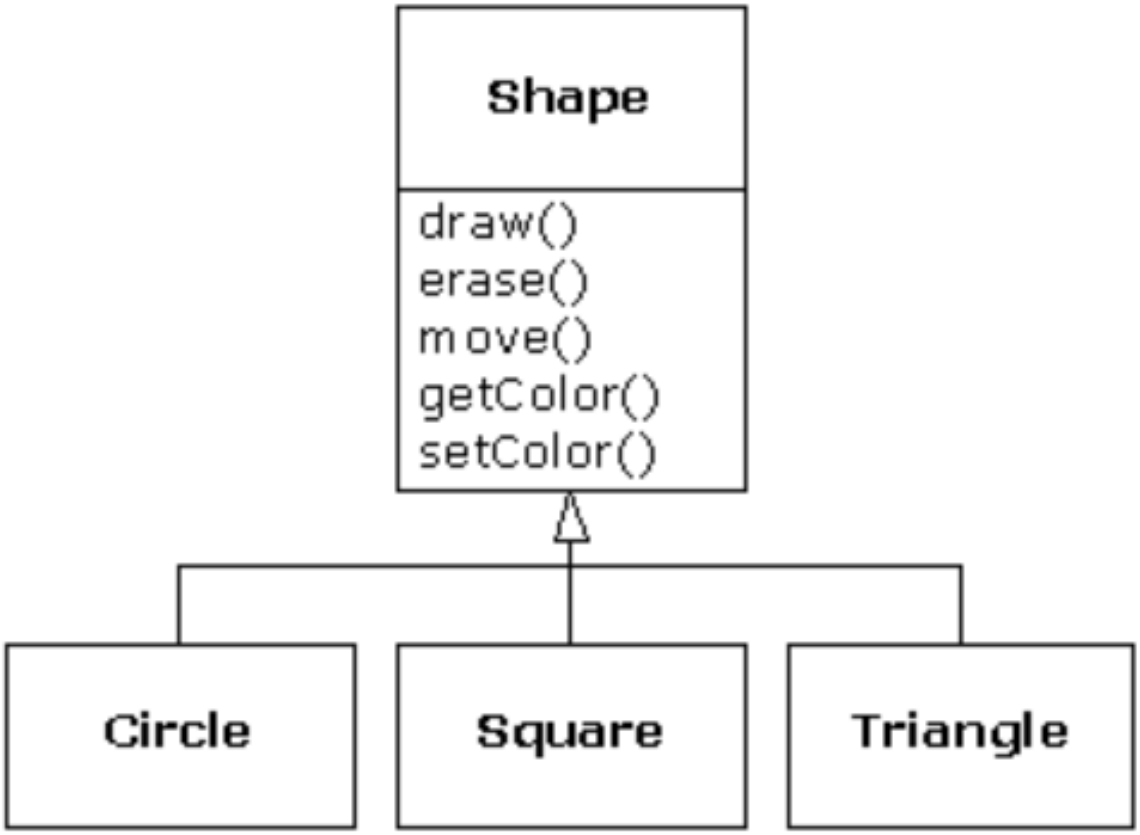
Capitolo 12

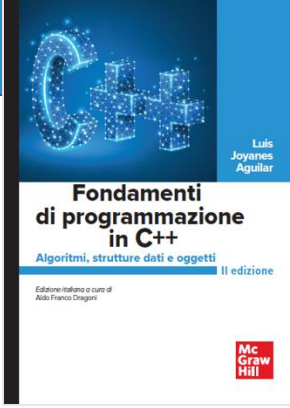
Classi derivate: ereditarietà e polimorfismo

Pag 315-340

Presenta: Prof. Misael Mongiovì

ereditarietà



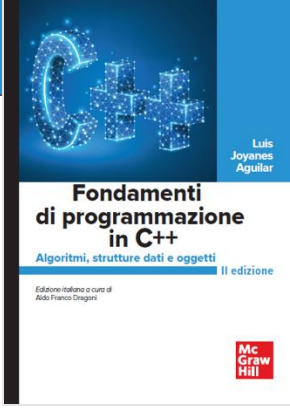


classi derivate

- una *classe derivata* eredita attributi e metodi dalla *classe base* già esistente
- anche l'ereditarietà è caratterizzata da uno *specificatore di accesso* (anche detto *tipo di ereditarietà*) che può essere `public`, `private` (default) o `protected`

```
class ClasseDerivata : [spec_acc] ClasseBase
{
  membri;
};
```

