

# *Classi e oggetti in C++*

Prof. CAPOBIANCO Ing. Nicolò



# Classi ed oggetti in C++ 1

Il C++ si distingue da C in quanto è un linguaggio orientato agli oggetti (object-oriented). Gli oggetti in C++ sono analoghi agli oggetti del mondo reale, come macchine, motociclette, biciclette: hanno uno *stato* ed un *comportamento*.

Un oggetto in C++ contiene attributi e metodi.

I primi sono variabili interne che permettono all'oggetto di memorizzare le informazioni (lo stato dell'oggetto).

I metodi sono funzioni interne all'oggetto che descrivono cosa esso possa fare.

Gli attributi costituiscono il nucleo dell'oggetto e vengono circondati e nascosti dai metodi (funzioni). Questo metodo viene chiamato **incapsulamento** delle informazioni.

Lo scopo è di non mostrare all'utilizzatore i dettagli dell'oggetto che non sono importanti dal punto di vista dell'utilizzo.

Nella vita reale è la stessa cosa.

---

---

## *Classi ed oggetti in C++ 2*

Infatti, per guidare una moto non è necessario conoscere come è costruita.

Anche se un meccanico cambia parte del motore un motociclista continua a guidarla regolarmente.

Analogamente, gli oggetti C++ permettono al programmatore di cambiare la parte nascosta del progetto senza dover modificare altre parti del programma.

Ovviamente, la parte visibile dell'oggetto (il suo comportamento) non deve cambiare.

---

---

# Classi ed oggetti in C++ 3

Una classe si definisce con la parola chiave **class**. Per esempio, la definizione di una classe di nome Moto può essere la seguente:

```
class Moto {  
    int accesa;  
    double velocita;  
};
```

Moto m;

Questa classe ha due attributi: accesa e velocita.

Abbiamo anche dichiarato una variabile di nome m e di tipo Moto.

**La variabile m è un oggetto, mentre Moto è una classe.**

A loro volta anche gli attributi possono essere degli oggetti.

Pensiamo ad una classe PUNTO con due coordinate ed una classe SEGMENTO che usa come attributi la classe PUNTO.

# Classi ed oggetti in C++ 4

Gli attributi di un oggetto, al contrario di una struct, non possono essere normalmente accessibili dall'esterno, ma possono essere utilizzati solo dai metodi dell'oggetto. Per esempio, l'istruzione `m.velocita=200` non è corretta perchè l'attributo velocità non è accessibile dall'esterno; si dice che è *privato*.

Per rendere pubblico un attributo, si deve usare la parola chiave `public`.

```
class Moto {  
    int accesa;  
public:  
    double velocita;  
};
```

Così, posso usare l'istruzione `m.velocita=200`

Se voglio rendere privati degli attributi, mettiamo la parola chiave *private*.

---

---

# Classi ed oggetti in C++ 5

Una classe, come detto, può contenere dei metodi, che normalmente sono privati. Per renderli pubblici è sufficiente mettere public:

Esempio della classe Moto.

```
class Moto {
    int accesa;
    double velocita;
public:
    void accendi()
    {
        accesa = 1;
    }
    void accelera(double x)
    {
        velocita = velocita + x;
    }
    double a_quanto_vado()
    {
        return velocita;
    }
};
```

---

---

# Classi ed oggetti in C++ 6

Per eseguire un metodo, si usa una sintassi simile a quella di una struttura:

```
m.accelera(10); //aumenta di 10 il valore dell'attributo velocita  
cout << m.a_quanto_vado(); // mostra su video il contenuto  
dell'attributo velocita.
```

Gli attributi di una classe sono visibili all'interno di tutti i metodi della classe, mentre le variabili locali di un metodo sono visibili solo all'interno del metodo.

Può succedere che un metodo di una classe possa richiamare un metodo di un'altra classe.

Esempio:

---

---

# Classi ed oggetti in C++ 7

```
class Punto {
    double x;
    double y;
public:
    void trasla(double dx, double dy) //metodo che permette di traslare le coordinate.
    {
        x = x + dx;
        y = y + dy;
    }
};
```

```
class Segmento {
    Punto a;
    Punto b;
public:
    void trasla(double dx, double dy) //permette di traslare i vertici del segmento
    {
        a.trasla(dx); //la classe si serve del metodo trasla della classe punto.
        b.trasla(dy);
    }
};
Segmento S;
s.trasla(7,88);
```

## *Classi ed oggetti in C++ 8*

Fino ad ora, il codice dei metodi era interno alla definizione della classe. Ottimo, ma poco pratico. Infatti, se il metodo è complesso avrà molte linee di codice e ciò renderà praticamente illeggibile la definizione della classe.

Il C++ permette di definire l'implementazione di un metodo al di fuori della classe, mantenendo all'interno della classe solo i prototipi dei metodi.

