



UNIVERSITÀ  
DI PARMA

# oop: ereditarietà

*Alberto Ferrari*

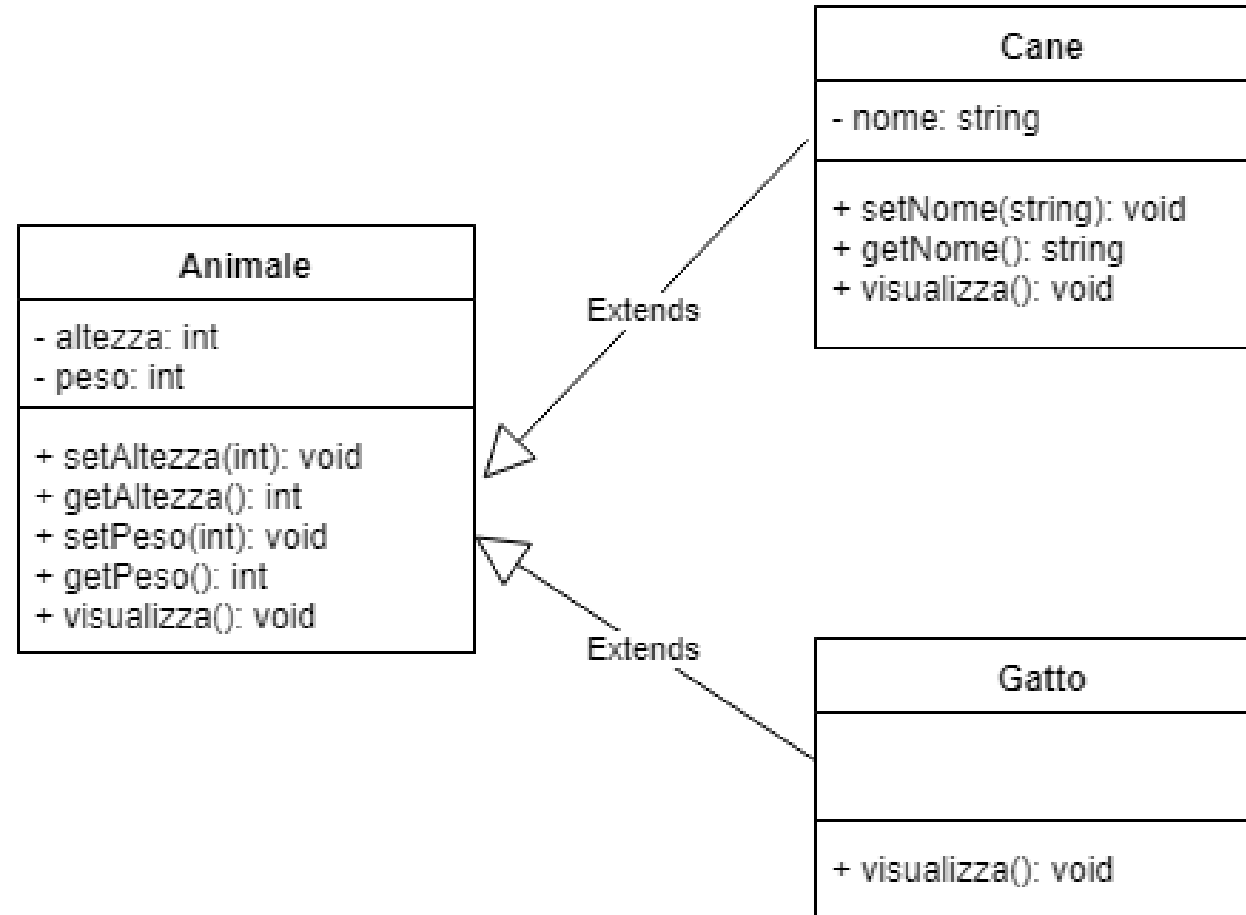
- l'**ereditarietà** permette di definire nuove classi partendo da classi sviluppate in precedenza
- una nuova classe (**classe derivata**) viene creata a partire da una classe esistente (**classe base**)
- la classe derivata viene definita esprimendo solamente le  **differenze**  che essa possiede rispetto alla classe base
- la classe derivata **eredita** le variabili membro e le funzioni membro della classe base