12. Classi derivate: ereditarietà e polimorfismo

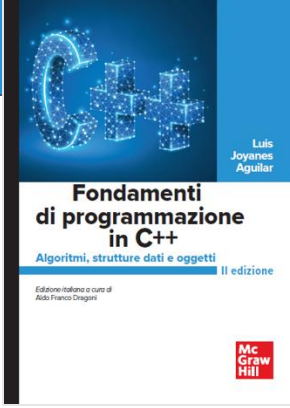



```
class Pubblicazione {
  public:
     void InserireEditore(const char *s);
     void InserireData(unsigned long dat);
  private:
     string editor;
     unsigned long data;
};

class Rivista : public Pubblicazione {
  public:
     void InserireTiratura(unsigned n);
     void InserireVenduto(unsigned long n);
  private:
     unsigned tiratura;
     unsigned long venduto;
};

class Libro : public Pubblicazione {
  public:
     void InserireISBN(const char *s);
     void InserireAutore (const char *s);
  private:
     string ISBN;
     string autore;
};
```

esempi di classi derivate

tipi di ereditarietà

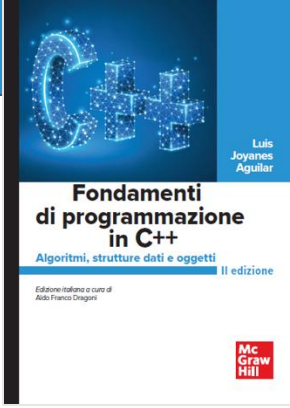
Tipo di ereditarietà	Accesso a membro classe base	Accesso a membro classe derivata
`public`	`public`	`public`
	`protected`	`protected`
	`private`	inaccessibile
`protected`	`public`	`protected`
	`protected`	`protected`
	`private`	inaccessibile
`private`	`public`	`private`
	`protected`	`private`
	`private`	inaccessibile



tipi di ereditarietà

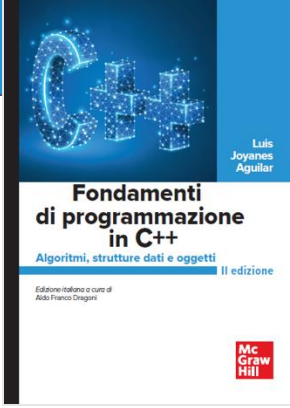
- `public` è il tipo di ereditarietà più utilizzato
- `private` è il tipo di ereditarietà per default
  quindi se si dimentica di specificare il tipo di ereditarietà, gli elementi pubblici della classe base saranno privati in quella derivata
- `protected` mantiene accessibili i membri pubblici e protetti nella classe derivata occultandoli all'esterno

12. Classi derivate: ereditarietà e polimorfismo




```
class ogg_geom {
public:
   ogg_geom(float x=0, float y=0) : xC(x), yC(y) {}
   void stampacentro()
   { cout << xC << " " << yC << endl; }
protected:
   float xC, yC;
};
class cerchio : public ogg_geom {
public:
   cerchio(float x_C, float y_C, float r) : ogg_geom (x_C, y_C)
   { raggio = r; }
   float area() {return PI * raggio * raggio; }
private:
   float raggio;
};
class quadrato : public ogg_geom {
public :
   quadrato(float x_C,float y_C,float x,float y) : ogg_geom(x_C,y_C)
   { x1 = x; y1 = y; }
   float area()
   {
      float a, b;
      a = x1 - xC;
      b = y1 - yC;
      return 4*a*b;
   }
private:
   float x1, y1;
};
```

esempio ereditarietà pubblica



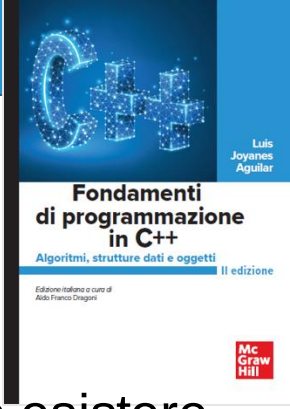
ereditarietà privata

- con l'ereditarietà privata un utente della classe derivata non ha accesso ad alcun elemento della classe base:

```
class ClasseDerivata : private ClasseBase {
public:
    // sezione pubblica
protected:
    // sezione protetta
private:
    // sezione privata
};
```

i membri pubblici e protetti della classe base diventano membri privati della classe derivata