Vediamo come si fa con l'esempio delle classi Punto e Segmento.

---

---

# Classi ed oggetti in C++ 9

```
class Punto {
    double x;
    double y;
public:
    void trasla(double dx, double dy) ; //prototipo del metodo
}; //termina la dichiarazione della classe
void Punto::trasla(double dx, double dy)
    {
        x = x + dx;
        y = y + dy;
    }
class Segmento {
    Punto a;
    Punto b;
public:
    void trasla(double dx, double dy) ;
};
void Segmento::trasla(double dx, double dy)
    {
        a.trasla(dx); //la classe si serve del metodo trasla della classe punto.
        b.trasla(dy);
    }
};
```

# Classi ed oggetti in C++ 10

All'interno di una classe possono esserci più metodi con lo stesso nome, ma devono avere diversi il numero e/o il tipo dei parametri (non è sufficiente avere diverso il tipo che ritorna). Questa possibilità viene chiamata *overloading* dei metodi.

E' possibile per una classe specificare uno o più *costruttori*. E' un metodo che ha lo stesso nome della classe e che ha lo scopo di inializzare gli attributi della classe.

Viene chiamato, al contrario degli altri metodi, automaticamente quando un oggetto viene definito o creato.

Per essi non deve essere indicato il tipo del risultato.

Come gli altri metodi, i costruttori possono essere più di uno grazie al meccanismo di overloading. In questo caso si distinguono per il numero e/o tipo di parametri diversi.

Il costruttore senza parametri viene detto costruttore di default.

---

---

# Classi ed oggetti in C++ 11

```
class Moto {
    int accesa;
    double velocita;
public:
    Moto(); //questo è il costruttore di default
    Moto(double v); //un altro costruttore
}

Moto::Moto() // questo è il costruttore di default
{
    accesa = 0;
    velocita = 0;
}

Moto::Moto(double v) // l'altro costruttore
{
    accesa = 0;
    velocita = v;
}

Moto m; // chiamiamo il costruttore di default

Moto m1(70.0); // chiamata al secondo costruttore
```

# Classi ed oggetti in C++ 12

Oltre ai costruttori è possibile specificare anche un *distruttore*.

È un metodo particolare che esegue alcune operazioni prima che un oggetto venga “distrutto”.

Un oggetto di una classe può essere distrutto o cancellato in due diverse situazioni:

- se l'oggetto è stato allocato dinamicamente con l'istruzione **new**, poi può essere de-allocato e cancellato con **delete**. (vediamo dopo)
- se l'oggetto è locale ad una funzione, esso viene cancellato quando la funzione ha termine.

Il distruttore è un metodo particolare con lo stesso nome della classe, senza parametri ed è preceduto dal simbolo ~.

Il distruttore viene sempre chiamato automaticamente quando un oggetto viene cancellato dalla memoria di un computer.

Anche per il distruttore non deve essere indicato il tipo del risultato.

Il distruttore è unico, non ammette quindi overloading.

---

---

# Classi ed oggetti in C++ 13

```
class Moto {
    int accesa;
    double velocita;
public:
    Moto(); //questo è il costruttore di default
    ~Moto(); // è il distruttore
    void accelera(double x);
}
Moto::Moto() // questo è il costruttore di default
{
    accesa = 0;
    velocita = 0;
}
Moto::~Moto(); //distruttore
{
    cout<< velocita;
}
void Moto::accelera(double x)
{
    velocita = velocita + x;
}
void prova()
{
    Moto m; //chiama il costruttore
    m.accelera(15); //chiama il distruttore.
}
void main()
{
    prova();
}
```

# Classi ed oggetti in C++ 14

I costruttori e i distruttori sono particolarmente importanti quando uno o più attributi sono puntatori a zone di memoria allocati dinamicamente.

Anche gli oggetti, come tutte le variabili in C++, possono essere creati dinamicamente:

```
Moto *m = new Moto();
```

Questa allocazione può essere anche fatta in un tempo successivo.

Un oggetto creato dinamicamente, quando non serve più, viene deallocato con l'istruzione:

```
delete m;
```

Questa istruzione, prima di liberare la memoria occupata dall'oggetto, chiama il suo distruttore.

E' possibile allocato un vettore di oggetti. Ma quando bisogna deallocare, è necessario specificare che si tratta di un vettore di oggetti:

```
delete [] m;
```

altrimenti non vengono chiamati i distruttori degli oggetti del vettore.

---

---

# **Polimorfismo ed Ereditarietà in C++ 1**

L'ereditarietà è una caratteristica propria dei linguaggi ad oggetti che permette di creare classi figlie di una classe genitore.

Una classe figlia può utilizzare (eredita) sia gli attributi sia i metodi della classe genitore, ed in più può anche avere degli attributi e/o metodi propri che la classe genitore non può utilizzare.

