolare:

1. **Aprire l'input file stream** usando il metodo `open(string.c_str())`;
 - a. Si può anche direttamente scrivere all'interno di parentesi, subito dopo la dichiarazione dell'input file stream
2. **Verificare se l'input file stream sia aperto**, usando il metodo `is_open()`; se è aperto, si procede a scrivere il contenuto; altrimenti, **si lancia un'eccezione**
3. **Chiudere l'input file stream** usando il metodo `close()`, una volta finita la scrittura

Di seguito un esempio:

```
ifstream fileInput("dati.txt");  
if (fileInput.is_open()) {
```

```

        ifs >> "hello";
        ifs.close();
    } else {
        throw "Can't read the file";
    }
}

```

Salva in file

`ofstream` (output file streaming) viene utilizzata per **aprire un file in modalità di scrittura**, permettendo di memorizzarci dati. Si scrive all'interno dell'`ofstream` come si scriverebbe all'interno di uno standard input/output. Se il file specificato non esiste, ne creerà uno nuovo; se esiste già, il suo contenuto verrà **sovrascritto**, a meno che non si specifichi la modalità `std::ios::app`, che permette di *aggiungere dati alla fine del file senza eliminare il contenuto* esistente. Dobbiamo seguire qualche passaggio particolare:

1. **Aprire l'output file stream** usando il metodo `open(string.c_str())`;
 - a. Si può anche direttamente scrivere all'interno di parentesi, subito dopo la dichiarazione dell'input file stream
2. **Verificare se l'output file stream sia aperto**, usando il metodo `is_open()`; se è aperto, si procede a scrivere il contenuto; altrimenti, **si lancia un'eccezione**
3. **Chiudere l'input file stream**, usando il metodo `close()`, una volta finita la scrittura.

Di seguito un esempio:

```

ofstream fileOutput("output.txt", ios::app);
if (fileOutput.is_open()){
    ofs << "hello";
    ofs.close();
} else {
    throw "Can't read file";
}

```

Tipi di dati

Esistono vari tipi di dati che si suddividono in diverse categorie. Tra i più comuni abbiamo troviamo i **built-in**, ovvero quelli predefiniti forniti dal linguaggio stesso. Questi tipi includono:

- **Bool**: utilizzato per rappresentare valori boolean, *8 bit come 1 bit*
- **Char**: tipo di dato per rappresentare singoli caratteri, *garantito almeno 1 byte (1 byte = 8 bit)*
- **Int**: tipo di dato numerico intero, *4 byte, garantito 2 byte*
- **Float**: tipo di dato numerico con virgola mobile, ma precisione singola, *4 byte*
- **Double**: come float, ma con precisione doppia, *8 byte*

In aggiunta a questi, esistono i **puntatori** e i **reference**, che sono strumenti avanzati per la gestione della memoria e dei dati. Non contengono direttamente un dato, ma un riferimento al dato vero.

Literali

Quando si lavora con numeri e dati testuali in C++, ci si riferisce spesso ai literali, che sono rappresentazioni dirette di valori. Quindi per esempio, 10 rappresenta un *intero*, 10.5 rappresenta un *valore double*, 10.5f rappresenta un *float*, e 0x0120 rappresenta un *numero in esadecimale*.

Modificatori

C++ permette di estendere o limitare la gamma di valori che un tipo di dato può assumere attraverso modificatori come:

- **Unsigned**: rimuove il segno dai tipi *char* o *int*, permettendo solo valori positivi
- **Long**: estende la precisione per i tipi *int* o *double*, *garantito almeno 32 bit*
- **Short**: riduce la precisione del tipo *int*

Questi modificatori devono essere messi prima di specificare il tipo della variabile.

Dimensioni e range

La dimensione e il range di un tipo di dato possono variare a seconda della piattaforma hardware e software, ma lo standard C++ garantisce delle dimensioni minime: un **char** ha una dimensione di almeno 8 bit; un **int** ha una dimensione di almeno 16 bit; un **long** ha una dimensione di almeno 32 bit.

Si possono determinare le dimensioni dei vari tipi di dati usando l'operatore `sizeof ()`.

Importante negli **array** che la prima cella è sì accessibile tramite un puntatore, **ma non è un puntatore in sé**. Il nome dell'array rappresenta l'indirizzo della prima cella di memoria, ma ogni accesso successivo è gestito tramite un *offset* rispetto a tale indirizzo. Questo perché il puntatore **non è un vero puntatore a memoria dinamica, poiché non punta a un'area modificabile, ma rappresenta un indirizzo fisso**.

Puntatori

Un puntatore è una **variabile che contiene l'indirizzo di un'altra variabile**. Viene dichiarato utilizzando l'asterisco davanti al nome della variabile. Il **deferenziamento di un puntatore**, ovvero l'accesso al valore a cui punta, si fa con l'operatore `*`.

Dal punto di vista architetturale, quando istanziamo una variabile, chiediamo al compilatore di **assegnare una locazione di memoria alla variabile** che abbiamo chiamato nel sorgente.

```
int *p; //non ancora inizializzato
int intero = 123; int *ptr = &intero;
//&intero restituisce l'indirizzo di 'intero'
int v = *ptr; //dereferenziamento: v ha il valore 123
```

Abbiamo anche vari casi “particolari”:

- Un puntatore può puntare, a sua volta, ad un altro puntatore, creando così una catena di indirizzamenti
 - o `int **q = &p;`
- Un puntatore **a void** è un puntatore che può riferirsi a qualsiasi tipo di dato, ma *non può essere dereferenziato senza prima essere castato*. Usualmente, non si usa in C++.
 - o `void *pvi = &intero;`
 - o `*((int*)pvi);` //necessario cast a `int*`

Aritmetica

È possibile svolgere operazioni aritmetiche sui puntatori. Ad esempio, aggiungere un valore a un puntatore sposterà il puntatore di quella quantità di elementi del tipo a cui punta. **Si possono sottrarre due puntatori, ma non sommarli**. Se si va “oltre”, allora si rischia una *segmentation fault*, dove l'OS blocca il programma perché tenterà di accedere ad aree di memoria alle quali non dovrebbe accedere.

```
int *p = &intero;
int *p2 = p+1; //p2 punta all'elemento dopo 'intero'
int diff = p2 - p; //differenza di posizione
```

Punti di forza e limiti

Sono potenti ma rischiosi.

Prima di accedere al puntatore, bisogna assicurare che non sia `nullptr` per evitare errori di runtime, e richiedono di essere **dereferenziati**. Questo permette di *scrivere nella cella puntata*.

```
if (ptr != nullptr)
    *ptr = 10;
```

Bisogna sempre fare attenzione a **rilasciare correttamente la memoria** con `delete` o `delete[]`, per evitare memory leaks.

Ogni puntatore è legato a un solo tipo di dato.

Array

Un array è una **sequenza contigua di elementi dello stesso tipo**, in memoria consecutiva, dove ciascuna cella non è inizializzata. Possiamo semplicemente dichiarare un array, o inizializzarlo sia al momento della dichiarazione, o successivamente, attraverso un ciclo for.

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

Possiamo accedere a ciascuna cella dell'array usando le parentesi quadre [**index**]. Si possono anche **leggere elementi al di fuori della dimensione dell'array** predefinita; questo perché *l'array è in realtà un puntatore nascosto*. Per assegnare direttamente al primo elemento dell'array, si può anche semplicemente usare il suo nome.

C++ supporta anche array con più dimensioni; se statici, allora si possono inizializzare le molteplici dimensioni direttamente in *dichiarazione*, altrimenti, se **dinamico**, allora è necessario farlo attraverso cicli for.

Se impostiamo una dimensione prima, allora sarebbe un **array statico**, dove la dimensione è fissata e non modificabile.

Array e puntatori

In C++, il nome di un array può essere utilizzato come un puntatore al primo elemento dell'array. Tuttavia, non è possibile spostare l'indirizzo di un array invece.

```
int *p = array;  
array++; //errore  
p++; //ok
```

Importante negli **array** che la prima cella è sì accessibile tramite un puntatore, **ma non è un puntatore in sé**. Il nome dell'array rappresenta l'indirizzo della prima cella di memoria, ma ogni accesso successivo è gestito tramite un *offset* rispetto a tale indirizzo. Questo perché il puntatore **non è un vero puntatore a memoria dinamica, poiché non punta a un'area modificabile, ma rappresenta un indirizzo fisso**.

Reference

Una reference è un alias di una locazione di memoria. Deve essere inizializzato al momento della dichiarazione, e non può essere riassegnato. Si ha quindi semplicemente **due nomi per lo stesso dato**.

```
int x = 10;  
int &ref = x; // 'ref' è un riferimento a 'x'
```

```
ref = 20; //modifica 'x', che ora vale 20
```

Punti di forza e limiti

Sono più facili da usare rispetto ai puntatori, ma **non possono essere rese nullptr**. Una volta create, non possono essere riassegnate a un'altra variabile.

Sono preferibili ai puntatori quando *non è necessario modificare l'oggetto puntato, o gestire la memoria manualmente*.

Struct

Le struct permettono di **raggruppare dati di diversi tipi in un'unica entità**. Possiamo anche inserire altre *struct* all'interno di struct esistenti. Per accedere ai membri di una struct, utilizziamo il punto . se abbiamo una *variabile*, o la freccia -> se abbiamo un puntatore.

```
struct Punto {  
    int x; int y; }  
Punto p;  
p.x = 10;  
Punto *ptr = &p;  
ptr->y = 20;
```

Per quanto riguarda la creazione di un oggetto a partire da una struct, possiamo *creare direttamente da una struct*, o è necessario inserire un costruttore all'interno della struct.

Per assegnare le variabili di una struct dal costruttore, si mette `variabile_struct(variabile_argomento)`. Questa si chiama **lista di inizializzazione**.

Per il costruttore di default, sarebbe importante mettere, all'interno di una struttura *private*, il valore di default delle variabili di una struct. Se non si mettessero, il costruttore di default lo lascerebbe non inizializzati con dei valori a caso.

```
struct S {  
    int i; double d;  
    S(){} //default construct  
    S(int i, double d) : i(i), d(d) {} //constructor  
private:  
    int i{ 0 };  
    double dt{ 0.0 };
```

Enum e typedef

Gli **enum** definiscono un insieme di **valori costanti interi** con nomi simbolici, rendendo il codice più leggibile.

```
enum Giorno { lun=1, mar=3, ..., nomeN = valoreN };  
Giorno g;
```

```
g = 90; //errore: invalid conversion from int to giorni
int d = lun; //ok
```

Il **typedef**, invece, permette di creare alias per tipi di dati esistenti. Serve soprattutto per **mascherare i tipi di dato concreti**.

```
typedef vecchio_tipo nuovo_nome;
typedef unsigned long int uli;
uli numero;
```

Costanti

Il modificatore **const** indica che un dato non può essere modificato dopo la sua inizializzazione. Questo vale anche per i puntatori, dove possiamo avere costanti di diversi tipi.