- l'ereditarietà privata è quella di default; essa occulta la classe base all'utente perché sia possibile cambiare l'implementazione della classe base o eliminarla del tutto senza richiedere alcuna modifica all'utente dell'interfaccia



costruttori ed ereditarietà

- bisogna che il costruttore della classe base venga chiamato per creare un oggetto (della classe base) prima che si attivi il costruttore della classe derivata (l'oggetto della classe base deve esistere prima di diventare un oggetto della classe derivata)

```
class B1 {
public:
   B1() { cout << "oggetto della classe B1" << endl; }
};
class B2 {
public:
   B2() { cout << "oggetto della classe B2" << endl; }
};
class D : public B1, B2 {
public:
   D() { cout << "oggetto della classe D derivata da B1 e B2" << endl; }
};
D d1;
```

Questo esempio scrive a schermo:

```
oggetto della classe B1
oggetto della classe B2
oggetto della classe D derivata da B1 e B2
```

costruttore di una classe derivata

```
ClasseDer::ClasseDer(paramD):ClasseBase(paramB), Inizial {
   // corpo del costruttore della classe derivata
};
```

supponiamo la classe `Punto3D` sia una classe derivata dalla classe `Punto`:

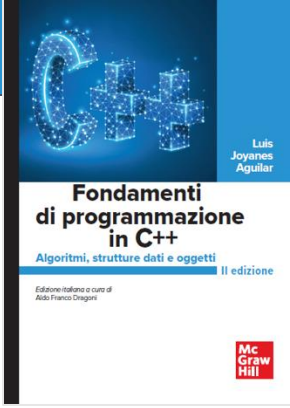
`Punto3D::Punto3D(listaPar):inizializzatore-costruttore`

nell'implementazione del costruttore di `Punto3D` si passano i parametri `xv` e `yv` al costruttore di `Punto`.

```
class Punto {
public:
   Punto(int xv, int yv);
   // ...
};
class Punto3D: public Punto {
public:
   Punto3D(int xv, int yv, int zv);
   void FissareZ();
private:
   int z;
};
Punto3D::Punto3D(int xv, int yv, int zv): Punto(xv, yv){
   FissareZ(zv);
}
```

