1/09/25

C++

È un linguaggio di programmazione general-purpose. Consente una programmazione a oggetti.
La OOP è un paradigma che organizza il codice attorno a degli oggetti, ovvero entità che combinano attributi (dati) e metodi (comportamenti) in un'entità.
I principi fondamentali sono:
**- astrazione**
**- incapsulamento**
**- ereditarietà**
**- polimorfismo**
Un oggetto è un'unità logica che combina: **Attributi (stato):** variabili o proprietà che descrivono le caratteristiche dell'oggetto & **Metodi (comportamenti):** funzioni o procedure che definiscono cosa l'oggetto sa fare o come può interagire con altri oggetti.

La distinzione tra **classe** e **oggetto** è cruciale. La classe è un **modello astratto** che definisce struttura e operazioni, come il progetto architettonico di una casa. L'oggetto è l'**istanza concreta** generata da quel modello, cioè la casa realmente costruita, con il suo colore e i suoi materiali.

**CLASSE**

è un *modello astratto* che descrive un insieme di oggetti con caratteristiche comuni. Contiene **attributi** (dati) e **metodi** (funzioni che operano su quei dati). La **classe** è come il progetto di una casa: descrive quanti piani ha, il numero delle stanze, il tipo di tetto. Non è una casa reale, ma un *modello*.

**OGGETTO**

è un'istanza concreta di una classe, cioè la realizzazione di quel modello. Mentre la classe definisce la *ricetta*, l'oggetto rappresenta il *piatto cucinato*. Un **oggetto** è la casa concreta costruita seguendo quel progetto: ha muri, porte, un colore preciso.

**ASTRAZIONE**

Consiste nel rappresentare i concetti complessi attraverso modelli semplificati: per esempio la classe *auto* può avere attributi come *colore* e *velocità*, e metodi come *accelera()* o *frena()*. Non ci interessa il dettagli di ogni componente interna al motore, ma solo ciò che serve al nostro scopo

**INCAPSULAMENTO**

È il principio per cui i dati interni di un oggetto vengono nascosti e resi accessibili solo tramite metodi controllati (getters e setters). Serve a ridurre la complessità e proteggere l'integrità dello stato interno.

**EREDITARIETA'**

Permette di definire nuove classi (dette derivate o figlie) a partire da classi esistenti (genitori o superclassi). La classe figlia eredita attributi e metodi della classe madre, evitando duplicazioni. Per esempio da *veicolo* possono derivare *auto, moto, bicicletta*, ciascuno con proprietà comuni (*ruote, velocità)* ma anche caratteristiche specifiche.

**POLIMORFISMO**

Indica la capacità degli oggetti di comportarsi in modi diversi a seconda del contesto, per condividendo la stessa interfaccia. Per esempio il metodo *muovi()* esiste in *auto* che in *uccello*, ma nel primo caso significa "rotolare sulle ruote", nel secondo *volare*. Il programma può invocare *muovi()* senza preoccuparsi del tipo concreto.

## COSTRUTTORE

È un metodo speciale di una classe in C++ che viene eseguito **automaticamente** quando viene creata una nuova istanza della classe. Ha lo stesso nome della classe, non ha tipo di ritorno (nemmeno void), serve a inizializzare gli attributi dell'oggetto.

Tipi:

- DEFAULT

È un costruttore che non prende parametri. Viene chiamato automaticamente quando crei un oggetto senza specificare valori iniziali. Se non lo scrivi tu, il compilatore **ne genera uno implicito** (ma attenzione: non inizializza i membri a zero, li lascia con valori indefiniti).

```
class Punto {
        int x;
        int y;
public:
        Punto() : x(0), y (0) {} //default constructor
};
```

- PARAMETERIZED CONSTRUCTOR

È un costruttore che accetta argomenti, permettendo di inizializzare i membri dell'oggetto con valori specifici al momento della creazione.

```
class Punto {
  int x, y;
public:
  Punto(int a, int b) : x(a), y(b) {} // costruttore parametrico
};
```

- COPY CONSTRUCTOR

È un costruttore che prende come parametro una **referenza costante a un oggetto della stessa classe** (const ClassName&).