È necessaria l'inizializzazione, è sempre possibile comunque inizializzare una variabile const con una variabile, dello stesso tipo, non const.

```
const int *p = nullptr; //Puntatore a intero costante
int *const p1 = 1; //Puntatore costante a intero
const int *const p3 = 2; //Puntatore costante a intero costante
```

Stringhe

Le stringhe in C++ possono essere gestite in due modi: con le **stringhe C**, ovvero array di caratteri terminati da `\0` (e che hanno solo l'operatore `[]`), o con la **classe string** di C++, la quale fornisce più funzionalità.

```
char strc[10] = "Ciao";
char strl[] = {'G','i','o','r','n','a','t','a','\0'};
std::string s1 = "Hello";
std::cout << s1 << s1.c_str() << std::endl //stampano "Hello"
```

Cast

I cast possono essere **impliciti** o **espliciti**. Sono impliciti quando il compilatore può convertire un tipo in un altro, senza perdita di dati o ambiguità.

Come implicito, possiamo avere `varTipo1 = varTipo2`, mentre come cast esplicito `varTipo1 = (Tipo1)varTipo2`.

Abbiamo invece quattro tipi principali di cast espliciti:

- **static_cast**: per la maggior parte delle conversioni di tipo sicure. È usato per conversioni tra tipi compatibili.
 - o `double d = 9.7;`

- ```
int i = static_cast<int>(d);
```
- **const\_cast**: per rimuovere o aggiungere l'attributo const.
    - o `const int ci = 10;`  
`int *pi = const_cast<int*>(&ci); //rimuove il const`  
`*pi = 20; //indefinito`
  - **reinterpret\_cast**: per conversioni di tipo poco sicure, permette la conversione tra tipi di puntatori non correlati
    - o `int i = 42;`  
`void *v = reinterpret_cast<void*>(&i); //converte int*`  
`in void*`  
`int *pi = reinterpret_cast<int*>(v); //converte in int*`
  - **dynamic\_cast**: per conversioni di tipo in contesti di ereditarietà, viene generalmente usato per il **casting in gerarchie di classi**, dove è coinvolto l'uso di polimorfismo. Funziona **solo** con puntatori, o referenze a classi.
    - o `class Base {virtual voi foo() {} };`  
`class Derived : public Base {};`  
`Base *b = new Derived();`  
`Derived *d = dynamic_cast<Derived*>(b);`  
`if (d) { //cast riuscito, se non riesce ritorna nullptr`  
`}`

## Lista di inizializzazione

Sintassi che permette di inizializzare i membri di una classe o struct **prima che il corpo del costruttore venga eseguito**.

È utile per inizializzare membri const o reference. La sintassi è la seguente:

```
class nomeClasse {
 public:
 Tipo membro1; Tipo membro2;
 //costruttore con lista di inizializzazione
 nomeClasse(Tipo val1, Tipo val2) : membro1(val1),
 membro2(val2) { } };
```

## Variabili

Sono elementi fondamentali della programmazione in C++, utilizzate per immagazzinare e manipolare dati. A seconda di dove e come vengono dichiarate, si possono distinguere diverse variabili.

## Variabili globali

Sono dichiarate **al di fuori di qualsiasi funzione**, e sono accessibili da qualsiasi parte del file in cui vengono definite.

Queste variabili esistono per tutta la durata dell'esecuzione del programma, e **mantengono il loro valore anche dopo che una funzione termina la sua esecuzione**.

Per far sì che una variabile globale sia utilizzabile in altri file, è necessario dichiararla come *extern* nel file header associato. In questo modo, la variabile sarà visibile anche all'interno di altri file sorgente che includono tale header, evitando di dover ridefinirla altrove.

```
int g; //g esiste prima del main, fino al termine del programma
int main(){
 g = 100; //ok
}
```

## Variabili locali

Sono dichiarate **all'interno di funzioni o blocchi di codice e sono visibili solo nel loro ambito di dichiarazione** (scope). Vengono allocate nello stack, una porzione di memoria dedicata alla gestione di dati temporanei.

Una volta terminata l'esecuzione della funzione o del blocco, lo Stack dealloca automaticamente queste variabili, liberando la memoria. Sono anche chiamati **dati automatici**.

## Variabili statiche

Si dichiarano con **static** come keyword. Se una variabile è dichiarata all'interno di una funzione, essa mantiene il suo valore tra diverse esecuzioni della funzione stessa. A differenza delle normali variabili locali, **vengono allocate solo una volta, alla prima esecuzione, e conservano il loro stato per tutta la durata del programma**.

Questo tipo di variabili possono essere utilizzate anche all'interno di classi, dove agiscono come *variabili condivise da tutte le istanze della classe*. Quindi, una variabile di classe static è **unica per la classe stessa**, e non per le singole istanze.

Quando viene dichiarata all'interno di un file, invece, limita al suo ambito di visibilità a quel file, impedendo che altre parti del programma possano accedervi.

### Esempio:

In questo caso, la variabile *count* è inizializzata a 0 solo la prima volta che la funzione viene eseguita. Ad ogni successiva chiamata, il valore non viene reinizializzato ma *incrementato*, consentendo alla funzione di tenere traccia del numero di chiamate.

```
void count(){
 static int count = 0;
 count++;
}
```

## Allocazione dinamica

I dati dinamici si riferiscono alla memoria che **viene allocata manualmente durante l'esecuzione del programma**, piuttosto che in fase di compilazione. **Si usano tramite puntatori.**

Questa memoria viene gestita tramite la keyword **new**, che consente di allocare oggetti o array in un'area della RAM chiamata **heap**. A differenza delle variabili nello Stack, i dati non vengono deallocati automaticamente. È necessario quindi usare esplicitamente **delete** (per eliminare il singolo valore) o **delete[]** (per eliminare l'array).

### Esempio:

```
int* p = new int; //alloca intero nell'heap
*p = 10; //assegna valore 10
delete p; //dealloca memoria
p = nullptr; //per sicurezza, dopo aver deallocato

int* arr = new int[10]; //alloca memoria
arr[0] = 5; //assegna valore al primo
delete[] arr; //dealloca
```

## Stack

Lo Stack è una porzione di memoria usata per gestire variabili locali e chiamate di funzione. È organizzato in modo **LIFO** (last in, first out). È molto veloce, ma limitato in dimensione e adatto solo a dati di vita breve. I dati automatici hanno visibilità e vita locale.

## Heap

È la porzione di memoria utilizzata per l'allocazione dinamica. La memoria allocata nello Heap **non viene gestita automaticamente**, e deve essere liberata con *delete*. È più flessibile, e può gestire grandi quantità di dati, ma è più lento e soggetto a problemi come i memory leak se non gestito correttamente.

## Problemi

Innanzitutto, è importante **non usare funzioni come malloc e calloc**, poiché quelle sono funzioni di C, da non usare in C++ poiché non supportano le classi.

Strumenti come gli **smart pointer**, introdotti in C++11, semplificano il processo di gestione della memoria, automatizzando la deallocazione. Strumenti quindi come `std::unique_ptr` e `std::shared_ptr` si occupano di rilasciare automaticamente la memoria quando non ci sono più riferimenti validi.

È importante evitare problemi come il **dangling pointer**, ovvero puntatori che fanno riferimento a memoria già deallocata, e i **memory leaks**, che possono portare a consumi eccessivi di memoria e a comportamenti imprevisti del programma.

## Make

È un potente strumento per automatizzare il processo di compilazione di codice sorgente, specialmente in progetti C++ che coinvolgono più file.

**Ottimizza la compilazione**, ricompilando solo i file che sono stati modificati, e gestendo in modo efficiente le dipendenze tra sorgenti e file oggetto.

Il funzionamento di make si basa su un file chiamato **Makefile**, che descrive le regole per compilare e collegare i vari componenti del progetto. Queste regole rappresentano il grafo delle dipendenze tra file sorgenti, file oggetto e l'eseguibile finale, permettendo di *capire quali file devono essere rigenerati quando uno dei sorgenti viene modificato*.

Si eseguono vari comandi nel makefile.

Il comando `g++ -c file.cpp` **compila** il file sorgente C++ generando un file oggetto (file.o), che contiene il codice in una forma intermedia ma non ancora eseguibile.

Il comando `g++ file.o -o eseguibile` **collega** uno o più file oggetto generando l'eseguibile finale, specificato con l'opzione -o.

Come esempio:

```
g++ -c -include main.cpp -o main.o
g++ -c -include src/Point.cpp -o Point.o
g++ -c -include src/Rectangle.cpp -o Rectangle.o
g++ -o main main.o Point.o Rectangle.o
```

## Makefile

È un file che **descrive le dipendenze tra file sorgenti, file oggetto ed eseguibile**. È composto da **regole**, ciascuna delle quali descrive come ottenere un **target** a partire dalle sue dipendenze. IMPORTANTE ricordarsi che, per scrivere i comandi, ci vuole un tab prima.

```
target : dipendenza1 dipendenza2...
 comando1
 comando2
```

Ogni target rappresenta un file che si vuole generare, e le dipendenze indicano i file di cui quel target dipende. Le azioni che seguono sono i comandi necessari per creare il target. Un po' di dettagli adesso.

## Variabili

È possibile definire variabili all'interno di un makefile per evitare la ripetizione di informazioni.

```
CXX = g++
CXXFLAGS = -Wall -g
```

CXX è la variabile per il compilatore, mentre CXXFLAGS contiene opzioni da passare al compilatore. Le variabili vengono richiamate con la sintassi **\$(CXX)**.

## Variabili automatiche

Vengono fornite anche variabili predefinite come **\$(@)** (nome del target), **\$(^)** (tutte le dipendenze), **\$(<)** (la prima dipendenza)

```
main: Point.o Rectangle.o main.o
 $(CXX) $(CXXFLAGS) -o $@ $^
 #$(@) = "main", $^ = "Point.o Rectangle.o main.o"
```

## VPATH

Per progetti con file sorgenti distribuiti in diverse directory, **VPATH** specifica le directory dove make deve cercare i file.

```
VPATH=./src:./include
```



## PHONY

La direttiva `.PHONY` indica che un target non è associato a un file fisico, come nel caso del target `clean`.

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

## Debug e ottimizzazione

Make si basa su un **grafo delle dipendenze** per determinare quali file rigenerare. Quando si modifica un file sorgente o un header, make *verifica la data di modifica dei file*, e decide quali target devono essere aggiornati.

Quando si desidera fare il debugging con `gdb`, è necessario compilare con le opzioni `-g -O0` per includere le informazioni di debug e disabilitare le ottimizzazioni.

Per il debug, è utile inoltre usare le **asserzioni**: sono espressioni usate per verificare condizioni a tempo di esecuzione. Se l'asserzione fallisce, il programma termina, facilitando il debug.

## Doxygen

Doxygen è uno strumento per la **generazione automatica della documentazione** a partire dai commenti nel codice sorgente. Inserendo commenti strutturati con una sintassi specifica, è possibile produrre documentazione in formati come HTML, PDF o LaTeX.

Per usare Doxygen, è sufficiente includere **commenti strutturati nel codice utilizzando una sintassi specifica**.

```
/**
 @brief Descrizione concisa
 @param Parametri funzione
 @return Valore di ritorno
 @author Autore del codice
 @file Descrive file e scopo
 @section Sezioni e sottosezioni
 @todo Parti di miglioramenti futuri
*/
```

Doxygen supporta anche la generazione automatica di diagrammi, se abilitato nelle configurazioni. Questi grafici sono utili per visualizzare le relazioni tra componenti complessi del codice, come ereditarietà o dipendenze tra file header.

Per iniziare a usare Doxygen, è necessario creare un file di configurazione usando il comando `doxygen -g`.