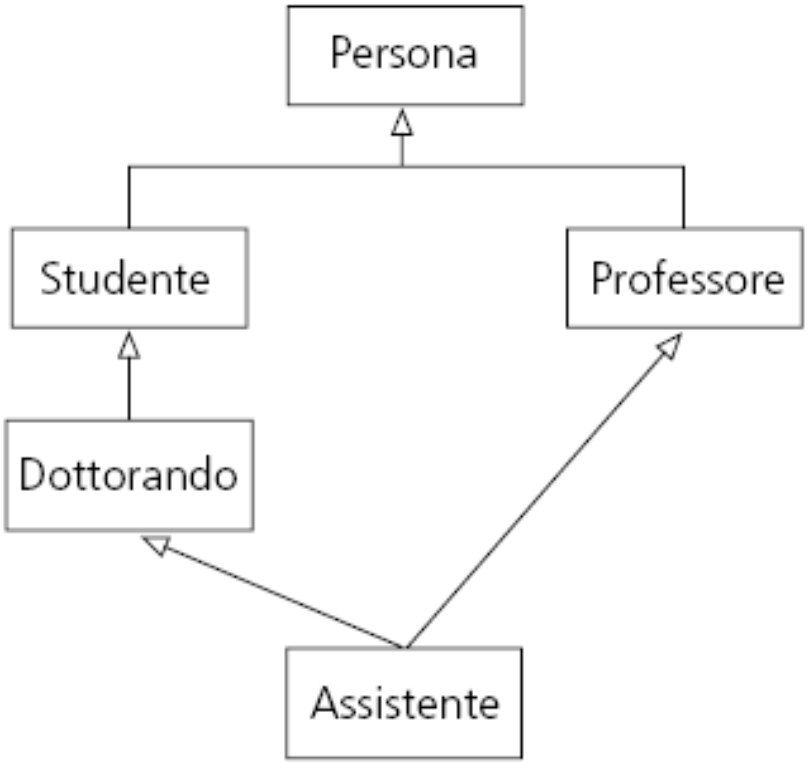
distruttori ed ereditarietà

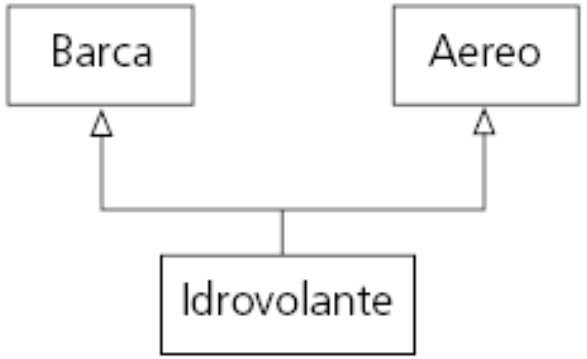
- i distruttori non si ereditano, ma se necessario si genera un distruttore di default
- si utilizzano normalmente solo quando un corrispondente costruttore ha assegnato spazio in memoria che deve essere liberato
- si gestiscono come i costruttori ma tutto va fatto in ordine inverso, vale a dire che s'inizia ad eseguire il distruttore dell'ultima classe derivata

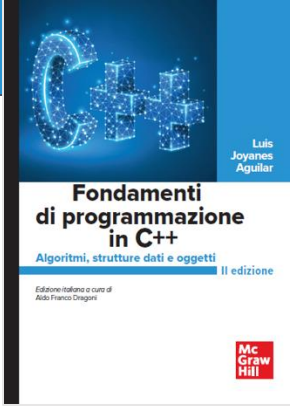


ereditarietà multipla

- una classe può ereditare attributi e comportamento di più di una classe base







ereditarietà multipla

- la sintassi è:

```
class Derivata : [virtual][tipo_accesso] Base1,
                 ...
                 [virtual][tipo_accesso] Basen {
  public:
  // sezione pubblica
  private:
  // sezione privata
  };
```

dove:

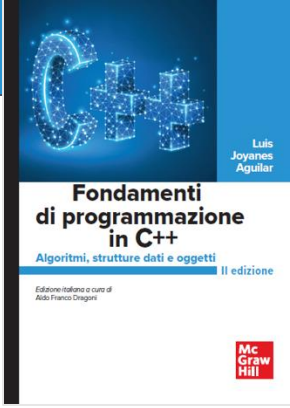
Derivata: nome della classe derivata

tipo_accesso: `public`, `private` o `protected`

Base1, Base2,... : classi base con nomi differenti

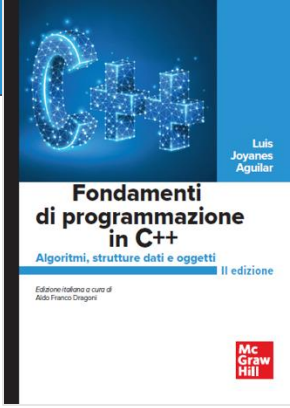
virtual.. : è opzionale e specifica una classe base compatibile

- Metodi o attributi che abbiano lo stesso nome in *Base1, Base2, Basen*, costituiranno motivo di ambiguità



binding dinamico

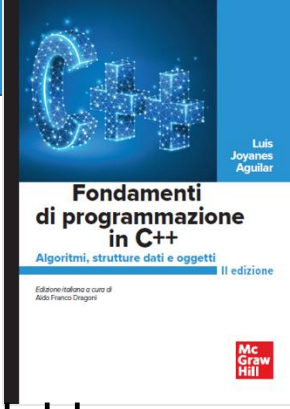
- *binding*
  connessione tra chiamata di funzione e codice che l'implementa
  - *statico*: se il collegamento avviene in fase di compilazione
  - *dinamico*: se la connessione avviene durante l'esecuzione
    solo durante l'esecuzione del programma si determinerà il binding effettivo (tipicamente tramite il valore di un puntatore ad una classe base) tra le diverse possibilità (una per ogni classe derivata)
- il binding dinamico offre un alto grado di flessibilità e praticità nella gestione delle gerarchie di classi, ma è meno efficiente di quello statico
- in C++
  - il binding per default è statico
  - per specificare il binding dinamico si fa precedere la dichiarazione della funzione dalla parola riservata **`virtual`**



