# classe derivata


- eredita attributi e metodi dalla *classe base* già esistente
- nasce dal bisogno di costruire classi specifiche (da classi generali)

- la dichiarazione deve includere il nome della classe base da cui deriva
  - eventualmente, uno specificatore indicante il tipo di ereditarietà (`public`, `private` o `protected`) secondo la seguente sintassi:

```
class ClasseDerivata : specific_accesso_opz ClasseBase {
    membri;
};
```

*Ereditarietà pubblica*: specificatore di accesso `public`;

- significa che i membri pubblici della classe base sono tali anche per quella derivata

*Ereditarietà privata*:

- specificatore di accesso `private`

*Ereditarietà protetta*:

- specificatore di accesso `protected`

Se si omette lo specificatore di accesso si assumerà per default `private`

# esempio di classi derivate


```
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;
};
```

# esempio di classi derivate


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

- L'oggetto `libro` contiene membri dato e metodi della classe `Pubblicazioni`
  - Inoltre contiene `ISBN` e `autore`
- Senza l'ereditarietà dovremmo riscrivere la nuova classe libro da zero

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

# tipi di ereditarietà


- In una classe, gli elementi pubblici sono accessibili a tutte le funzioni, quelli privati sono accessibili soltanto ai membri della stessa classe e quelli protetti possono essere acceduti anche da classi derivate (*proprietà dell'ereditarietà*)
- Tre tipi di ereditarietà: *pubblica*, *privata* e *protetta*,
  - la più utilizzata è la prima
- Una classe derivata non può accedere a variabili e funzioni private della sua classe base
- Per occultare dettagli una classe base utilizza normalmente elementi protetti invece che elementi privati
- Usando ereditarietà pubblica gli elementi protetti sono accessibili alle funzioni membro di tutte le classi derivate
- per default, l'ereditarietà è privata;
  - se si dimentica la parola riservata `public`, gli elementi della classe base saranno inaccesibili

# ereditarietà pubblica


- *ogni* classe derivata ha accesso agli elementi pubblici e protetti della sua classe base
- gli elementi pubblici si ereditano come elementi pubblici; gli elementi protetti come protetti
- si rappresenta con lo specificatore `public` nella derivazione di classi
- Guardiamo subito un esempio

# 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

# ereditarietà protetta


- i membri pubblici e protetti della classe base diventano membri protetti della classe derivata ed i membri privati della classe base diventano inaccessibili

| 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 |

# costruttori ed ereditarietà


- una classe derivata è una specializzazione della corrispondente classe base,
- il costruttore della classe base deve essere chiamato 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 << "C-B1" << endl; }
};

class B2 {
public:
   B2() { cout << "C-B2" << endl; }
};

class D : public B1, B2 {
public:
   D() { cout << "C-D" << endl; }
};

D d1;
```

Questo esempio visualizza:

```
C-B1
C-B2
C-D
```


# Riassumendo: Sintassi del Costruttore

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

- Vediamo subito un esempio
- Costruiamo le classi Punto e Punto3D

# distruttori ed ereditarietà

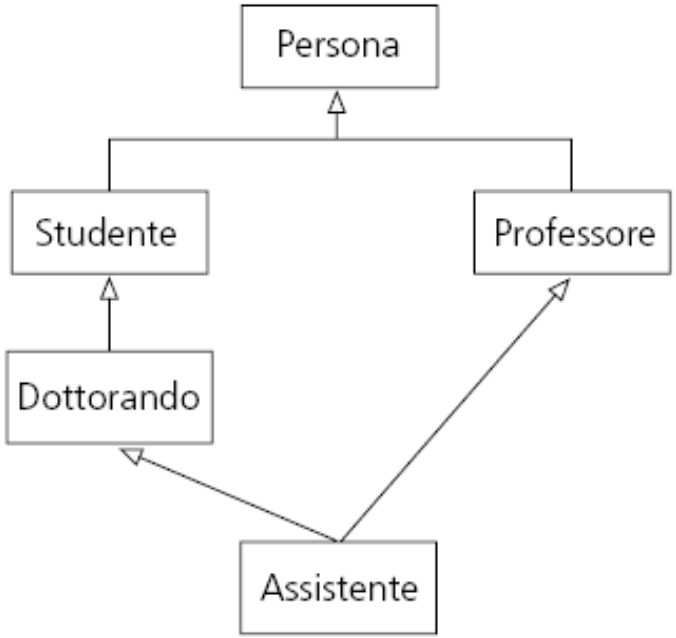

- i distruttori non si ereditano, ma se necessario si genera un distruttore di default

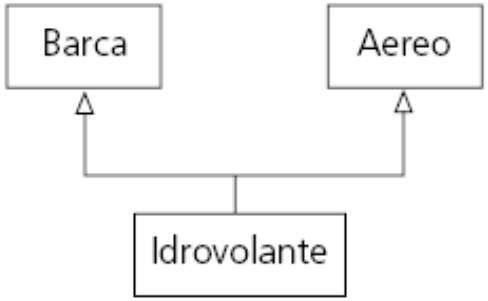
- si utilizzano normalmente solo quando un corrispondente costruttore ha assegnato spazio in memoria che deve essere liberato
- si gestiscono come i costruttori
  - tutto va fatto in ordine inverso: s'inizia ad eseguire il distruttore dell'ultima classe derivata
- Vediamo un esempio

# ereditarietà multipla

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



# ereditarietà multipla


- Una classe derivata da più classi base
- Sintatticamente più complessa di quella base

```
class Derivata :   [virtual][tipo_accesso] Base1,
                   [virtual][tipo_accesso] Base2,
                   [virtual][tipo_accesso] Basen {
public:
   // sezione pubblica
private:
   // sezione privata
...
};
```

*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.

Funzioni o dati membro che abbiano lo stesso nome in *Base1*, *Base2*, *Basen*, costituiranno motivo di ambiguità

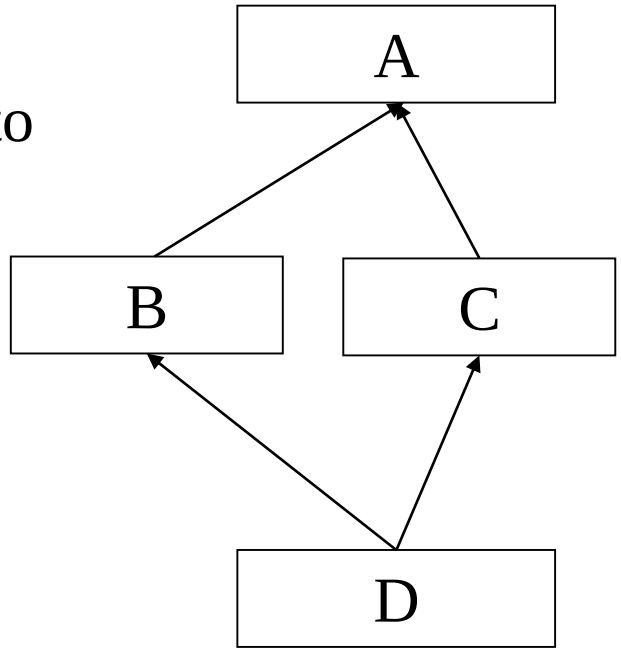
 

# Uso di virtual: dettagli

Problema del diamante

- B e C ereditano dalla classe A e la classe D eredita sia da B che da C
- se un metodo in D chiama un metodo definito in A, da quale classe viene ereditato?

Differenti linguaggi di programmazione hanno risolto quest'inconveniente in modi diversi.

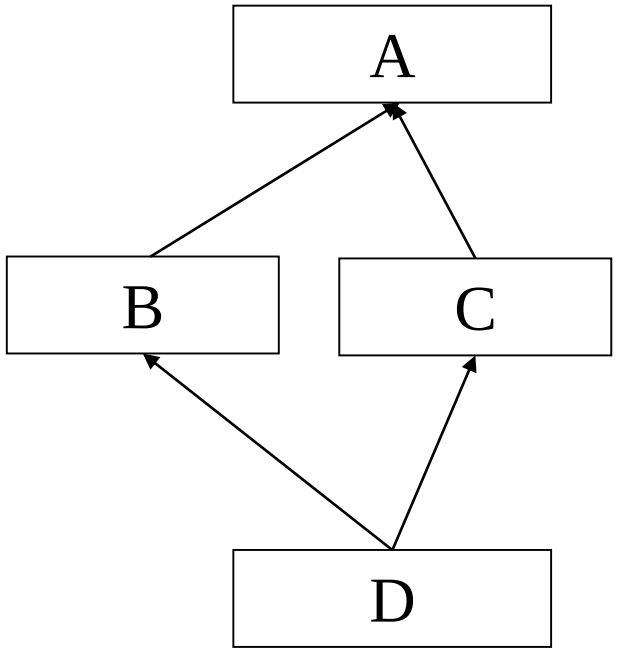


# Uso di virtual: dettagli

- C++, per default, segue ogni percorso (di ereditarietà) separatamente
  - D conterrà due (separati) oggetti di A

- Se le ereditarietà da A a B (e da C ad A) sono "virtual", C++ crea un solo oggetto A



# Ambiguità