Eseguendo, viene generata la documentazione in base al contenuto del codice sorgente, e ai commenti strutturati.

## Valgrind

È uno strumento per analizzare l'uso della memoria dinamica nei programmi C++. Di solito, l'esecuzione viene effettuata con questo comando da Linux: `valgrind ./program`.

Questo produce un **report dettagliato sulla memoria usata dall'heap**; la sezione chiamata *HEAP SUMMARY* mostra il numero di blocchi allocati, e se ci sono perdite di memoria (*memory leaks*). Un concetto importante anche è il **dato perso**, ossia un blocco di memoria non più accessibile perché *il puntatore che lo referenziava è stato smarrito o sovrascritto*. Un blocco di memoria è orfano del suo puntatore, e non c'è più modo di accedervi.

Il tool individua anche **accessi errati alla memoria**, come spostamenti in aree non valide, o tentativi di leggere/scrivere in zone non allocate.

## Esempio

Vediamo questo come esempio.

```
int* ptr = new int[10];
delete[] ptr;
ptr = nullptr;
```

Con un codice del genere, Valgrind non mostrerà alcun errore. Tuttavia, se omettiamo `ptr = nullptr`, Valgrind segnalerebbe che la memoria è stata rilasciata, ma il puntatore punta ancora ad una zona oramai inutilizzata (quindi, dangling pointer).

## Funzioni

Permettono di **suddividere un programma in blocchi logici e riutilizzabili**. Ogni funzione esegue un compito specifico, e può ricevere **parametri** in ingresso, e restituire un **valore** in uscita. L'uso delle funzioni rende il codice più modulare, organizzato e mantenibile.

Si definisce con un *tipo di ritorno*, un *nome*, e una *lista di parametri* (facoltativo).

```
tipo_di_ritorno nome_funzione(par_1 nome_par1, ...) {
 //corpo funzione
```

```
 return valore;
}
```

Se la funzione non restituisce nulla, il tipo di ritorno sarà `void`. Per chiamare la funzione, basta usare il suo nome e passare i valori necessari.

## Passaggi di parametri

Il passaggio di parametri avviene in diverse modalità. Quando si passa un parametro a una funzione, si inizializza una variabile locale nella funzione chiamata.

### Per valore

In questo modo, si crea quindi una **copia indipendente del parametro**. Modifiche all'interno della funzione non influiscono sulla variabile originale.

```
void incrementa(int n){ n++; }
```

Il valore che è stato impostato prima di una chiamata a questa funzione non verrà cambiato, proprio perché non ha un tipo di ritorno nella funzione, e il valore modificato rimane nel blocco della funzione.

È sicuro, ma non permette di modificare il valore originale.

### Per puntatore

La funzione **riceve l'indirizzo di memoria della variabile originale**; questo significa che la funzione **può accedere e modificare direttamente il valore puntato**.

```
void incrementa(int* n){
 (*n)++; //dereferenzia il puntatore e aumenta il valore
 puntato
}
incrementa(&val); //passa l'indirizzo
```

Potente e flessibile, ma richiede attenzione per evitare errori con i puntatori nulli o memoria non valida.

### Per reference

È un modo più intuitivo e sicuro di modificare i parametri. Con le reference, non si lavora con indirizzi o puntatori, ma si passa direttamente un **alias** della variabile originale. Questo

consente alla funzione di modificare la variabile originale senza dover usare la sintassi dei puntatori.

```
void incrementa(int& n) {
 n++; //incrementa direttamente l'originale
}
incrementa(val); //passa una referenza a val
```

È meno flessibile rispetto ai puntatori, poiché una reference non può essere riassegnata, ma è *più facile e sicura*.

## Parametro costante

Il qualificatore **const** può essere usato per indicare che il parametro non verrà modificato.

```
void stampa(const int& n) {
 std::cout << n;
}
```

È spesso utile per passare oggetti complessi tramite referenze senza modificarli, ottimizzando l'uso della memoria.

## Valori di default

I parametri delle funzioni possono essere definiti con valori di default, che vengono utilizzati se l'argomento corrispondente non è specificato durante la chiamata della funzione. Queste inizializzazioni devono essere inserite all'interno del file **.h**, e non all'interno del file **.cpp**.

Questi valori di default possono essere utili perché, se passiamo un valore a una funzione, allora la funzione prenderà il valore che le abbiamo passato; altrimenti, userà quello di default. Possono essere:

- **Valori di default per singolo parametro:** `void f(int i = 1);`
- **Valori di default per più parametri:** `void g(int x = 1, int y = 2);`
- **Valori di default nei parametri multipli con omissione parziale:** `void h(int a = 1, int b = 2, int c = 3);` non è necessario specificare tutti gli argomenti
- **Default sui parametri di tipo puntatore o riferimento costante:** `void f(const int& x = 10);` utile per evitare la copia e mantenere un valore immutabile

## Linee guida

Quando si progetta una funzione, è importante considerare il modo in cui i parametri vengono passati, per garantire l'efficienza e il corretto comportamento.

### Modifica dei dati

Se i dati devono essere modificati all'interno della funzione, è preferibile passarli **per puntatore o per riferimento**. In entrambi i casi, la funzione avrà accesso diretto ai dati originali, permettendo di modificarli.

```
void f(Oggetto *o){
 if (o != nullptr){
 //modifica l'oggetto puntato da 'o'
 }
}
```

Oggetto obj; f(&obj);

Prima di accedere o modificare l'oggetto, è buona pratica controllare che il puntatore non sia nullo. In questo modo, la funzione può alterare l'oggetto originale.

Passandolo come riferimento invece:

```
void f(Oggetto &o){
 //modifica direttamente l'oggetto
}

Oggetto obj; f(obj);
```

Si evita la necessità di controllare la validità del puntatore, dato che un riferimento è sempre valido.

### Dati non modificabili

Se i dati non devono essere modificati dalla funzione, il metodo di passaggio dipende dalle dimensioni di dati.

#### Dati piccoli

Possono essere passati per valore, o valore costante.

In questo esempio, f1a ( ) riceve una copia di i come costante, quindi non può modificarne il valore all'interno della funzione.

```
void f1a(const int v){
 //non lo modifica, lo usa
}
```

In f1b ( ), i viene passato per valore e può essere modificato, ma questa modifica avrà effetto solo all'interno della funzione, e non al di fuori di essa.

```
void f1b(int v){
 //usa e modifica v
}
```

## Dati grandi

Dovrebbero essere passati per puntatore costante, o riferimento costante, per evitare copie inutili, migliorando così le prestazioni.

In questo esempio, `f2 ( )` riceve un puntatore a Oggetto costante, mentre `f3 ( )` riceve un riferimento costante. Entrambe le funzioni possono usare l'oggetto senza modificarlo, garantendo che il dato rimanga intatto.

```
void f2(const Oggetto *o){
 if (o != nullptr)
 //usa 'o' senza modificarlo
}

void f3(const Oggetto &o){
 //usa 'o' senza modificarlo
}
```

## Dati con puntatori

Passando un oggetto per valore, C++ crea una copia dell'oggetto, ma i puntatori all'interno dell'oggetto non vengono copiati. Questo può portare a situazioni pericolose in cui più oggetti puntano allo stesso blocco di memoria.

In questo esempio, se la funzione `f ( )` modifica il contenuto di `elem`, anche l'elemento originale `obj` sarà modificato, poiché **entrambi puntano agli stessi dati**. Questo comportamento può essere difficile da gestire.

```
struct Oggetto {
 int *elem; int n_elem; };

int main(){
 Oggetto obj;
 int buffer[10];
 obj.elem = buffer;
 obj.n_elem = 10;
 f(obj);
}
```

`f (obj )` riceve una copia di `obj`, ma non dei dati puntati da `elem`.

## Funzioni inline

Rappresentano un'**ottimizzazione in cui il compilatore può decidere di sostituire la chiamata a una funzione con il suo codice**, evitando il cambio di contesto (salvataggio dello stato sullo Stack e recupero dei valori). La sostituzione del codice non viene effettuata dal compilatore quando la funzione è troppo grande.

C'è una *riduzione del costo del cambio di contesto* dovuto alla chiamata di funzione, ed una maggiore velocità di esecuzione quando la funzione è piccola.

Lo svantaggio è che ci sarà un *incremento della dimensione dell'eseguibile*, poiché il codice della funzione viene copiato ogni volta che viene chiamata, e una ridotta riusabilità del codice.

```
inline int massimo(int a, int b){
 return (a > b) ? a : b;
}
```

## Overloading

L'overloading permette di **definire più funzioni con lo stesso nome purchè abbiano una firma diversa**. Questo rende il codice più leggibile e flessibile.

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

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

## Funzioni template

Permettono di scrivere codice **generico** e **riutilizzabile**, adattabile a diversi tipi di dati. I template funzionano come *prototipi di codice*, e vengono istanziati con tipi specifici al momento dell'uso.

```
template <typename T1, ...> retval nomeFunzione(parametri);
```

Quando i parametri template appaiono tra gli argomenti della funzione, il compilatore può *dedurli automaticamente*; se i parametri template non appaiono tra gli argomenti, devono essere specificati esplicitamente nella chiamata. Abbiamo quindi due casi per questo.

| Caso 1: i parametri template compaiono come tipi dei parametri della funzione                                                    | Caso 2: i parametri template <i>non</i> compaiono come tipi dei parametri della funzione, o compaiono come tipo di ritorno       |
|----------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------|
| <pre>template &lt;typename T&gt; void Function1(const T &amp;valore) {...}</pre>                                                 | <pre>template &lt;typename T&gt; void Function2(int v){ T temp; ... } template &lt;typename T&gt; T Function3(int v) {...}</pre> |
| <pre>Function1&lt;double&gt;(1.123); Function1(10); //T=int Function1("e"); //T=const Function1&lt;char*&gt;(10); //errore</pre> | <pre>Function2&lt;double&gt;(10); //T=double Function2(1) //errore</pre>                                                         |
| Il compilatore <b>può dedurre</b> il parametro template T, dal tipo del dato passato come parametro; si può specificare comunque | Il parametro template T non è dedotto; è necessario specificarlo nella chiamata                                                  |

Nel caso si abbiano più parametri, è necessario mettere all'inizio della lista quelli che non sono deducibili automaticamente dal compilatore. Quelli che non sono deducibili devono essere sempre specificati comunque.

Le funzioni **cast** sono delle funzioni template, per esempio.

## Esempio

```
//setta i dati di array ad un valore dello stesso tipo
template <typename T>
void set_data(T *array, unsigned int size, const T &value){
 for (unsigned int i = 0; i < size; ++i) array[i] = valore;
}

//setta i dati di un array ad un valore
template <typename T, typename Q>
void set_data (T *array, unsigned int size, const Q &valore){
 for (unsigned int i = 0; i < size; ++i)
 array[i] = static_cast<T>(valore);
}
```


## Classi template

Permettono di definire strutture dati generiche come *vector* o *map*; ogni istanza con parametri diversi viene considerata un tipo distinto.

```
template <typename T> //typename T = int per parametri di default
class ClasseGenerica {
 T valore;
public:
 T getValore() { return valore; } };

```

Tale implementazione è particolare poiché, se definiamo una classe come generica, **dovremmo mettere anche la definizione di essa all'interno del file header**, il che diventerà un file .hpp. Bisogna quindi **copiare** tutte le dichiarazioni delle varie funzioni, cambiare tutti i tipi di dati necessari in template, e *modificare certe dichiarazioni in modo che passi una reference rispetto a una copia*. Solitamente, si riferisce comunque al tipo che è un template, e inoltre bisogna farlo poiché alcuni tipi non supportano la copia.

|                                                |                                                                                                                                                                    |                                                                  |
|------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------|
| <pre>void fill(value_type value) { } ...</pre> | <div style="text-align: center;"><small>typedef T value_type</small><br/></div> | <pre>void fill(<b>const</b> value_type &amp;value) { } ...</pre> |
|------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------|

## Namespace

Un **namespace** in C++ è un meccanismo che permette di organizzare il codice in blocchi logici, evitando conflitti tra nomi. È possibile isolare questi nomi all'interno di uno specifico contesto.