Serve a creare un nuovo oggetto come *copia* di uno già esistente. Il compilatore ne genera uno di default, ma se la classe gestisce risorse dinamiche (heap, file, socket), conviene scriverlo per evitare copie superficiali (*shallow copy*).

```
class Punto {
  int x, y;
public:
  Punto(const Punto &other) : x(other.x), y(other.y) {}
};
```

### *Initializer list*

*La **member initializer list** è un costrutto di C++ che permette di **inizializzare direttamente** i membri della classe **prima** che entri nel corpo del costruttore.*

*Classe::Classe(parametri) : membro1(valore1), membro2(valore2), ... { }*

***IMPORTANTE: Efficienza: evita doppie operazioni (costruzione vuota → assegnazione). Necessità tecnica:** alcuni membri **devono per forza** essere inizializzati in una initializer list, perché non possono essere assegnati dopo: Membri const (devono avere un valore al momento della costruzione), Riferimenti (&) che devono legarsi subito a una variabile.*

## DISTRUTTORE

È un metodo speciale che viene eseguito **automaticamente** quando un oggetto viene distrutto (ad esempio, quando esce dallo scope o viene chiamata delete). Ha lo stesso nome della classe preceduto da tilde ~. Non accetta parametri e non ha tipo di ritorno. Serve a rilasciare risorse.

**PUBLIC, PRIVATE, PROTECTED**

In C++ le **classi** hanno membri (attributi e metodi). La **visibilità** di questi membri è regolata da **access specifiers** (specificatori di accesso):

- **public** → accessibile da chiunque (anche fuori dalla classe). Tutti possono accedere ai membri pubblici, anche da codice esterno alla classe. Tipicamente usato per i **metodi di interfaccia** e meno per i dati. Espone le funzionalità al mondo esterno (API)
- **private** → accessibile solo dall'interno della classe stessa. Solo la classe stessa può accedere ai membri privati. Nemmeno le classi derivate possono accedervi. Usato per **nascondere lo stato interno** ed evitare che venga modificato liberamente (incapsulamento).
- **protected** → accessibile dalla classe stessa **e** dalle classi derivate. Simile al private, ma con una differenza: I membri **protected** non sono visibili all'esterno, ma lo sono per le **classi derivate**. Utile quando vuoi dare alle sottoclassi la possibilità di usare o estendere certi dettagli, ma non al mondo esterno.

Se non specifichi nulla, per **class** i membri sono **private** di default, mentre per **struct** sono **public** di default.

Gli access specifiers realizzano il principio di **incapsulamento**, uno dei pilastri della OOP: Si definisce **cosa** è visibile (interfaccia pubblica). Si nasconde **come** funziona internamente (implementazione privata/protetta).

**GETTER & SETTER**

**Getter**: è un metodo pubblico che serve a **ottenere** (leggere) il valore di un attributo privato di una classe - (o *Accessor method*) → Restituisce il valore di un attributo privato, È quasi sempre marcato come const perché leggere non deve modificare l'oggetto.

**Setter**: è un metodo pubblico che serve a **modificare** (scrivere) il valore di un attributo privato. R - (o *Mutator method*) → Permette di cambiare un attributo privato , Puoi aggiungere logica per validare i valori.

In **C++ i getter e i setter non esistono "automaticamente"** (a differenza di altri linguaggi che hanno le *properties* come C#). Sono **metodi che deve scrivere l'utente** all'interno della classe, di solito con i prefissi **get** e **set** seguiti dal nome dell'attributo, ma il nome non è imposto dal linguaggio: è solo una **convenzione di stile** (per rendere chiaro il significato a chi legge il codice).

## Il puntatore `this` e l'operatore `*this`

Ogni metodo non statico in una classe riceve implicitamente un parametro chiamato *this*, è un puntatore all'oggetto corrente, cioè all'istanza su cui si sta lavorando. Se la classe è *Contact*, dentro i suoi metodi *this* ha tipo *Contact**. Serve per distinguere i membri della classe dalle variabili locali o dai parametri.

```
class Contact {
private:
  std::string _name;
public:
  Contact(std::string name) {
    this->_name = name;   // uso di this per distinguere dal parametro
  }
};
```

***DIFFERENZA TRA THIS→ E THIS****

this→ è un puntatore: this→Contact*. Si usa per accedere al membro della classe.

*this è l'oggetto stesso (dereferenziazione del puntatore). Restituisce una referenza all'oggetto corrente. SI usa se bisogna restituire l'oggetto stesso, ad esempio in un operatore di assegnazione o nei metodi fluenti.

## PUNTATORI IN C++

un puntatore è una variabile che non contiene un valore normale, ma l'indirizzo di memoria di un'altra variabile.