```
class Finestra {
private:
    …
public:
    void apri();
    …
}
```

```
class Rubinetto {
private:
    …
public:
    void apri();
    …
}
```

- le due funzioni potrebbero essere implementate in modo diverso
- se creassimo

```
class FinRub : public Finestra, public Rubinetto {…}
FinRub f;

f.apri();          // errore
f.Rubinetto::apri();
f.Finestra::apri();
```


# Precedenza

- Funzioni di classi derivate che hanno lo stesso nome di altre funzioni in classi base
- Ereditarietà semplice: le funzioni delle classi derivate hanno precedenza sulle base.

```
class Base {
public:
    void f(int);
    void f(char);
};
```

```
class Derivata: public Base {
public:
    void f(char);
};
```

```
Derivata d;

d.f(4);       // errore dato che Base::f(int) è occultata da
              // Derivata.f(char)
d.f('A');
```

# binding dinamico


- per *binding* s'intende la connessione tra la chiamata di una funzione ed il codice che l'implementa
- *Statico*:
  - il collegamento avviene in fase di compilazione,
  - Tipico di molti linguaggi imperativi
- *Dinamico*
  - la connessione avviene durante l'esecuzione
  - fa sì che il codice da eseguire verrà determinato solo all'atto della chiamata;
  - 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)

# binding dinamico vs binding statico


- Vantaggio principale (del binding dinamico): offre un alto grado di flessibilità e praticità nella gestione delle gerarchie di classi
- Svantaggio principale: è meno efficiente di quello statico
- I linguaggi più strettamente OO offrono solo binding dinamico
- In C++ il binding per default è quello 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 e che, in questo caso, la funzione sarà invocata direttamente tramite un puntatore
- si deve qualificare un metodo di una classe con `virtual` solo quando 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 calcolare_area(void) const;
   virtual void disegnare(void) const;
   // altre funzioni membro che definiscono un'interfaccia
   // per tutti i tipi di figure geometriche
};
```