In questo esempio, la variabile `valore` è contenuta nel namespace `MioSpazio`, perciò deve essere richiamata come `MioSpazio::valore`, per evitare conflitti con altre variabili con lo stesso nome.

L'istruzione `using` consente di semplificare

l'accesso agli elementi all'interno di un namespace, senza dover riscrivere il nome completo ogni volta.

```
namespace MioSpazio {
 int valore = 42;
}
int y = MioSpazio::valore;
using namespace MioSpazio;
int x = valore;
```

## Recovery degli errori

Permette di **gestire situazioni eccezionali in modo da prevenire crash del programma**, o comportamenti indesiderati.

## Gestione delle eccezioni

Avviene principalmente tramite il costrutto `try-catch`. Un blocco di codice che può potenzialmente generare un'eccezione viene inserito in questo blocco. Se si verifica un'eccezione, viene *lanciata* usando la parola chiave `throw`, e viene catturata dal blocco `catch`, che segue immediatamente il blocco `try`.

Il metodo `what` della classe base `std::exception` restituisce un *messaggio descrittivo dell'eccezione*, di tipo `const char*`.

```
try {
 int* arr = new int[1000000000000];
} catch (std::exception& e) {
 std::cerr << "Error: " << e.what() << std::endl;
}
```

## Lancia eccezioni

Il meccanismo di gestione delle eccezioni inizia quando *viene lanciata un'eccezione* usando la parola chiave `throw`. Questo **interrompe il normale flusso del programma**, e cerca un blocco `catch` che sia in grado di gestire l'eccezione lanciata.

```
void myFunction() {
 throw std::runtime_error("Errore generico");
}
```

## Propagazione delle eccezioni

Se un'eccezione **non viene gestita** all'interno del blocco try-catch in cui è stata lanciata, viene propagata al livello superiore dello stack delle chiamate, fino a trovare un **gestore**. Se non viene trovata alcuna gestione per quell'eccezione, il programma termina. Viene propagato lungo lo stock delle chiamate. Quindi, molte volte è importante mettere una **throw** all'interno della catch, per non dover propagare al resto dell'app l'eccezione, ma gestirlo lì all'interno.

## Principali eccezioni

- **std::exception**: da cui derivano la maggior parte delle eccezioni
  - o **std::bad\_alloc**: allocazione dinamica di memoria con new fallisce
  - o **std::runtime\_error**: errori generici durante l'esecuzione del programma
  - o **std::out\_of\_range**: accedere a un elemento al di fuori dei limiti di un contenitore
  - o **std::invalid\_argument**: un argomento non valido è stato passato ad una funzione

Bisogna includere un file header, come `<exception>`, `<new>` (per `bad_alloc`) e `<stdexcept>`.

## Classi

Una classe è un tipo di dato definito dall'utente che funge da blueprint per creare oggetti. Definisce i dati, chiamati *membri*, e le funzioni, chiamate *metodi*, che rappresentano il comportamento dell'oggetto. Da esse, si possono creare oggetti.

Le classi in C++ sono simili a quelle in Java dal punto di vista del design e del concetto, ma differiscono per alcune caratteristiche.

- C++ **permette la dichiarazione della classe in un file e l'implementazione dei suoi metodi in un altro.**
- Per avere il polimorfismo, il meccanismo deve essere attivato esplicitamente.
- C++ consente di **ereditare da più classi contemporaneamente**
- C++ consente di differenziare chiaramente tra **metodi che modificano e quelli che non modificano lo stato dell'oggetto**
- C++ permette di **sovraccaricare e ridefinire gli operatori** per adattarli alle esigenze specifiche della classe

Una classe può essere dichiarata in due forme:

- **Dichiarazione combinata:** la dichiarazione della classe e la definizione dei metodi possono essere nello stesso file, con questi componenti:
  - o **Member data:** dati privati della classe
  - o **Member functions:** funzioni che operano sui dati membri di quella classe, ed è legata a un'istanza della classe stessa
  - o **Modifiers:** modificatori di accesso, come public, private o protected
- **Separazione della dichiarazione della definizione:** è possibile dichiarare i membri della classe nel file .h, e implementare le loro funzioni nel file .cpp .

Le funzioni fondamentali sono: **costruttore di default, distruttore, costruttore copia, operatore di assegnamento.**

Public interface, access modifier, member data, member functions

#### DateClass.h

```
class Cdate {
public:
 typedef int date_elem;
private:
 date_elem mDay, mMonth, mYear;
 bool isvalid (date_elem pD, date_elem pM, date_elem pY);
public:
 void init(date_elem pD, date_elem pM, date_elem pY);
 void add_day(date_elem pD); //this whole method also
...
};
```

#### DateClass.cpp

```
#include "DateClass.h"
void Cdate::init(date_elemento d, date_elem pM, date_elem pY) {
 if(isvalid(pD, pM, pY) {mDay=pD; mMonth=pM; mYear=pY; } }
//implementazione di tutti gli altri metodi del file h
```

## main.cpp

```
int main() {
 Cdate d;
 Cdate::date_elem e=10;
 d.init(7,11,2007);
 ...
}
```

Per inserire l'implementazione di un metodo, o per richiamare un metodo o una variabile dell'interfaccia pubblica, è importante mettere il **namespace** con il nome della classe prima.

## Puntatore this

È un puntatore implicito presente in ogni metodo non statico di una classe, che **punta all'istanza corrente dell'oggetto su cui è chiamato il metodo**. Si può usare in due modi principalmente:

- **this->**: si usa per accedere ai membri dell'oggetto corrente
- **\*this**: dereferenzia il puntatore this, e restituisce un riferimento all'oggetto corrente.

## Information hiding

### Constness

Nei metodi delle classi, occorre definire i parametri come **const** quando l'oggetto o il metodo non altera lo stato della classe.

Data questa limitazione:

- Se si vuole che una funzione membro sia chiamabile su un oggetto const, allora la funzione deve essere dichiarata *const*
- In una funzione *const*, tutte le variabili membro della classe sono viste **read-only**
- Una funzione *const* può usare solo funzioni const
- I costruttori sono sempre usabili
- Di un oggetto *const* si possono usare solo metodi const

```
int getX() const { return x; } //non modificante
void setX(int pX) { x = pX; } //modificante
```

## Modificatori di accesso

Come in Java e in qualsiasi altro linguaggio, controllano la visibilità e l'accesso ai membri della classe:

- **Public**: membri accessibili anche all'esterno della classe
- **Private**: membri accessibili solo all'interno della classe

- **Protected:** membri accessibili solo dall'interno della classe e delle sue classi derivate
- **Friend:** concede a una funzione o a un'altra classe l'accesso ai membri privati e protetti di una classe

## Costruttore

Un costruttore è una funzione speciale chiamata per inizializzare un nuovo oggetto della classe. Un costruttore può essere un **costruttore di default**, che viene chiamato automaticamente quando viene creato un oggetto senza parametri, e un **costruttore con parametri**, che può inizializzare i dati membro con valori specificati.

Quando un oggetto viene creato, allora *viene allocata la memoria grezza, vengono chiamati i costruttori dei membri dati, e viene chiamato il costruttore dell'oggetto*. Questo è quello che succede **nell'ultima riga del codice d'esempio** scritto qua a fianco.

Di seguito i due tipi di costruttori, con le loro chiamate:

### Implementazione



Deve sempre esistere un qualche tipo di costruttore; in assenza, *il compilatore crea automaticamente quello di default*, che non fa nulla di specifico. È possibile **non avere un costruttore di default**, ma solo se esistono altri costruttori.

Nella creazione di array di oggetti, viene chiamato **solo il costruttore di default per ogni elemento**; semplicemente, non è possibile usarne altri.

I costruttori con **un solo parametro** possono essere utilizzati dal compilatore per fare delle conversioni di tipo implicite, tipo tra diversi tipi di oggetto. Se non si vuole questa semantica, bisogna premettere la keyword **explicit** ai costruttori.

```
//.h
class Cdate {
public:
 //default
 Cdate(); ...
Private: ... };

//.cpp
//definizione costruttore
Cdate::Cdate() {
 mDay=1;...}
void f(){ CDate date;
}
```

## Costruttore di copia

È utilizzato per inizializzare un nuovo oggetto come **copia di un altro oggetto esistente della stessa classe**, ad esempio

passando un oggetto **per valore** a `ClassName(const ClassName &other);`  
una funzione, restituendo un oggetto  
da una funzione, o semplicemente assegnando un oggetto a un altro al momento della dichiarazione.

Per default, C++ fornisce un costruttore di copia implicito che copia i membri dell'oggetto sorgente in quelli dell'oggetto destinazione *bit a bit*, quindi senza eseguire alcuna operazione utile specifica, e richiamando ricorsivamente i costruttori di copia delle variabili membro.

Se la classe gestisce risorse che non possono essere semplicemente copiate, è necessario quindi effettuare una **deep copy**, dove vengono duplicate anche le risorse a cui i membri puntano, creando copie indipendenti per evitare che due oggetti puntino alla stessa risorsa, il che potrebbe causare problemi di gestione delle risorse.

Ad esempio:

```
ClassName(const ClassName &other){
 ptr = new int(*other.ptr); }
```

## Initialization list

È una sintassi usata nei costruttori per **inizializzare i membri di una classe prima che venga eseguito il corpo del costruttore**. In caso di costanti membro o oggetti membro che non hanno un costruttore di default, un'*Initialization list* è *obbligatoria*, in quanto non sarebbe possibile utilizzare l'assegnamento nel corpo del costruttore.

```
class TrigOps {
 const double PI;
public:
 TrigOps(void) : PI(3.14) { ... }
};
```

## Definizione di tipi di variabili

Per evitare che altri programmatori o l'utente capisca quali tipi di variabile il programma usa internamente nei suoi metodi, è utile **definire i tipi di variabile** con un tipo di default, in modo da nascondere questo dettaglio. Per esempio:

```
typedef int value_type; //instead of directly using int everywhere
```

Ovviamente, il typedef può essere poi utilizzato sia nel file .h e nel file .cpp.

## Distruttore

Il distruttore è chiamato automaticamente quando un **oggetto esce dallo scope o viene deallocato**. Il suo ruolo principale è deallocare eventuali risorse allocate dall'oggetto, specialmente se sono stati utilizzati puntatori per allocare memoria sullo heap. Il suo nome deve sempre iniziare con ~.

```
~CDate(void);
CDate::~CDate() {}
```

Quello che esegue, passo per passo, quando un oggetto esce dallo scope, è che *viene effettuata la chiamata al distruttore, viene effettuata la chiamata ai distruttori dei dati membro, viene effettuata la deallocazione*, ed infine **l'oggetto non è più disponibile**.

