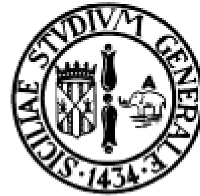


Polimorfismo, Relazioni e Inheritance

Corso di C++

INFN – LNS

13 Dicembre 2010

Corrado Santoro

Polimorfismo

- Ripartiamo dalla classe “Punto”
- Il costruttore richiede due parametri (le coordinate del punto)
- Se noi non conosciamo le coordinate al momento della creazione della variabile?
- Possiamo avere un “costruttore vuoto”?

```
class Punto {
...
public:
        Punto(int initial_x, int initial_y);
...
};
…
  Punto p(??,??);
```

Polimorfismo

- Ogni metodo di una classe può avere “più forme” (**polimorfismo**)
- Dato un metodo A, posso dichiarare un metodo A' con lo stesso nome di A **ma con parametri differenti (tipo e numero)**
- E' utile quando una stessa operazione potrebbe ricevere differenti parametri e quindi comportarsi diversamente
- Esempio: il metodo “intersect” della classe Cerchio
  - Se riceve un Cerchio, controlla se i due Cerchio si intersecano
  - Se riceve un Punto, controlla se il punto è interno al Cerchio
- L'operazione è “concettualmente” la stessa, ma cambia di funzionamento a seconda del parametro fornito

Polimorfismo

- Possiamo avere un costruttore senza parametri? **SI!**

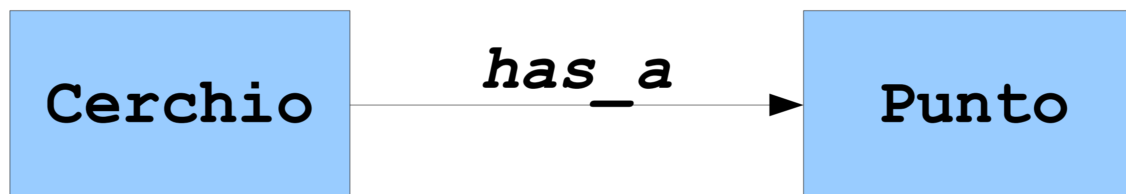
```
class Punto {
        int X, Y;
public:
       Punto();
       Punto(int initial_x, int initial_y);
...
};
…
Punto::Punto()
{
  X = 0; Y = 0;
}

Punto::Punto(int initial_x, int initial_y)
{
  X = initial_x; Y = initial_y;
}

  Punto p1;
  Punto p2(20,30);
```

Relazioni: Le classi “Punto” e “Cerchio”

- La classe Punto
  - rappresenta un punto dello spazio 2D
  - possiede le coordinate X e Y
- La classe Cerchio
  - Rappresenta un cerchio nello spazio 2D
  - Possiede le coordinate del centro e il raggio
- Ma cos'è il “centro del cerchio”?
  - E' un punto esso stesso
  - E' quindi rappresentabile con un oggetto di classe Punto



Punto e Cerchio … insieme

```
class Punto {
      int X;
      int Y;
public:
       Punto();
       Punto(int initial_x, int initial_y);
       void setXY(int newX, int newY);
       void move(int offsetX, int offsetY);
       int getX();
       int getY();
};

class Circle {
      Punto center;
      int radius;
public:
       Circle(int initial_x, int initial_y, int r);
       void setXY(int newX, int newY);
       void setRadius(int r);
       float circumference();
       float area();
};
```

Facciamo un “progetto”

- Per ogni classe:
  - Definizione
  - Implementazione
- Se la classe B (Cerchio) deve usare la classe A (Punto), la classe B ha bisogno **solo** della definizione di A, non della sua implementazione
- **Possiamo separare definizione e implementazione in file sorgenti differenti?**

Facciamo un “progetto”

`punto.h` — Punto (definizione)

`circle.h` — Cerchio (definizione)

`punto.cpp` — Punto (implementazione)

`circle.cpp` — Cerchio (implementazione)