funzioni virtuali

- `virtual` anteposto alla dichiarazione di una funzione indica al compilatore che
  - essa può essere definita in una classe derivata
  - la funzione, se invocata tramite un puntatore alla classe base, avrà comportamenti differenti in base alla classe della specifica istanza
- si deve qualificare virtual un metodo di una classe solo se esiste la possibilità che da quella classe se ne possano derivare altre
- le funzioni virtuali servono nella dichiarazione di classi astratte e nel polimorfismo

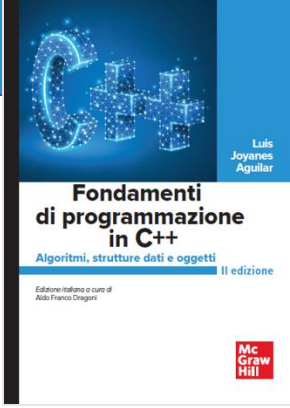
```
class figura {
  public:
   virtual double calcola_area();
   virtual void disegna();
```



esempio funzioni virtuali

- ogni classe derivata deve definire le sue proprie versioni delle funzioni dichiarate virtuali nella classe base; per esempio, le classi `cerchio` e `rettangolo` derivano dalla classe `figura`, debbono entrambe definire i propri metodi `calcola_area` e `disegna`

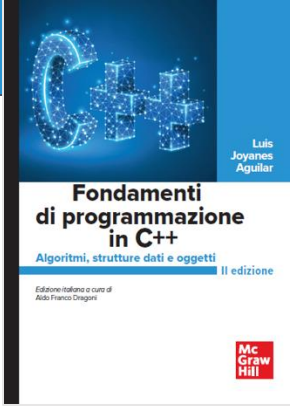
```
class cerchio : public figura {
   public:
      double calcola_area();
      void disegna();
   private:
      double xc, yc;          // coordinata del centro
      double raggio;          // raggio del cerchio
};

#define PI 3.14159

double cerchio::calcola_area() { return(PI * raggio * raggio);}
void cerchio::disegna() {
// funzione "disegna"
}
```



polimorfismo

- proprietà in base alla quale oggetti differenti possono rispondere in maniera diversa ad una stessa invocazione

```
class figura {
   public:
   virtual void Copia();
   virtual void Disegna();
   virtual double Area();
   };
class cerchio : public figura {
   public:
   void Copia();
   void Disegna();
   double Area();
   };
class rettangolo : public figura {
   public:
   void Copia();
   void Disegna();
   double Area();
   };
```



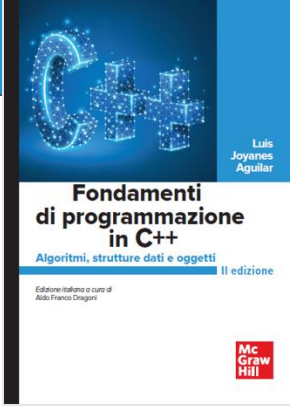
polimorfismo

- si può passare lo stesso messaggio ad oggetti differenti:

```
cerchio MioCerchio;
rettangolo MioRettangolo;
switch(...) {
case ...:
   MioCerchio.Disegna(); d = MioCerchio.Area();
   break;
case ...:
   MioRettangolo.Disegna(); d = MioRettangolo.Area();
   break;
};
```

- oppure con il binding dinamico

```
// crea e inizializza un array di figure
figura* figure[]= {new cerchio, new rettangolo, new triangolo};
...
figure[i].Disegna();
```



vantaggi del polimorfismo

- permette di utilizzare una stessa interfaccia (come i metodi `Disegnare` ed `Area`) per lavorare con oggetti di diverse classi
- rende il sistema più flessibile
- C++ conserva i vantaggi della compilazione statica mentre altri linguaggi OOP adottano un polimorfismo assoluto che si scontra con l'idea della verifica dei tipi
- le sue applicazioni più frequenti sono:
  - specializzazione di classi derivate
  - strutture di dati eterogenei
  - gestione delle gerarchie di classi