Come nel caso dei costruttori, se non è definito un distruttore, **il compilatore ne crea uno che non fa nulla**. È importante **mai chiamare il distruttore esplicitamente**.

## Pattern R.A.I.I.

Il pattern R.A.I.I. (*Resource Acquisition Is Initialization*) è una tecnica di gestione delle risorse utilizzata principalmente in C++ per **garantire un controllo sicuro e automatizzato sulle risorse**.

Il principio alla base di RAII è che **l'acquisizione di una risorsa viene associata alla durata di vita di un oggetto**. Quando un oggetto RAII viene creato, **acquisisce la risorsa necessaria durante la sua inizializzazione**. In seguito, quando l'oggetto esce dallo scope, il suo **distruttore** viene automaticamente chiamato e **rilascia la risorsa acquisita**. Questo approccio evita *perdite di risorse* e semplifica il codice, eliminando la necessità di gestire manualmente il rilascio delle risorse.

## Operatori

### Assegnamento

L'operatore = consente di assegnare i valori di un oggetto a un altro **dello stesso tipo**. Deve essere definito come membro della classe, e se non viene esplicitamente dichiarato, il compilatore ne genera automaticamente uno (questa si chiama *memberwise assignment*).

Definendone uno personalizzato, è importante **gestire il self-assignment** per perdere la perdita di risorse; senza ciò, si rischia di liberare risorse prima di riutilizzarle, causando errori o comportamenti indesiderati.

```
ClassName &ClassName::operator=(const ClassName &rhs) {
```

```

 if (this != &rhs) { //controllo self-assignment
 //lavoriamo su un tmp piuttosto che l'originale
 ClassName temp(rhs);
 this->swap(temp);
 }
 return *this;
}

```

## Flusso di output

L'operatore << si può sovraccaricare in una classe, e in questo caso bisogna definire una funzione *friend* o *non membro*, che accetta un oggetto di tipo `std::ostream`, per l'output, e un oggetto della propria classe, e restituisce un riferimento al flusso di output. Un esempio potrebbe essere questo.

```

friend std::ostream& operator<<(std::ostream& so, const MyClass&
obj) {
 so << "bla" << obj.x << "bla" << obj.y;
 return so;
}

```

## Accesso ad array

L'operatore [] può essere sovraccaricato, per permettere l'accesso agli elementi di una struttura dati personalizzata. Bisogna quindi definire un metodo membro che si comporti come un accedere a un elemento, usando un *indice* e restituendo un riferimento all'elemento in modo che possa essere modificato. Un esempio potrebbe essere questo.

```

int& operator[] (size_t index) {
 assert(index < size);
 return arr[index]; }

const int& operator[] (size_t index) const { //versione sola
lettura
 assert(index < size);
 return arr[index]; }

```

È dichiarato come **friend** all'interno della classe, così può accedere direttamente ai membri privati di MyClass. Alternativamente, si può definire l'operatore fuori dalla classe, e utilizzare getter pubblici se non si vuole usare *friend*.



## Funtori

Sono oggetti in C++ che si comportano come funzioni grazie al sovraccarico dell'operatore di chiamata `operator()`. Le sue istanze quindi possono essere invocate come se fossero funzioni. Si usano per specificare una **policy** dell'algoritmo generico. Sono vantaggiosi per:

- **Personalizzazione degli algoritmi:** è possibile definire comportamenti su misura per l'elaborazione dei dati
- **Riutilizzo del codice:** i funtori, combinati con i template, permettono di implementare algoritmi generici adattabili a più contesti
- **Adattabilità:** per uno stesso tipo di dato, si possono implementare strategie multiple, superando i limiti degli operatori predefiniti
- **Stato:** possono avere variabili membro che mantengono uno stato, cosa che una semplice funzione non può fare facilmente. Possono essere passati come argomenti ad altre funzioni.
  - o Molte volte vengono implementati all'interno di una **struct**; sono utili quindi perché, in questo modo, si possono avere diversi modi per confrontare gli stessi oggetti, usando la stessa sintassi.

Di seguito, un esempio:

```
class Moltiplicatore {
public:
 Moltiplicatore(int fattore) : fattore_(fattore) {}
 int operator() (int valore) const {
 return valore * fattore_;
 }
private:
 int fattore_;
};

int main() {
 Moltiplicatore moltiplicaPer2(2); // crea funtori che
 moltiplica per due
 std::cout << moltiplicaPer2(5); return 0; } // = 10
```

Di seguito, un esempio di diverse implementazioni per il confronto di variabili:

```
struct compare_pippo{
 bool operator()(const pippo &p1, const pippo &p2){
 return p1.i < p2.i;
 } };

struct compare_pippo2{
 bool operator()(const pippo &p1, const pippo &p2){
 return p1.d < p2.d;
 } };
```

## Keyword static

Si possono applicare sia all'interno di una classe che di una funzione.

Quando viene utilizzata con una variabile membro di una classe, essa diventa condivisa tra tutte le istanze della classe, il che significa che **ogni oggetto della classe non avrà una copia separata della variabile ma condivide la stessa**.

La variabile **static** esiste indipendentemente dalle istanze della classe e può essere riferita, se dichiarata *public*, qualificandola con il nome della classe. Lo si accede usando il namespace, come `ClassName::static_variable`.

Le **funzioni statiche** di una classe sono *funzioni che non dipendono da una specifica istanza della classe*, ma sono legate alla classe stessa. Non può accedere ai membri non statici della classe, e **una funzione static può chiamare solo altre funzioni e accedere a variabili statiche della classe**. L'uso di `static` nelle funzioni membro ha senso solo se almeno una di queste funzioni è pubblica, in quanto consente l'accesso a variabili e metodi senza la necessità di stanziare un oggetto.

## Classi innestate

Sono classi dichiarate all'interno di un'altra classe. Queste classi possono essere utilizzare per **raggruppare logiche e funzionalità** che sono strettamente legate alla classe contenitore.

Una classe innestata può accedere ai membri della classe esterna, ma il suo utilizzo dipende dalla visibilità dei membri.

Possono essere:

- **Statiche** non possono accedere direttamente ai membri non statici della classe esterna
- **Non statiche** possono farlo, visto che sono una classe che ha una relazione stretta con l'oggetto della classe esterna

Inoltre poi, se è dichiarata **public** o `protected`, può essere accessibile direttamente dall'esterno, qualificandola con il nome della classe contenitore, ma solo nei contesti consentiti dalla visibilità.

```
// .h
class X {
 int i,j;
 class Y {
 int k,1;
 void fy(X &px);
 public: ... };

int main(){
 X::Y cy;
 cy.fx(...); ...
}
```

```
public:
 void fx(Y &py); };
```

## Getter e setter

I getter e setter sono metodi utilizzati per **accedere e modificare i membri privati di una classe**, seguendo il principio di incapsulamento.

Un **getter** è un metodo che consente di leggere il valore di una variabile membro privata, mentre un **setter** è un metodo che consente di modificarne il valore. Abbiamo varie implementazioni che possiamo usare in C++, in base a quale stile di programmazione vogliamo seguire.

## Metodo java

Il metodo java è classico, dove abbiamo un metodo get e set, scritto proprio come in Java. Di solito, sono scritti in questo modo.

```
type get_value(parameter) const {
 return value; }
type set_value(parameter){
 value = parameter; }
```

## Metodo C++

In C++ si preferisce un metodo di scrittura molto più consolidato, semplice e pulito, **un unico metodo che può essere sia getter che setter**. Se il parametro è fornito, allora agisce come un setter; altrimenti, se non è fornito, semplicemente agisce come un getter. Se si vuole che sia soltanto un getter ma con la stessa sintassi, è necessario mettere la keyword *const* in fondo alla firma.

```
type &value(parameter) const{
 return buffer[index]; }
```

## Metodo swap

È un metodo fondamentale per ciascuna classe, utilizzato per **scambiare i valori di due variabili**. Questo può essere molto utile in vari contesti, come nel caso di algoritmi di ordinamento, o per gestire risorse in modo efficiente. È definito come segue.

```
void swap(MyClass& other){
 std::swap(this->a, other.a);
 std::swap(this->b, other.b); }
```

## Composizione

La composizione consente di **costruire oggetti tramite la combinazione di oggetti più semplice**. Si tratta di una relazione del tipo *has-a*.

Ad esempio, una classe `Person` può essere composta da oggetti di tipo `Address` e `PhoneNumber`.

```
class Address { /*...*/ };
class PhoneNumber { /*...*/ };
class Person {
public:
 //metodi
private:
 std::string name; Address address;
 PhoneNumber voiceNumber; };
```

## Ereditarietà

L'ereditarietà stabilisce una relazione **is-a** tra classi; una classe derivata può utilizzare i metodi e i membri della classe base, permettendo la riutilizzazione del codice.

```
class People { /*...*/ };
class Student : public People { ... };
```

Ci possono essere vari tipi:

- **Public inheritance**: quando una classe derivata eredita pubblicamente da una base, i membri pubblici e protetti della base rimangono accessibili nella derivata con lo stesso livello di visibilità.
- **Private inheritance**: i membri pubblici e protetti della classe base diventano privati nella derivata.

| Public inheritance                                                   | Private inheritance                        |
|----------------------------------------------------------------------|--------------------------------------------|
| Membri pubblici e protetti: <b>accessibili con lo stesso livello</b> | Membri pubblici e protetti: <b>privati</b> |

## Name hiding

Se una classe derivata *ridefinisce* od effettua *overloading* di un metodo della base, i metodi della base non saranno accessibili direttamente *senza specificare il nome della classe base*.

```
class Base {
public:
 void g() { ... }
```

```
void g(int, int) { ... } };
```

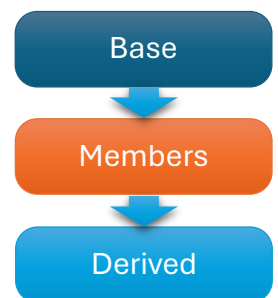
```
class Derived : public Base {
public:
 void g(int) { ... } };
d.g(9);
d.g(); //errore: no matching function
d.Base::g(1,2);
```

## Costruttori

Quando viene chiamato il costruttore di una classe derivate:

1. *La parte base* viene costruita per **prima**
2. dopo vengono costruiti i **dati membro della classe derivate**
3. per ultimo viene eseguito il **costruttore della classe derivate**.

Se non specificato, il compilatore chiama automaticamente i costruttori di default della classe base e degli oggetti membro; quelli delle classi base vengono chiamati in cascata a partire da quello più lontano. I costruttori secondari della classe base o degli oggetti membro devono essere chiamati esplicitamente usando l'initialization list.



## Copy constructor

Se non è definito nella classe derivata, allora è automaticamente definito dal compilatore, e contiene:

- Chiamata automatica al copy constructor della classe base
- Chiamate automatiche ai copy constructor dei dati membro nella derivata

Se invece è **definito nella classe derivata**, sono necessarie chiamate esplicite, nell'initialization list, ai vari copy constructor della classe base e oggetti membro, altrimenti vengono usati i constructor di default.