`main.cpp` — Main Program

Compiler

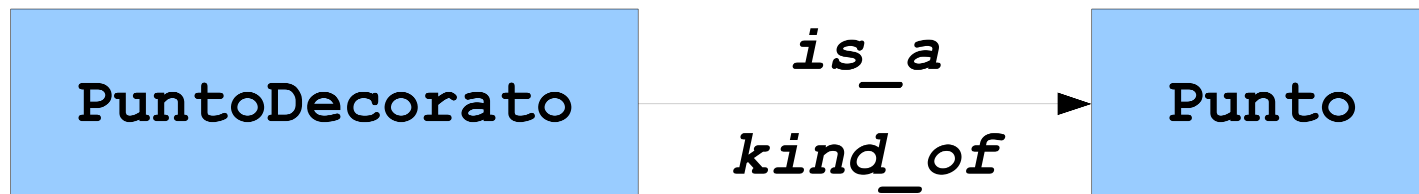
EXE

Ereditarietà

- Abbiamo creato un insieme di classi di oggetti geometrici
- Possiamo usarle per risolvere problemi di geometria
- Oppure possiamo **anche** usarle per disegnare gli oggetti in un pannello grafico
- In questo secondo caso, sarebbe utile avere altre proprietà in più
  - Colore del bordo
  - Colore dell'interno
  - Spessore punto/linea

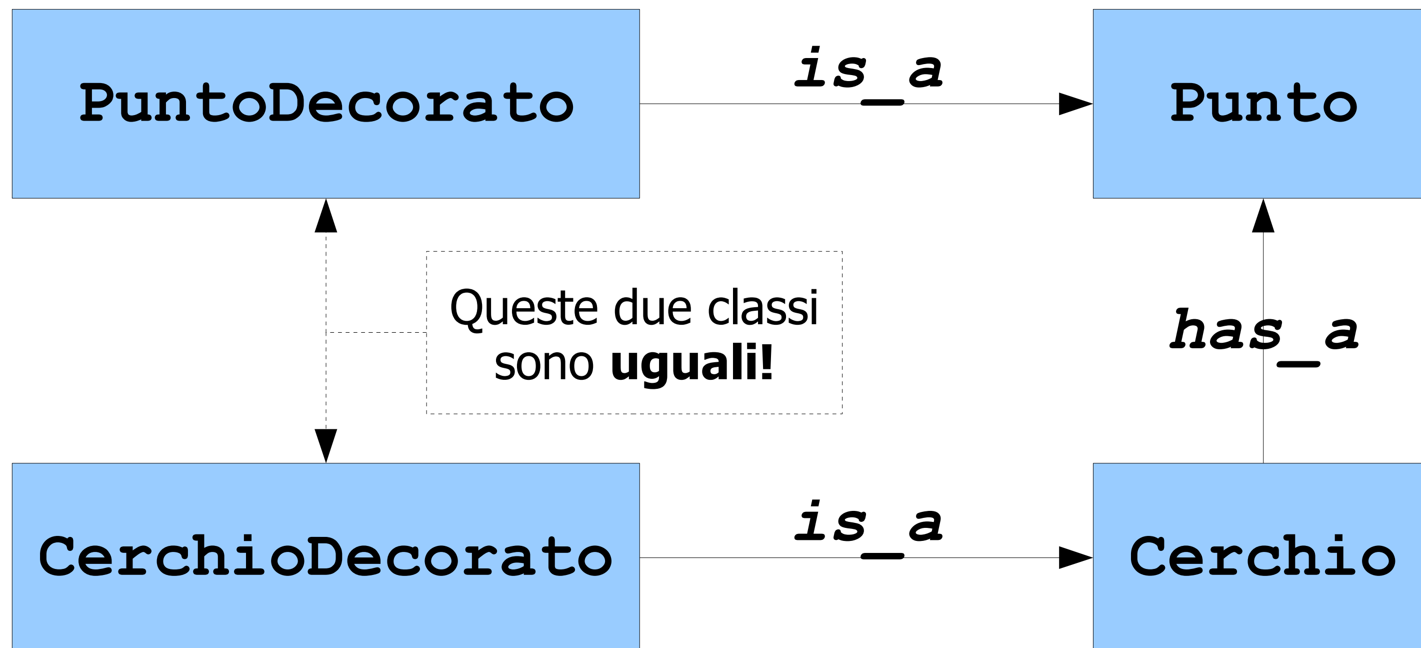
Ereditarietà

- … modifichiamo la classe Punto, Circle … ??
- NO! Non è "saggio", i colori ci servono solo nel caso in cui dobbiamo disegnare l'oggetto
- Ci serve qualcosa che crei una nuova classe:
  - Identica a quella vecchia (Punto)
  - Ma con "qualcosa in più"
- Creiamo una classe "PuntoDecorato" che **eredita** tutte le caratteristiche di "Punto"