- la classe derivata può **aggiungere** variabili membro e funzioni membro
- la classe derivata può ***cambiare la definizione*** di una funzione membro ereditata
- possiamo avere ereditarietà per
  - ***estensione***  
(***aggiunta*** di nuove variabili e/o funzioni, *eventualmente overloading*)
  - ***ridefinizione***  
(***overriding*** di funzioni)

Animale
- altezza: int - peso: int
+ setAltezza(int): void + getAltezza(): int + setPeso(int): void + getPeso(): int + visualizza(): void

Cane
- altezza: int - peso: int - nome: string
+ setAltezza(int): void + getAltezza(): int + setPeso(int): void + getPeso(): int + setNome(string): void + getNome(): string + visualizza(): void

Gatto
- altezza: int - peso: int
+ setAltezza(int): void + getAltezza(): int + setPeso(int): void + getPeso(): int + visualizza(): void



# le classi della gerarchia in C++

## Animale.h e Animale.cpp

```
#ifndef ANIMALE_H
#define ANIMALE_H
class Animale {
public:
    Animale(int = 0 ,int = 0);
    int getAltezza() { return altezza; }
    void setAltezza(int val) {altezza=val;}
    int getPeso() { return peso; }
    void setPeso(int val) { peso = val; }
    void visualizza() const;
private:
    int altezza;
    int peso;
};
#endif // ANIMALE_H
```

```
#include "Animale.h"
#include <iostream>

using namespace std;

Animale::Animale(int a, int p):
    altezza(a), peso(p) {}

void Animale::visualizza() const {
    cout << "altezza " << altezza
        << " peso " << peso << endl;
}
```

```
#ifndef CANE_H
#define CANE_H
#include <Animale.h>
#include <string>
using namespace std;
class Cane : public Animale {
public:
    Cane(int=0, int=0,
        string nome="Pluto");
    string getNome() { return nome; }
    void setNome(string val) { nome=val; }
    void visualizza() const;
private:
    string nome;
};
#endif // CANE_H
```

```
#include "Cane.h"
#include<iostream>

using namespace std;

Cane::Cane(int a, int p, string n ):
    Animale(a, p) {
    setNome( n );
}

void Cane::visualizza() const {
    cout << "Sono un cane di nome: "
    << nome ;
    Animale::visualizza();
}
```

```
#ifndef GATTO_H_INCLUDED
#define GATTO_H_INCLUDED
#include <Animale.h>
#include <string>

using namespace std;
class Gatto : public Animale
{
    public:
        Gatto( int = 0 , int = 0);
        void visualizza() const;
    private:
};

#endif // GATTO_H_INCLUDED
```

```
#include "Gatto.h"
#include<iostream>

using namespace std;

Gatto::Gatto(int a, int p ): Animale(a, p)
{ }
void Gatto::visualizza() const
{
    cout << "Sono un gatto ";
    Animale::visualizza();
}
```



- un costruttore della classe base ***non viene ereditato***
- può essere ***invocato*** nella definizione del costruttore della classe derivata per inizializzare le variabili ereditate
- se non è invocato, il costruttore di ***default*** della classe base viene invocato ***automaticamente***

```
Cane::Cane(int a, int p, string n ): Animale(a, p)
{
    setNome( n );
}
```

- i ***membri privati*** della classe base ***non sono referenziabili*** nelle definizioni delle funzioni membro della classe derivata
  - verrebbe violato il principio di ***incapsulamento***
- le funzioni membro della classe derivata possono accedere alle variabili membro private della classe base tramite le funzioni ***accessor*** e ***mutator*** (se presenti)
- le ***funzioni membro private*** della classe base ***non*** sono ***accessibili*** (di fatto non sono ereditate)

- una variabile o funzione membro qualificata come *protected* può essere referenziata nelle funzioni membro di una classe derivata
- le variabili membro *protected* agiscono come se fossero *protected* in ogni classe derivata
- molti ritengono che l'uso di variabili membro *protected* **comprometta l'incapsulamento**
- è buona norma utilizzare *protected* **solo** quando assolutamente **necessario**