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

## Operatore di assegnamento

Se non è definita nella classe derivata, allora è automaticamente definito, e contiene:

- Chiamata all'op= della classe base
- Chiamate agli op= dei dati membro della classe derivata

Se invece è **definito nella classe derivata**, è necessario gestire esplicitamente gli assegnamenti.

```
class Derived : public Base{
 Member m;
public:
 Derived &operator=(const Derived& other) {
 if (this != &other) {
 Base::operator=(other);
 m = other.m; }
 return *this;
 }
};
```

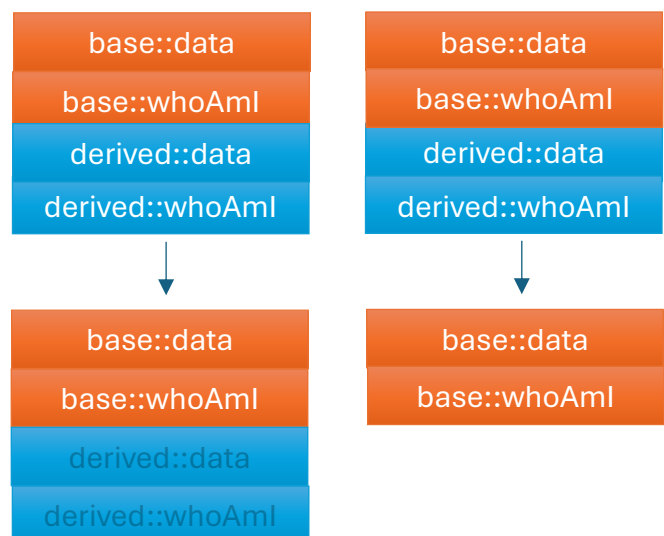
## Upcasting

L'upcasting viene effettuato quando succede un **passaggio da un tipo specifico a un tipo generale**, usando quindi un oggetto derivato come se fosse quello base.

È sempre sicuro e non richiede alcuna operazione esplicita, poiché il compilatore lo gestisce automaticamente. Tuttavia, quando si effettua l'upcasting, le funzionalità specifiche della classe derivata **non sono direttamente accessibili**, a meno che non si utilizzi il downcasting.

L'upcasting si fa con **reference/pointer**. Se lo si facesse per valore, passandolo direttamente, succederebbe lo **slicing**, dove i contenuti "extra" vengono rimossi, portando a perdite di dati e funzionalità.

```
class base {
 int data;
public:
 void whoAmI() { ... } };
class derived : public base {
 int data;
public:
 void whoAmI() { ... } };
void ExecuteIt(base &b){ }
void ExecuteIt(base b){ }
```



## Downcasting

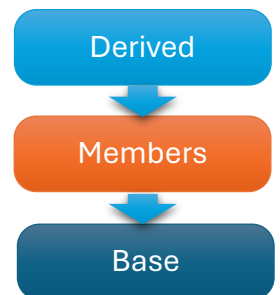
È il processo inverso dell'upcasting: consiste nel **convertire un oggetto della classe base in un oggetto della classe derivata**. Tuttavia, il downcasting è **potenzialmente pericoloso**, poiché non esiste garanzia che l'oggetto della base sia effettivamente un'istanza della derivata. Quindi, **non funziona con classi non polimorfe**.

```
int main(void) {
 Instrument *I = new Harp; Instrument::Instrument()
 ... Harp::Harp()
 delete I; Instrument::~~Instrument()
 ...
 Harp *h = new Harp; Instrument::Instrument()
 ... Harp::Harp()
 delete h; Harp::~~Harp()
} Instrument::~~Instrument()

```

## Distruttori

I distruttori sono sempre chiamati automaticamente in ordine automaticamente **in ordine inverso** rispetto ai costruttori: parte quindi da quello della *classe derivata*, poi *gli oggetti membro*, e poi *quelli della classe base a partire da quella più vicina*.



## Polimorfismo

Permette di utilizzare metodi definiti in una classe base, ma con *implementazioni specifiche nelle classi derivate*. Questo è reso possibile tramite il meccanismo di **funzioni virtuali**. Possono essere **polimorfismo statico** (compile-time), come l'overloading di funzioni od operatori, o **polimorfismo dinamico** (runtime), ottenuto tramite funzioni virtuali e derivazione.

La parola chiave `virtual` rende il metodo polimorfo; consente alle derivate di fornire una propria implementazione tramite overriding. Una volta definito `virtual`, il metodo è definito in questo modo in **tutte le classi derivate**.

```
class Instrument {
public:
 Instrument(void) {
 cout << "Instrument::Instrument()" << endl; }
 virtual ~Instrument::Instrument(void) {
 cout<<"Instrument::~~Instrument()"<<endl; } };
class Harp : public Instrument {
public:
 Harp(void) {
 cout<<"Harp::Harp()"<endl; }
 ~Harp(void) {
```

```

 cout<<" Harp::~Harp()"<<endl; } };
int main(void) {
 Instrument *i = new Harp;
 delete i; //il distruttore di Harp viene chiamato
 correttamente
}

```

### Metodo virtuale puro

Una classe può contenere metodi virtuali puri, dichiarati con la sintassi `virtual type nome(param)=0;`. In questo modo, la classe con metodi virtuali puri è una **abstract class**, ed è quindi impossibile da istanziare, ma è usabile come una pura interfaccia per le classi derivate; quindi, può anche definire un metodo virtuale puro, ma rimane non istanziabile. Solo le classi derivate che fanno override sono istanziabili.

## Iteratori

È un concetto centrale nella libreria standard C++ per l'accesso e la manipolazione di sequenze di dati all'interno di un container (*vector*, *list*, *map*). Sebbene gli iteratori non siano veri e propri puntatori, offrono una semantica e sintassi simile, permettendo operazioni tipiche di un puntatore come `*`, `->`, `++`. Consentono di **interagire con i dati senza rilevarne l'effettiva struttura**, promuovendo l'indipendenza dall'implementazione. Ci sono cinque categorie principali:

- **Output iterator**, per scrivere valori in una sequenza
- **Input iterator**, per leggere valori da una sequenza
- **Forward iterator**, per leggere/scrivere in sequenze iterabili una sola volta in avanti
- **Bidirectional iterator**, permette l'iterazione sia in avanti che all'indietro
- **Random access iterator**, supporta l'accesso casuale, come un puntatore, a qualunque elemento della sequenza

Spesso è dichiarato come **classe interna di un container**; quando è necessario usare un `const_iterator` insieme ad un `iterator`, è comune includere una **forward declaration** di `const_iterator` prima di definire `iterator`, per permettere il confronto tra i due tipi di iteratore.

```

template <class T> class Container {
public:
 class const_iterator; // Forward declaration

 class iterator {
public:
 typedef std::forward_iterator_tag iterator_category;
 typedef T value_type;
 typedef ptrdiff_t difference_type;

```



```

typedef T* pointer;
typedef T& reference;

iterator();
iterator(const iterator& other);
iterator& operator=(const iterator& other);
~iterator();

reference operator*() const;
pointer operator->() const;
iterator operator++(int); // Post-incremento
iterator& operator++(); // Pre-incremento
bool operator==(const iterator& other) const;
bool operator!=(const iterator& other) const;

friend class const_iterator;
bool operator==(const const_iterator& other) const;
bool operator!=(const const_iterator& other) const;

private:
 T* ptr;
 friend class Container;
 explicit iterator(T* p) : ptr(p) {}
};

iterator begin();
iterator end();
};

```

## Funzioni di base

Ogni iteratore fornisce metodi per muoversi attraverso i dati e accedervi:

- **begin()** restituisce un iteratore che punta all'inizio della sequenza
- **end()** restituisce un iteratore che punta a un elemento fittizio dopo la fine della sequenza
- **Operatori** come:
  - **\***: ritorna il dato riferito dall'iteratore, quindi un *dereferenzamento*
  - **->**: ritorna il puntatore al dato riferito dall'operatore
  - **++**: operatore di iterazione post-incremento/pre-incremento
  - **=**: uguaglianza
  - **!=**: diversità

## Conversione const

È utile implementare la conversione da iterator a const\_iterator, passando un iterator a una funzione, senza ulteriori modifiche.

```
class const_iterator {
public:
 const_iterator(const iterator& other); // Costruttore di
conversione
 const_iterator& operator=(const iterator& other); //
Assegnamento
};
```

## Begin e end

È comune rendere il costruttore di iterator privato ed accessibile solo alla classe container; questo assicura che *l'iteratore sia creato correttamente*, e usato solo per accedere ai dati del container. Di solito, questo costruttore viene invocato nei metodi begin o end del container, che **restituiscono rispettivamente un iteratore al primo elemento della sequenza e un iteratore a un elemento fittizio oltre l'ultimo**.

```
iterator begin() { return iterator(data); } // Inizio della
sequenza
iterator end() { return iterator(data + size); } // Fine della
sequenza
```

## Classe container

È importante includere la classe container all'interno della classe dell'iteratore, poiché *rendere una classe amica consente a quella classe di accedere ai membri privati e protetti dell'iteratore, incluso il suo costruttore privato*. Includendolo in questo modo, **si limita la creazione degli iteratori al solo contenitore**, che può assicurarsi che l'iteratore sia correttamente inizializzato o puntato ad una posizione valida.

Il costruttore è da dichiararsi dunque **privato**, poiché soltanto la classe friend sarà quella che è in grado di chiamare il costruttore per creare un oggetto iteratore.

```
...
private:
 friend class CLASSE_CONTAINER_PADRE;
 iterator(...) { }
```

## Traits degli operatori

Forniscono ai container e agli algoritmi generici informazioni sui tipi utilizzati dagli iteratori:

- **iterator\_category** specifica la categoria dell'iteratore, come `std::forward_iterator_tag`, `std::bidirectional_tag`, `std::random_access_iterator_tag`, etc)
- **value\_type**: tipo dei valori puntati dall'iteratore
- **pointer**: puntatore al tipo degli elementi
- **difference\_type**: tipo per rappresentare la differenza tra due iteratori, solitamente `std::ptrdiff_t`

## Miglioramenti in C++11

C++11 introduce significativi miglioramenti al linguaggio, rendendolo più moderno, efficiente e sicuro.

### Nuovi tipi di dati

- **long long**: introduce il supporto per interi a 64 bit, utili per rappresentare numeri molto grandi
  - o `long long maxValue = 0223372036854775807;`
- **nullptr**: offre un tipo dedicato per rappresentare un puntatore nullo, eliminando le ambiguità legate all'uso di 0 come valore nullo, o di NULL
  - o In contesti in cui '0' poteva essere interpretato sia come un puntatore nullo sia come un valore intero, si creavano ambiguità
  - o `int* ptr = nullptr;`
- **enum classes**: gli enumerativi ora possono avere un tipo specifico, aumentando la sicurezza e prevenendo errori legati al confronto tra tipi diversi
  - o `enum class Colori { Rosso, Verde, Blu };`
  - o `Colori colore = Colori::Rosso;`

### Deduzione del tipo

Semplifica la scrittura del codice con strumenti per la deduzione automatica dei tipi.

#### auto

Deduce il tipo di una variabile in base alla sua inizializzazione, riducendo la verbosità del codice.

```
auto numero = 42; //numero = int
auto pi = 3.14; //pi = double
```

## decltype

Permette di ottenere il tipo di una variabile esistente, utile per mantenere consistenza nei tipi.

```
int a = 5;
decltype(a) b = 10; //b è int
```

## Espressioni lambda

Le espressioni lambda sono funzioni anonime definite **inline**, ideali per operazioni rapide o temporanee, ad esempio nei cicli o negli algoritmi; la sintassi è la seguente:

```
[capture](parameters) -> returnType { body }
```

Di seguito un esempio:

```
int limite = 10;
auto minoreDelLimite = [=](int x) { return x < limite; };
std::cout << minoreDelLimite(5); //output: 1 (true)
```

## Capture clause

Indica quali variabili dello **scope esterno** possono essere usate all'interno della lambda e come accedervi. Si possono anche combinare le varie modalità.