Ereditarietà

- Lo stesso principio che abbiamo usato per derivare la classe PuntoDecorato, possiamo usarlo per derivare la classe CerchioDecorato, tuttavia....



Ereditarietà Multipla

Punto
- int x,y
- setXY
- Move
- getX
- getY

DecoratedObject
- int thickness
- int color
- ....

Cerchio
- Punto center
- int radius
- setRadius
- setPosition
- circumference
- area

PuntoDecorato

CerchioDecorato

Ereditarietà Multipla: Sintassi

```
class DecoratedObject {
...
};

class Punto {
...
};

class PuntoDecorato : public Punto,
                      public DecoratedObject {
public:
       PuntoDecorato();
       PuntoDecorato(int initial_x, int initial_y);
};

PuntoDecorato::PuntoDecorato()
                       : Punto(),
                         DecoratedObject()
{
   ...
}
```

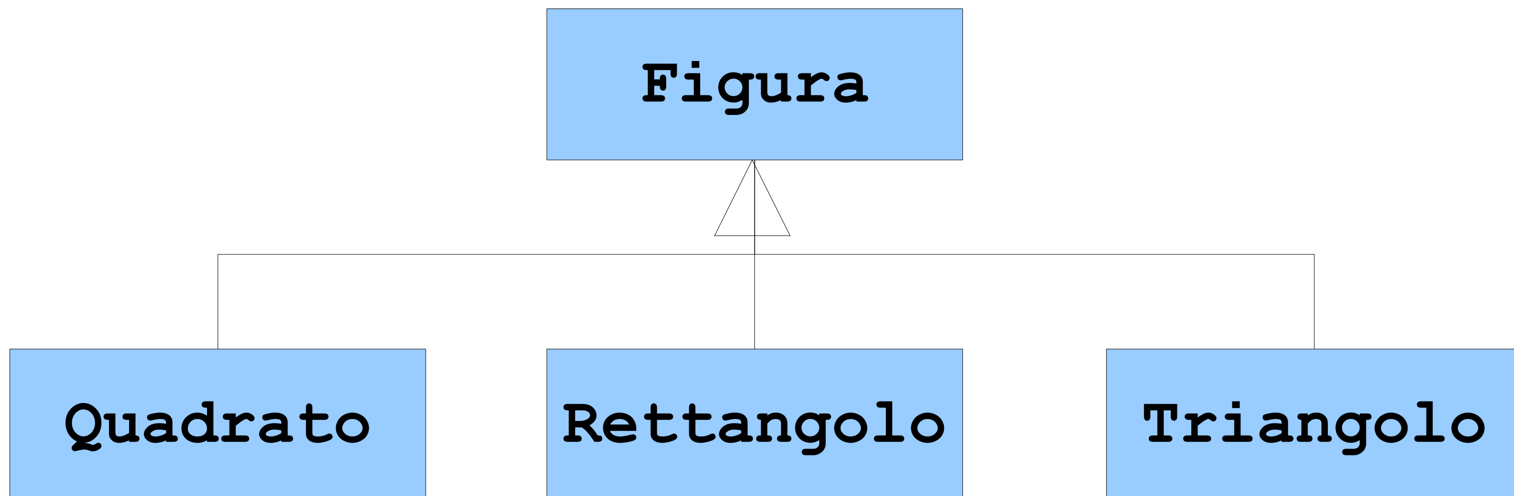
Metodi virtuali e “virtual inheritance”