- una funzione *ridefinita* in una classe derivata ha lo **stesso numero e tipo di parametri** della funzione della classe base (**overriding**)
- una funzione *sovraccaricata* in una classe derivata ha un **diverso numero e/o tipo di parametri** rispetto alla funzione della classe base e la classe derivata ha entrambe le funzioni (**overloading**)

- una classe derivata può ridefinire una funzione della classe base
- è possibile invocare su un oggetto della classe derivata la **versione** della funzione data nella **classe base**
- si utilizza l'operatore ::, che in questo caso è **obbligatorio**, altrimenti la funzione chiamante continuerebbe in realtà a chiamare se stessa generando un loop

```
void Gatto::visualizza() const
{
    cout << "Sono un gatto ";
    Animale::visualizza();
}
```

- un **oggetto** di una classe **derivata** può essere usato **ovunque** può essere usato un **oggetto** della classe **base**
- un oggetto di una classe derivata ha ***più di un tipo***
- ***Cane is a Animale***

- oltre alle funzioni membro private *non vengono ereditati*
  - *costruttori*
  - *distruttori*
  - *costruttori di copia*
  - *operatori di assegnamento*
- se non vengono definiti vengono creati quelli di *default*

- relazione “*is a*”
  - esempio: un Gatto *is a* Animale
- relazione “*has a*”
  - esempio: un Computer *has a* Processore



```
class Cerchio{
public:
    Cerchio(double =1, int= 0, int =0 );
    ...
    double circonferenza();
    double area();
private:
    double raggio;
    int x;
    int y;
};
```

```
#define M_PI
3.14159265358979323846264338327950288

Cerchio::Cerchio(double r,int vx,int vy):
raggio(r), x(vx), y(vy)
{}

double Cerchio::circonferenza()
{
    return 2*M_PI*raggio;
}

double Cerchio::area()
{
    return M_PI*pow(raggio,2);
}
```

```
class Cilindro : public Cerchio
{
public:
    Cilindro(double=1,int=0,int=0,
              double=1);
    double getAltezza() { return altezza; }
    void setAltezza(double v) {altezza= v;}
    double supTotale();
private:
    double altezza;
};
```

```
Cilindro::Cilindro(double r, int vx,
                  int vy, double h)
    :Cerchio(r,vx,vy)
{
    altezza = h;
}

double Cilindro::supTotale()
{
    return area()*2+circonferenza()*altezza;
}
```