- `[]`: nessuna variabile catturata
- `[=]`: tutte le variabili catturate per copia; i valori *non possono essere modificati all'interno della lambda*
- `[&]`: tutte le variabili catturate per riferimento; *le modifiche nella lambda influenzano le variabili originali*
- `[this]`: cattura il puntatore `this`, utile per accedere ai membri di una classe
- `[var]`: cattura specifica di `var` per copia
- `[&var]`: cattura specifica di `var` per riferimento

## Initializer list

È possibile inizializzare contenitori con una lista di valori, rendendo il codice più compatto e leggibile. Di seguito, un esempio:

```
std::vector<int> numeri = {1, 2, 3, 4};
```

## Costruttori delegati

I costruttori possono richiamarsi a vicenda, semplificando l'inizializzazione di oggetti. Di seguito, un esempio:

```
class Esempio {
public:
 Esempio() { /* codice */ }
 Esempio(int x) : Esempio() { /* codice */ } };
```

## Miglioramenti in C++14

Introduce ancora altri miglioramenti semplificando ulteriormente il codice, e migliorando l'esperienza dello sviluppatore.

### Letterali binari e separatori di cifre

C++14 consente di rappresentare numeri in **formato binario** e di utilizzare **apici** come separatori per migliorare la leggibilità dei numeri.

- **Letterali binari:** iniziano con il prefisso `0b` o `0B`, seguiti da una sequenza di 0 e 1
- **Separatore di cifre:** ' può essere usato per raggruppare cifre, utile per numeri lunghi

Di seguito, un esempio:

```
int binario = 0b1101; //13 in decimale
long grande = 1'048'576;
long hex = 0x10'0000;
```

### Lambda generiche

Ora supportano l'uso di `auto` nei parametri, rendendole **polimorfe**. Questo le rende più flessibili, permettendo di gestire tipi diversi senza dover dichiarare esplicitamente il tipo dei parametri.

```
auto somma = [](auto a, auto b){ return a + b; };
std::cout << somma(5,10); //output = 15
```

### Attributo `[[deprecated]]`

Consente di segnare funzioni, classi o altre entità come **obsolete**, mostrando un messaggio di avviso al compilatore se vengono utilizzate. Può essere utilizzato senza messaggio, o con un messaggio personalizzato.

```
[[deprecated("Usa nuovaFunzione() invece")]]
void oldFunzione();
```

## Template variabili

Consentono di dichiarare costanti o variabili parametriche con un *template*. Sono particolarmente utili per rappresentare valori costanti come pi, per vari tipi di dati.

```
template<class T>
constexpr T pi = T(3.1415926535897932385);
```

## Deduzione del tipo di ritorno

Il compilatore può dedurre il **tipo di ritorno** di una funzione quando si utilizza `auto`, eliminando la necessità di specificarlo esplicitamente.

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

## decltype(auto)

Estende la capacità di deduzione del tipo con `decltype(auto)`, che preserva riferimenti e qualificatori come `const`.

```
const int x = 0;
auto x1 = x; //int perde const
decltype(auto) x2 = x; //const int
```

## Sequenze di interi compile-time

Introduce la classe template `std::integer_sequence` per generare sequenze di interi in **fase di compilazione**, utile in combinazione con altre funzionalità come le tuple

```
template<typename T, std::size_t N>
decltype(auto) convertToTuple(const std::array<T, N>& arr){
 return arrayToTuple(arr, std::make_index_sequence<N>()); }
}
```

## Qt

Qt è una libreria open-source utilizzata per la **creazione di interfacce grafiche utente cross-platform**. Nata in Norvegia grazie alla Trolltech, è stata in seguito acquisita da Nokia nel 2008, per poi passare sotto il controllo di Digia nel 2012.

Offre strumenti potenti e flessibili per sviluppare applicazioni su diverse piattaforme. Supporta compilatori come *GCC* e *MS Visual Studio*, e dispone di un ambiente di sviluppo

integrato, **QtCreator**, che include un designer per la creazione rapida di interfacce grafiche, e un browser per accedere alla documentazione completa. Di seguito, un esempio.

```
#include <QApplication>
#include <QLabel>
int main(int argc, char *argv[]){
 QApplication app(argc, argv);
 QLabel label("Hello Qt!");
 label.show();
 return app.exec();
}
```

## Architettura

Si basa su un'architettura modulare e flessibile che comprende:

- **Condivisione implicita:** un meccanismo che minimizza l'utilizzo delle risorse sfruttando una copia per riferimento *copy-on-write* fino a quando i dati non vengono modificati. Una volta che vengono modificati, allora una copia indipendente viene creata automaticamente. È interamente automatizzato anche in contesti multi-thread.
- **Modello ad oggetti:** estende le funzionalità della programmazione orientata agli oggetti di C++, introducendo proprietà che possono essere aggiunte dinamicamente e recuperate a runtime.
- **Sistema di eventi:** gli eventi, derivati dalla classe `QEvent`, consentono una gestione avanzata delle interazioni, come nel caso del click di un pulsante o del movimento del mouse.

## Segnali e slot

Sono due tratti distintivi di Qt, ed è un meccanismo utilizzato per la comunicazione tra oggetti.

Un **segnale** è emesso da un oggetto in risposta a un evento, mentre uno **slot** è la funzione che risponde a un determinato segnale. Questa soluzione garantisce sicurezza *tipologica* e *separazione* delle responsabilità tra gli oggetti.

Si usa la funzione `QObject::connect()` per connettere un segnale e uno slot di due oggetti, con la seguente sintassi:

```
QObject::connect(sender, SIGNAL(signal), receiver, SLOT(action));
```

Questo approccio è **type-safe**, cioè garantisce che il tipo dei parametri del segnale corrisponda a quelli dello slot. Inoltre, i segnali e gli slot non richiedono che gli oggetti conoscano i dettagli l'uno dell'altro, favorendo un design modulare.

## Widget e gestione delle interfacce

Sono elementi grafici fondamentali di un'interfaccia utente in Qt; esempi includono *pulsanti*, *menu*, *barre di scorrimento*, *finestre*. Ogni widget può agire sia come elemento grafico, indipendente, sia come contenitore per altri widget.

Per organizzarli, si utilizzano i layout manager: **QHBoxLayout**, **QVBoxLayout**, **QGridLayout**.

## Input e output

Gestisce l'accesso ai file e ad altre risorse tramite classi dedicate: **QFile** per operazioni di lettura e scrittura su file; **QTextStream** per gestire contenuti testuali e conversioni di codifica; **QDataStream** per la lettura e scrittura di file binari con supporto per vari tipi di dati. Di seguito, un esempio.

```
QFile file("example.txt");
if (file.open(QIODevice::WriteOnly)) {
 QTextStream out(&file);
 out << "Qt" << endl;
 file.close(); }
```

## QGuiApplication

La classe QGuiApplication gestisce il **flusso di controllo** (ciclo degli eventi, inizializzazione, finalizzazione dell'applicazione, gestione delle sessioni) e **impostazioni principali della GUI**.

Nelle applicazioni GUI che utilizzano Qt esiste un oggetto QGuiApplication, mentre in quelle non GUI si usa QCoreApplication. In quelle basate su QWidget, si usa QApplication.

Le principali aree di responsabilità sono:

- **Inizializzare l'applicazione** con le impostazioni desktop dell'utente, come palette(), font() e styleHints()
- Eseguire la **gestione degli eventi**, ovvero ricevere gli eventi dal sistema di finestre sottostante e inviarli ai widget pertinenti
- Fornire la **localizzazione delle stringhe** tramite translate()
- Poiché conosce le finestre dell'applicazione può chiedere quale finestra si **trova in una posizione**, usando topLevelAt(), ottenere un elenco di topLevelWindows(), etc
- Fornisce supporto per la **gestione delle sessioni**, quindi consente alle applicaizoni di terminare quando l'utente si disconnette, annullare un processo di spegnimento, conservare lo stato dell'applicazione per una sessione futura con isSessionRestored(), sessionId(), commitDataRequest(), saveStateRequest()



## Data Visualization

Basato su Qt e OpenGL, sfrutta l'accelerazione hardware per offrire funzionalità avanzata. Consente di **creare interfacce 3D interattive e dinamiche** per rappresentare *mappe di profondità, dati medicali, e grandi quantità di dati provenienti da sensori*. Supporta visualizzazioni come grafici a barre, a dispersione, o di superfici 3D.

## Series

Una serie è una combinazione di **un insieme di elementi di dati logicamente connessi**, gestiti da un proxy di dati, e **proprietà visive che descrivono come devono essere visualizzati gli elementi di dati**.

Ogni tipo di visualizzazione ha il proprio tipo di series, e tutti i grafici possono avere più serie aggiunte contemporaneamente.

```
Q3DBars graph;
QBar3DSeries *series = new QBar3DSeries;
QLinearGradient barGradient(0, 0, 1, 100);
barGradient.setColorAt(1.0, Qt::white);
barGradient.setColorAt(0.0, Qt::black);
series->setBaseGradient(barGradient);
series->setColorStyle(Q3DTheme::ColorStyleObjectGradient);
series->setMesh(QAbstract3DSeries::MeshCylinder);
graph->addSeries(series);
```

## Data proxies

I dati che gli utenti desiderano visualizzare possono essere disponibili in molti formati che spesso non possono essere usati direttamente. Implementa **proxy di dati** che possono essere alimentati dall'utente con i propri dati in un formato noto.

Ogni tipo di visualizzazione ha un **proprio tipo di proxy di base**, che prende i dati in un formato adatto a tale visualizzazione. Il proxy di base per QBar3DSeries è **QBarDataProxy**, memorizza le righe di oggetti **QBarDataItem**, il quale memorizza un valore a barra singola. Ulteriori *typedef* sono forniti per container QBarDataArray e QBarDataRow.

```
Q3DBars graph;
QBarDataProxy *newProxy = new QBarDataProxy;
QBarDataArray *dataArray = new QBarDataArray;
dataArray->reserve(10);
for (int i = 0; i < 10; i++) {
 QBarDataRow *dataRow = new QBarDataRow(5);
 for (int j = 0; j < 5; j++)
 (*dataRow)[j].setValue(myData->getValue(i,j));
 dataArray->append(dataRow);
}
newProxy->resetArray(dataArray);
```

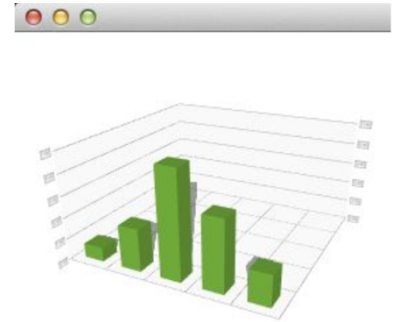
```
graph->addSeries(new QBar3DSeries(newProxy));
```

Gli oggetti series possono **possedere solo un singolo proxy alla volta**; il proxy esistente viene cancellato quando un altro è impostato sulla serie.