Questo meccanismo si può ripetere, pertanto una classe figlia può, a sua volta, essere una classe genitore di altre classi figlie costruendo così un albero di ereditarietà.

Come è possibile avere più classi figlie da una classe genitore, è anche possibile che una classe figlia provenga da più classi genitori (ereditarietà multipla).

Per semplicità si farà sempre riferimento ad una ereditarietà singola, ossia un solo genitore per figlio).

La regola pratica da seguire per la costruzione di un albero di ereditarietà dice che se un oggetto A è anche un oggetto B, allora la classe A eredita dalla classe B. Vediamo un esempio:

---

---

# Polimorfismo ed Ereditarietà in C++ 2

```
class genitore {
    int x1;
public:
    int x2;
};
class figlio : genitore {
    int y1;
public:
    int y2;
    void prova();
};
void figlio::prova()
{
    y1 = 1;
    y2 = 2;
    x2 = 3; // non si può usare x1, perchè è di tipo privato e continua ad esserlo anche
            // per il figlio.
}
genitore x;
figlio y;
```

---

---

# Polimorfismo ed Ereditarietà in C++ 3

Una classe ha accesso solo ai metodi e attributi della classe genitore che non sono di tipo privato.

Gli attributi della classe genitore, quando ereditati, possono modificare le regole di visibilità.

Riferendoci all'esempio precedente, `x2` viene ereditato perchè inizialmente pubblico. La classe figlia lo usa (per esempio nel metodo `prova()`) ma l'attributo non è più visibile all'esterno della classe. Pertanto, l'istruzione `y.x2 = 200` non è corretta; mentre lo sono le istruzioni `y.y2 = 200` e `x.x2 = 150`.

In generale, la regola stabilisce che gli attributi, una volta ereditati, diventano privati. Per modificare questa regola, si utilizza l'istruzione `public`:

```
class figlio : public genitore {
```

Così tutti gli attributi e i metodi della classe genitore mantengono le stesse regole di visibilità. In questo modo, l'istruzione `y.x2 = 200` è corretta.

Ciò non toglie che un attributo privato del genitore continua a restare tale; l'istruzione `y.x1 = 200` continua a non essere corretta.

Quando le classi ereditano, può essere utile usare oltre a `private` e `public` una regola di visibilità intermedia detta `protected`. I metodi e gli attributi `protected` sono visti solamente nella classe dove sono definiti e anche nelle classi figlie, ma mai in altri punti del programma. Esempio:

---

---

# *Polimorfismo ed Ereditarietà in C++ 4*

```
class genitore {
    int x1;
protected:
    int x2;
};
class figlio : public genitore {
    int y1;
public:
    int y2;
    void prova();
};
void figlio::prova()
{
    y1 = 1;
    y2 = 2;
    x2 = 3;
}
genitore x;
figlio y;
```

---

---

# Polimorfismo ed Ereditarietà in C++ 5

Ciò comporta che `x2 = 3` contenuto nel metodo `figlio::prova()` continua ad essere corretto; mentre non lo è una istruzione del genere: `x.x2=20` perchè `x2` essendo `protected` non è visibile al di fuori delle classi genitore e figlie.

Ricordiamo che:

- le istruzioni `private`, `protected` e `public` hanno validità fino a quando si incontra un'altra istruzione di visibilità;
- normalmente, se non specificato diversamente, tutti gli attributi ed i metodi sono di tipo `private`;
- le classi figlie oltre ad aggiungere metodi ed attributi nuovi ed usare quelli della classe genitore da dove provengono, possono anche modificare le regole di visibilità dei metodi e degli attributi ereditati;

# Polimorfismo ed Ereditarietà in C++ 6

Una classe figlia può ridefinire i metodi e gli attributi delle classi genitori modificandoli opportunamente. Questo meccanismo di *overriding* maschera il metodo originale e può essere applicato sia ai metodi sia agli attributi.

L'overriding dei metodi e degli attributi può causare delle difficoltà quando dalla classe figlia si vuole accedere ai metodi o attributi mascherati nella classe genitori.

L'overriding dei costruttori segue regole un poco diverse:

- avviene non tra funzioni con lo stesso nome e con gli stessi argomenti, ma semplicemente tra costruttori con gli stessi argomenti;
- nel caso in cui si esegua l'overriding di un costruttore, il costruttore della classe figlia che ne maschera della classe genitore deve esplicitamente richiamare uno dei costruttori della classe genitore.
- infine, la sintassi con cui il costruttore della classe figlia chiama il costruttore della classe genitore è diversa dalla sintassi di chiamata dei metodi.

# *Polimorfismo ed Ereditarietà in C++ 7*

Il poliformismo è la capacità che ha un oggetto di una certa classe di contenere un oggetto di una classe diversa, purchè figlia della prima classe.

Questo è possibile perchè un oggetto figlio è come un oggetto genitore.

Il poliformismo può presentare dei problemi se usato insieme all'overriding.