# funzioni virtuali


- ogni classe derivata deve definire le sue proprie versioni delle funzioni dichiarate virtuali nella classe base;
- se le classi `cerchio` e `rettangolo` derivano dalla classe `figura`, debbono entrambe definire le funzioni membro `calcolare_area` e `disegnare`

```
class cerchio : public figura
{
   public:
      virtual double calcolare_area(void) const;
      virtual void disegnare(void) const;
      // ...
   private:
      double xc, yc;         // coordinata del centro
      double raggio;          // raggio del cerchio
};
#define PI 3.14159       // valore di "pi"
// Implementazione di calcolare_area
double cerchio::calcolare_area(void) const
{
   return(PI * raggio * raggio);
}
// Implementazione della funzione "disegnare"
void cerchio::disegnare(void) const
{
   // ...
}
```

# funzioni virtuali: binding statico


- Consideriamo ancora l'esempio delle figure;

```
cerchio c1;
rettangolo r1;
double area= c1.calcolare_area()
```

- Il compilatore capisce che si richiede di invocare la funzione relativa alla classe `cerchio`
- Chiama direttamente questa funzione (collegata ad uno specifico codice) durante la compilazione
  - Binding statico

# binding dinamico tramite funzioni virtuali


- Consideriamo il caso in cui le funzioni vengono chiamate tramite un puntatore a `figura` del tipo:

```
figura* s[10];  // 10 puntatori ad oggetti figura
int i, numfigure = 10;
// crea figure e immagazzina puntatori nell'array s
// disegna le figure
   for (i = 0; i < numfigure; i++)
      s[i] -> disegnare();
```

- Binding Dinamico: il compilatore C++ non può sapere l'implementazione specifica della funzione `disegnare()` che sarà chiamata a tempo d'esecuzione

# binding dinamico tramite funzioni virtuali


```
figura* s[10];  // 10 puntatori ad oggetti figura
int i, numfigure = 10;
// crea figure e immagazzina puntatori nell'array s
// disegna le figure
   for (i = 0; i < numfigure; i++)
      s[i] -> disegnare();
```

- Un puntatore a una classe derivata è anche un puntatore alla classe base
- Possiamo utilizzare un riferimento o un puntatore ad una classe base al posto di uno alla classe derivata, senza effettuare conversione esplicita di tipi
- Il reciproco non è vero!
  - non si può utilizzare un riferimento o un puntatore alla classe derivata invece di un riferimento o un puntatore a un'istanza di una classe base

# polimorfismo


- è la proprietà in base alla quale oggetti differenti possono rispondere in maniera diversa ad uno stesso messaggio

```
class figura {
   tipoenum tenum;          //tipoenum è un tipo enumerativo
public:
   virtual void Copiare();
   virtual void Disegnare();
   virtual double Area();
};
class cerchio : public figura {
   ...
public:
   void Copiare();
   void Disegnare();
   double Area();
};
clas rettangolo : public figura {
   ...
public:
   void Copiare();// il polimorfismo permette ad oggetti differenti
                  // di avere metodi con lo stesso nome
   void Disegnare();
   double Area();
};
```

# polimorfismo


- si può passare lo stesso messaggio ad oggetti differenti:

```
switch(...) {
...
case Cerchio:
   MioCerchio.Disegnare();
   d = MioCerchio.Area();
   break;
case Rettangolo:
   MioRettangolo.Disegnare();
   d = MioRettangolo.Area();
   break;
...
};
```

usa binding statico: in fase di compilazione si sceglie quale tipo di oggetto prendere

- con binding dinamico:

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

# Osservazioni sul polimorfismo


- permette quindi di utilizzare una stessa interfaccia (come i metodi `Disegnare` ed `Area`) per lavorare con oggetti di diverse classi

- può essere implementato con array di puntatori a oggetti di classi (derivate) diverse che condividono metodi con lo stesso nome

Regole

- Creare una gerarchia di classi con le operazioni importanti definite nei metodi dichiarati `virtual` nella classe base
- Implementare i metodi virtuali nelle classi derivate
- Riferire gli oggetti di tali classi tramite riferimenti o puntatori

# vantaggi del polimorfismo


- Rende il sistema più flessibile
- C++ conserva anche i vantaggi della compilazione statica mentre altri linguaggi dell'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