## 3D Bar Graphs

Presentano dati utilizzando barre **raggruppate per categoria**. Le classi principali da usare sono Q3DBars per creare il grafico, e QBar3DSeries e QBarDataProxy per gestire data e proprietà.

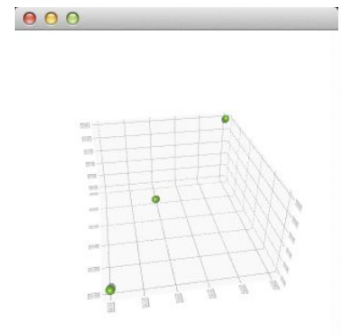
```
Q3DBars bars;
bars.setFlags(bars.flags() ^
Qt::FramelessWindowHint);
bars.rowAxis()->setRange(0,4);
bars.columnAxis()->setRange(0,4);
QBar3DSeries *series = new QBar3DSeries;
QBarDataRow *data = new QBarDataRow;
*data << 1.0f << 3.0f << 7.5f << 5.0f << 2.2f;
series->dataProxy()->addRow(data);
bars.addSeries(series); bars.show();
```



## 3D Scatter Graphs

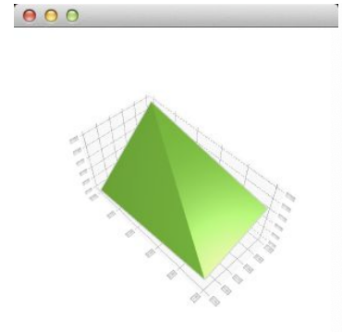
Presentano i dati servendosi di punti 3D. La classe Q3DScatter deve essere utilizzata per creare il grafico, mentre le classi QScatter3DSeries e QScatterDataProxy sono usate per gestire *l'inserimento dei dati nel grafico, il controllo delle proprietà del grafo*.

```
Q3DScatter scatter;
scatter.setFlags(scatter.flags() ^
Qt::FramelessWindowHint);
QScatter3DSeries *series = new QScatter3DSeries;
QScatterDataArray data;
Data << QVector3D(0.5f, 0.5f, 0.5f)
 << QVector3D(-0.3f, -0.5f, -0.4f)
 << QVector3D(0.0f, -0.3f, 0.2f);
series->dataProxy()->addItem(data);
scatter.addSeries(series); scatter.show();
```



## 3D Surface Graphs

La classe Q3DSurface deve essere utilizzata per creare il grafico, e le classi QSurface3DSeries e QSurfaceDataProxy sono invece utilizzate per gestire *l'inserimento dei dati nel grafico, il controllo delle proprietà del grafo.*



```
Q3DSurface surface;
surface.setFlags(surface.flags() ^
Qt::FramelessWindowHint);
QSurfaceDataArray *data = new QSurfaceDataArray;
QSurfaceDataRow *dataRow1 = new QSurfaceDataRow;
QSurfaceDataRow *dataRow2 = new QSurfaceDataRow;
*dataRow1 << QVector3D(0.0f, 0.1f, 0.5f)
 << QVector3D(1.0f, 0.5f, 0.5f);
*dataRow2 << QVector3D(0.0f, 1.8f, 1.0f)
 << QVector3D(1.0f, 1.2f, 1.0f);
*data << dataRow1 << dataRow2;
QSurface3DSeries *series = new QSurface3DSeries;
series->dataProxy()->resetArray(data);
surface.addSeries(series); surface.show();
```

## Qt charts

Fornisce un set di component grafici facili da usare. Pertanto, usando il **Qt Graphics View Framework**, pertanto i grafici possono essere facilmente integrati con le moderne interfacce utente. Dobbiamo includere questo:

```
#include <QtCharts>
using namespace QtCharts;
```

Gestisce la rappresentazione grafica di diversi tipi di serie ed altri oggetti annessi ai grafici, come la *legenda* e gli *assi*.

È un **QGraphicsWidget** che può essere visualizzato in una **QGraphicsScene**. Altrimenti, si visualizza un grafico usando la classe QChartView invece di QChart.

Le serie viste nel modulo QtDataVisualization, sono presenti anche in Qt charts. I dati relativi a ciascun tipo di grafico sono rappresentati usando classi derivate dalla classe QAbstractSeries.

```
QLineSeries *series = new QLineSeries();
series->add(0,6);
series->add(2,4);
chartView->chart()->addSeries
```

Qt Charts consente di personalizzare l'intervallo di **valori sugli assi**, cambiandone il tipo, ed è possibile impostare un asse e mostrare una linea con segni di graduazione, linee della griglia e sfumature.

Tutte le tipologie di assi sono specializzazioni di `QAbstractAxis`: `QValueAxis`, `QCategoryAxis`, `GDateTimeAxis`, etc.

Una **legenda** è un oggetto grafico che mostra la legenda di un grafico. Gli oggetti di legenda non possono essere creati o cancellati, ma possono essere referenziati tramite la classe `QChart`. Lo stato della legenda viene aggiornato da `QChart` o `ChartView`, quando le serie cambiano.

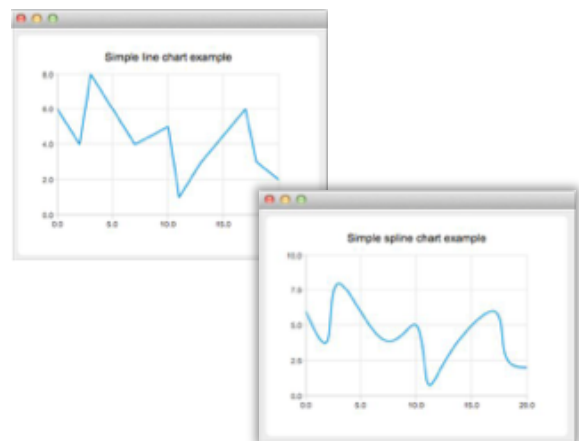
Una legenda può essere posizionata *sopra/sotto a destra/sinistra*. È possibile anche nascondere i singoli indicatori della legenda, o dall'intera legenda.

I marcatori della legenda possono essere modificati utilizzando la classe padre `QLegendMarker`, e le sottoclassi per ciascun tipo di serie (`QAreaLegendMarker`, `QBarLegendMarker`, etc).

### Grafici a line e spline

I grafici a linee e spline presentano i dati come una serie di punti dati collegati da linee. In un grafico **a linee**, i punti dati sono collegati da linee rette. In un grafico **spline**, sono collegati da una spline, la quale viene disegnata usando `QPainterPath`.

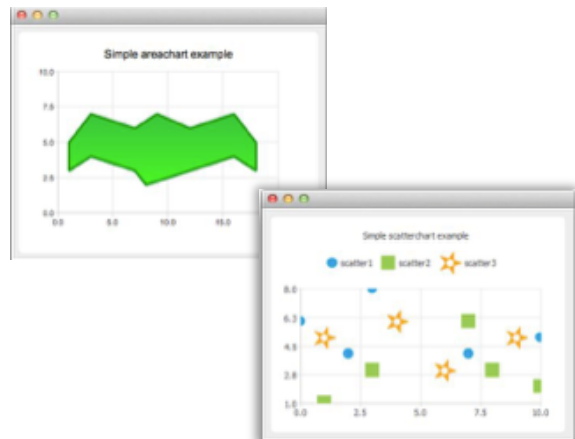
| Grafico  | Implementazione/usa                                                                                                                                                                         |
|----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| A linee  | <code>QLineSeries</code>                                                                                                                                                                    |
| A spline | <code>QSplineSeries</code> , eredita da <code>QLineSeries</code><br>Tipo <code>SplineSeries</code> che eredita da <code>LineSeries</code><br>Spline disegnata con <code>QPainterPath</code> |



### Grafici area e dispersione

I grafici ad area presenta i dati come un'area limitata da due linee, mentre il grafico a dispersione è costituito da una collezione di punti.

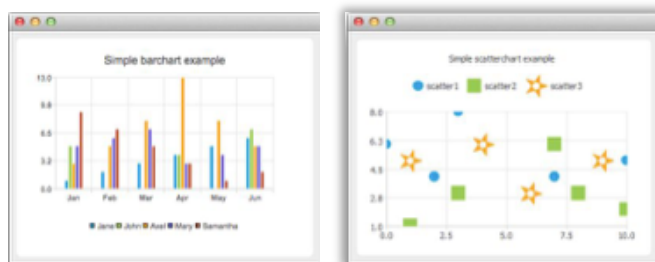
| Grafico          | Implementazione/usa                                                                                                                |
|------------------|------------------------------------------------------------------------------------------------------------------------------------|
| Area             | QAreaSeries<br>Asse x: unico limite,<br>QLineSeries/LineSeries<br>Si può usare<br>QLineSeries/LineSeries<br>come entrambi i limiti |
| A<br>dispersione | QScatterSeries                                                                                                                     |



### Grafici a barre

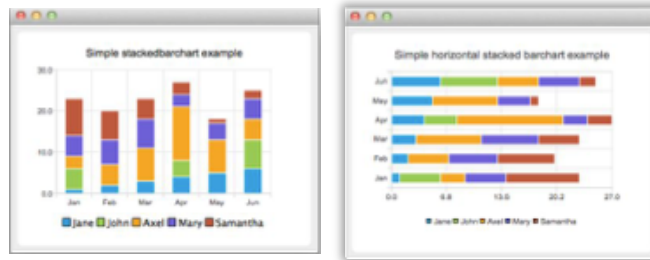
Un grafico a barre presenta i dati come barre orizzontali o verticali raggruppate per categoria.

- **QBarSet**: set di barre in un grafico a barre
- **QAbstractBarSeries**: classe padre astratta per tutte le classi di serie di barre e il tipo AbstractBarSeries è il padre di tutti i tipi di serie di barre.
  - o Il tipo di serie determina come vengono presentati i dati
- **QBarSeries**: dati come barre verticali raggruppate per categoria
- **QHorizontalBarSeries**: dati come barre orizzontali



- **QStackedBarSeries**: serie di dati come barre impilate verticalmente con una barra per categoria

- Per la versione orizzontale la classe è **QHorizontalStackedBarSeries**



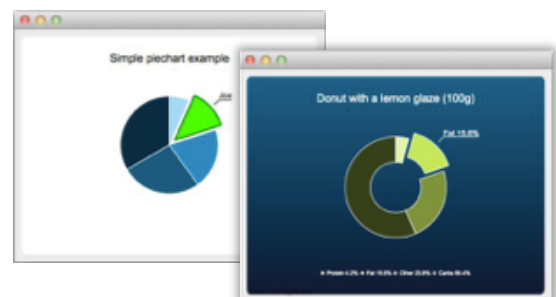
- **QPercentBarSeries**: presenta verticalmente una serie di dati in percentuale
  - Per la versione orizzontale la classe è **QHorizontalPercentBarSeries**



### Grafici a torte

Presentano dati come torte i cui dati ne costituiscono le fette. Può essere convertita in ciambella specificando la dimensione del buco centrale compresa tra 0.0 e 1.0

| Grafico | Implementazione/usa |
|---------|---------------------|
| Torta   | QPieSeries          |
| Fette   | QPieSlice           |



## Esame

Struct vs classe: contenitori di dati

Chiamata a funzione

Cos'è un orfano

Locazione di memoria (allocazione di puntatori)

Compila o non compila

Default membri della classe

9,5/15 – migliora gli esercizi