La logica del puntatore si può paragonare a un biglietto con l'indirizzo di casa: non contiene la casa, ma ti dice dove trovarla. Se ho una variabile `int x = 42;`, scrivere `int *p = &x;` significa che `p` conserva l'indirizzo di `x`. Per leggere o modificare il valore puntato devo "dereferenziare" il puntatore, cioè usare l'operatore `*p`. Così, `*p = 100;` cambia direttamente il contenuto di `x`.

I puntatori sono indispensabili quando gli oggetti devono vivere al di là dello scope in cui sono creati. In C, e di conseguenza in C++, l'uso di `new` riserva memoria nello heap e restituisce un puntatore, mentre `delete` libera quella memoria. Senza questa gestione manuale avremmo meno flessibilità, ma anche meno responsabilità.

È interessante osservare la differenza tra `this` e `*this`. All'interno di un metodo di una classe, `this` è un puntatore all'oggetto, quindi ha tipo `Classe*`. Se lo dereferenzi con `*this`, ottieni una referenza all'oggetto stesso, cioè un alias del contenitore reale. Questo è particolarmente utile negli **operatori di assegnazione**, dove si ritorna spesso `*this` per permettere la concatenazione di assegnazioni.

### L'operatore &

il suo significato dipende dal contesto. Quando lo usi davanti a un identificatore già esistente, per esempio scrivendo `&x`, stai chiedendo l'`indirizzo di memoria` di quella variabile. In questo caso si parla dell'**`operatore "address-of"`**. Se ho una variabile int n = 42;, l'espressione &n non vale più "42", ma un numero esadecimale che rappresenta la posizione in memoria in cui quel 42 è memorizzato.

Quando invece & appare nella dichiarazione di una variabile, significa qualcosa di diverso: la variabile che stai dichiarando sarà un **`riferimento`**, cioè un alias per un oggetto già esistente. Scrivere **int &r = n;** *vuol dire che r non è una nuova cella di memoria, ma un altro nome per la stessa cella di n*. Se cambi r, cambia anche n, e viceversa. Un **riferimento** è un **alias** di una variabile esistente, dichiarato con & in **dichiarazione**: `int& r = x;`. A differenza di un puntatore, un riferimento **non può essere nullo**, **non può essere riassegnato** dopo l'inizializzazione e si usa con la stessa sintassi della variabile originale.

# LA CLASSE STD::STRING

È una classe della Standard Template Library. Si trova nel namespace std e definita dall'header <string>.

È una specializzazione della classe template std::basic_string che può rappresentare stringhe di caratteri di tipo diverso come char, wchar_t char16_te …

A differenza delle **`C-string`** (array di caratteri terminati da `\0`), `std::string` gestisce automaticamente la memoria, conosce la propria lunghezza e fornisce metodi pronti per manipolare il testo.

Immagina di dover gestire una frase come `"Ciao Mondo"`. Con gli array C-style dovresti dichiarare un `char arr[20] = "Ciao Mondo";` e stare attenta a non uscire dai limiti e a non dimenticare il carattere `\0` finale.

Con std::string basta:

```
#include <iostream>

#include <string>

int main() {

   std::string s = "Ciao Mondo";

   std::cout << s << std::endl;

}
```

## METODI classe std::string

Costruttori più usati:

std::string() → crea una stringa vuota

std::string(const char *s) → inizializza da una C-String

std::string(size_t n, char c) → crea una stringa con n copie del carattere c

std::string(const std::string &other) → copy constructor

## Metodi di accesso e informazioni

Il metodo size() (alias length()) restituisce la quantità di caratteri attualmente memorizzati. È molto più veloce di strlen delle C-string, perché la lunghezza viene mantenuta internamente e non va ricalcolata a ogni chiamata. Il metodo empty() controlla se la stringa è vuota, evitando confronti manuali con "". Per accedere a un singolo carattere puoi usare l'operatore di indice operator[], che restituisce una referenza al carattere, oppure il metodo at(), che in più verifica che l'indice non sia fuori dai limiti, lanciando un'eccezione in caso contrario. Il metodo **substr** (abbreviazione di *sub-string*) serve a **estrarre una parte della stringa** originale, restituendo un nuovo oggetto std::string.

L'**operatore []** della classe std::string serve per *accedere direttamente a un singolo carattere all'interno della stringa*. La sua firma è simile a questa:

```
char &operator[](size_t pos);
const char &operator[](size_t pos) const;
```

Quindi, se ho una std::string s = "Ciao";, posso scrivere s[0] per ottenere il carattere 'C'.
La differenza tra le due versioni è che la prima restituisce un riferimento modificabile (puoi cambiare il carattere), mentre la seconda si applica alle stringhe costanti e ti permette solo di leggere.