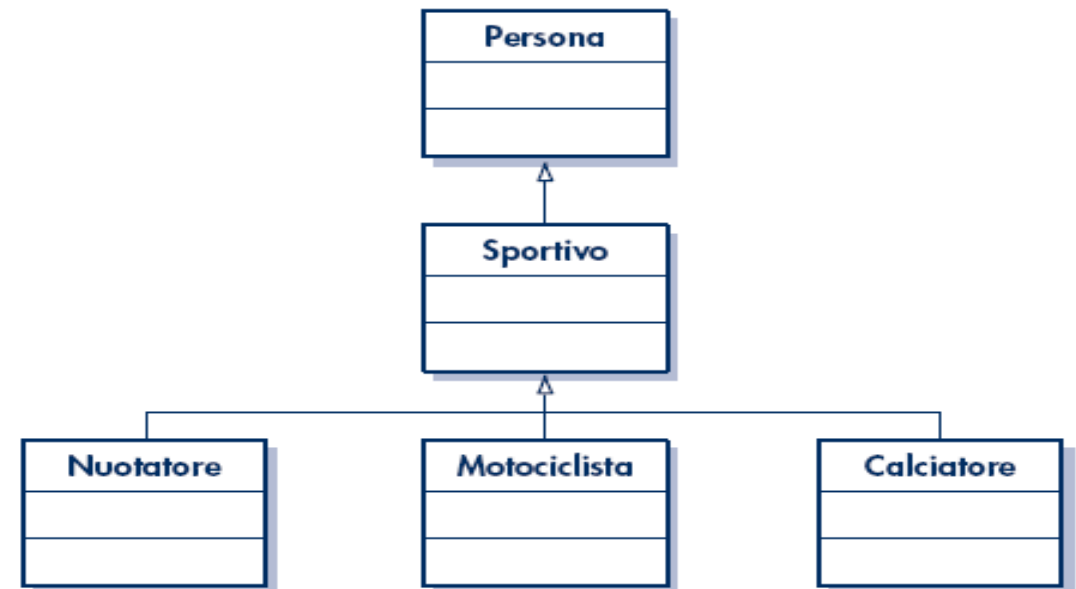
```
class Cilindro
{
    public:
        Cilindro(Cerchio , double = 1);
        double getAltezza() {return altezza;}
        void setAltezza(double v){altezza= v;}
        double supTotale();
    private:
        Cerchio base;
        double altezza;
};
```

```
Cilindro::Cilindro(Cerchio c, double h)
    : base(c), altezza(h) {}

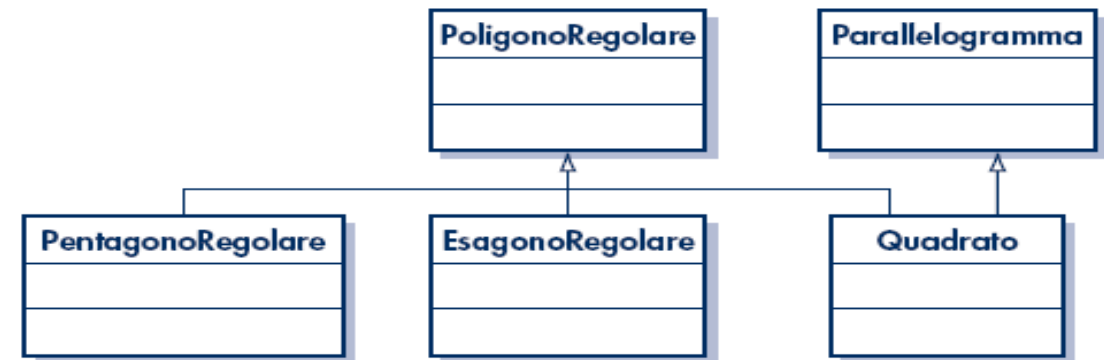
double Cilindro::supTotale()
{
    return base.area()*2
        +base.circonferenza()*altezza;
}
```

- ***ereditarietà protetta***: i membri pubblici della classe base sono protetti nella classe derivata quando sono ereditati
- ereditarietà privata: nessun membro della classe base può essere referenziato nella classe derivata
  - la relazione “is a” non è valida
  - sono raramente usate

- l'ereditarietà può estendersi a più livelli generando quindi una **gerarchia di classi**
- una classe derivata può, a sua volta, essere base di nuove sottoclassi
- **Sportivo** è sottoclasse di **Persona** ed è superclasse di **Nuotatore**, **Motociclista** e **Calciatore**
- nella parte alta della gerarchia troviamo le **classi generiche**, scendendo aumenta il **livello di specializzazione**



- una classe derivata può avere ***più di una classe base***
- possono esserci situazioni ***ambigue***
- richiede una ***conoscenza approfondita*** del linguaggio
- in ***alcuni linguaggi*** (es Java) ***non è ammessa*** l'ereditarietà multipla



- un **oggetto** di tipo **sottoclasse** è contemporaneamente e **automaticamente** anche di tipo **superclasse**
- quando è necessario utilizzare un oggetto di tipo superclasse è **possibile** utilizzare un oggetto di tipo **sottoclasse**
- **al contrario invece la regola non vale**
- ogni oggetto di tipo **Sportivo** è anche un oggetto di tipo **Persona**
  - *è vero che uno sportivo è una persona*
- **non è vero il contrario**
  - *una persona non è necessariamente uno sportivo*

- è possibile dichiarare un ***puntatore*** ad un oggetto di una ***classe***
- l'operatore ***new*** alloca memoria per l'oggetto e ***chiama il costruttore***
- ***delete*** rilascia la memoria e chiama il ***distruzione*** della classe
- esempio

```
Cerchio *pCerchio;
```

```
pCerchio = new Cerchio;
```

```
pCerchio = new Cerchio(10.5, 45, 30);
```

```
...
```

```
delete pCerchio;
```



- l'operatore **\*** (*indirection operator*) ritorna un sinonimo dell'oggetto a cui il suo operando (un puntatore) punta
- tramite questo operatore è possibile **accedere** alle funzioni membro dell'oggetto puntato

```
Cerchio *pc1;
```

```
pc1 = new Cerchio(10,2,4);
```

```
cout << "area " << (*pc1).area() << endl;
```

```
cout << "area " << pc1->area() << endl;
```

- ***puntatore->metodo()*** e ***(\*puntatore).metodo()*** sono due forme di scrittura equivalenti

funzioni virtual

**polimorfismo**

- *polimorfismo e funzioni virtuali* sono un meccanismo fondamentale per realizzare *sistemi estensibili*
  - consentono di trattare gli *oggetti di tutte le classi* di una gerarchia come se fossero oggetti della classe base
- il risultato è la scrittura di programmi *più semplici* (*meno branching logic*), in cui viene favorito il testing ed il mantenimento del codice

- quando una *funzione* membro di una *classe base* è dichiarata *virtual*
- se la funzione è *chiamata* tramite un *oggetto*, la risoluzione del riferimento avviene a *tempo di compilazione* (*static binding*) e la funzione chiamata è quella della classe dell'oggetto
- se la funzione è *chiamata* tramite un *puntatore* il programma sceglie a *tempo di esecuzione* la funzione della classe appropriata (*late binding* o *dynamic binding*)

```
class ChessPiece {
    public:
        ChessPiece();
        virtual ~ChessPiece();
        virtual void move();
    private:
        char column;
        int row;
        char color;
        std::string name;
};
...
void ChessPiece::move() {
    std::cout << "No piece" << std::endl;
}
```

```
class Bishop : public ChessPiece {
    public:
        Bishop();
        virtual ~Bishop();
        void move();
    private:
};
...
Bishop::Bishop() {
    setName("Bishop");
}
...
void Bishop::move() {
    std::cout << "Bishop @ " << getColor()
        << ", " << getRow() << std::endl;
}
```

```
int main()
{
    ChessPiece cp;
    Bishop b;
    cp = b;
    cp.move();          // output no piece
    ChessPiece *pPiece;
    pPiece = new Bishop();
    pPiece->move();     // output Bishop @ ...
    return 0;
}
```

- **cp = b**
  - assegnamento a una variabile superclasse di una variabile sottoclasse
- **cp.move()**
  - static binding
  - eseguito il metodo di ChessPiece
- **pPiece = new Bishop()**
  - un puntatore a una superclasse punta a un oggetto di una sottoclasse
- **pPiece->move()**
  - dynamic binding
  - eseguito il metodo di Bishop

- capacità di oggetti appartenenti a classi che derivano da una classe base comune di ***rispondere in modi diversi*** alla chiamata di una certa funzione
- si implementa tramite le ***funzioni virtuali***:
  - alla chiamata di una funzione virtuale tramite un ***puntatore*** alla classe base, il programma sceglie la ridefinizione corretta della funzione nella classe derivata appropriata
- ***aumento di generalità***: è il runtime a doversi occupare delle specificità, non il programmatore
- ***estendibilità***: il codice è scritto indipendentemente dai tipi derivati, nuovi tipi possono essere aggiunti senza dover apportare modifiche a quanto già sviluppato

- se una classe ha una o più funzioni membro virtuali il compilatore crea una ***tabella*** che per ogni ***funzione virtuale*** contiene l'***indirizzo*** in memoria del codice della funzione
- quando viene creato un ***oggetto*** della classe, la sua descrizione contiene un ***puntatore alla tabella*** delle funzioni virtuali
- quando una ***funzione virtuale*** viene chiamata usando un ***puntatore*** all'oggetto, il sistema runtime ***usa la tabella*** per decidere ***quale definizione*** della funzione usare
- le funzioni virtuali introducono ***overhead***



- se una funzione è *virtuale pura* la sua definizione non è necessaria
- una classe con una o più funzioni virtuali pure è detta *astratta*
- una *classe astratta* può essere usata solo come classe base per derivare altre classi, *non si possono creare oggetti*

```
class ChessPiece {  
    public:  
        ChessPiece();  
        virtual void move() = 0;  
    private:  
        char column;  
        int row;  
        char color;  
        std::string name;  
};
```