- Supponiamo di voler progettare una libreria di classi geometriche
- Pensiamo a Quadrato, Rettangolo, Triangolo, Cerchio, …
- Ci interessa rappresentare le proprietà di ogni oggetto ma anche calcolare il perimetro e l'area
- Infine vogliamo che il nostro programma abbia una funzione che riceve una figura geometrica **qualsiasi** e ne stampa il perimetro!

```
void StampaPerimetro(???? figura_geometrica);
```

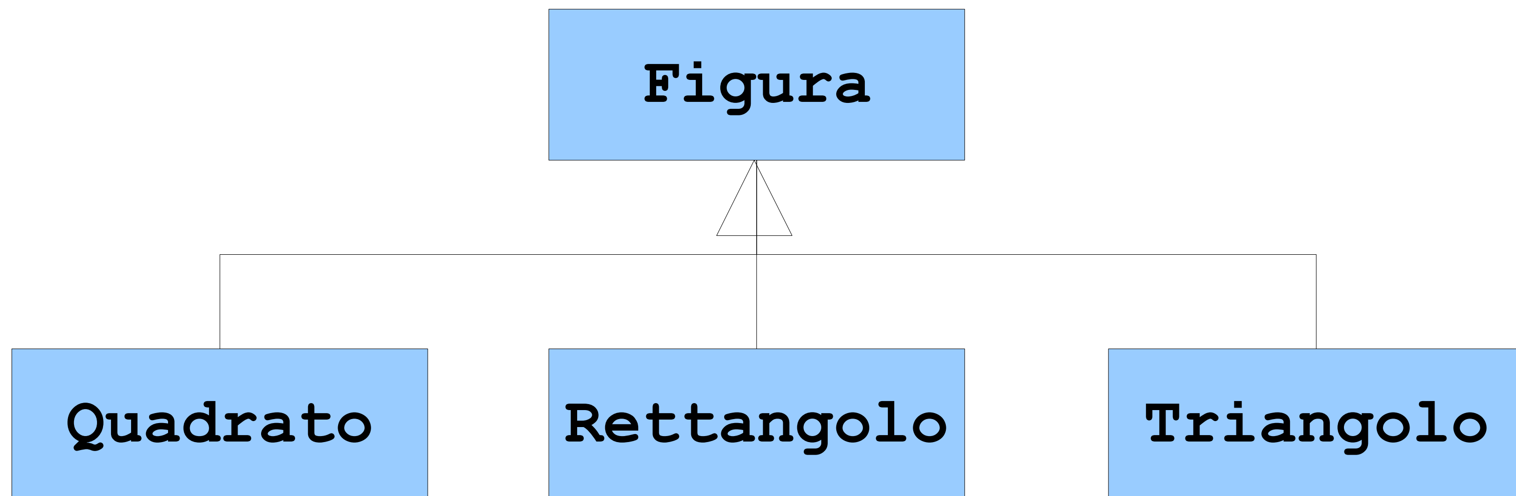
Ereditarietà e Generalizzazione

- Pensiamo a qualcosa che possa “generalizzare” le nostre figure geometriche...
- Per esempio, una classe Figura da cui discendono Quadrato, Rettangolo, …



Ereditarietà e Generalizzazione

- Il **concetto** di perimetro è qualcosa che appartiene alla classe Figura
- Tuttavia il **calcolo** del perimetro è qualcosa che riguarda le singole classi
- Ci serve qualcosa che ci permetta di
  - **Definire** il perimetro in Figura
  - **Implementarlo** nelle singole classi



Ereditarietà e metodi virtuali

```
class Figura {
public:
  virtual float perimetro() { };
};

class Quadrato : public Figura {
public:
  float perimetro();
};

void Quadrato::perimetro()
{
  …
}

StampaPerimetro(Figura &f) { .... }

...
  Quadrato q(10);
  StampaPerimetro(q);
```

Classi Astratte

- Cosa fa il metodo “perimetro” di Figura?
  - **NULLA**!
- Cosa rappresenta in realtà la classe Figura?
  - E' una figura geometrica **indefinita**
  - Diventa **definita** solo nel momento in cui viene ereditata da una classe specifica
- Avrebbe senso **creare** un oggetto di classe Figura?
  - NO!
- Una classe “non instanziabile” (di cui non ha senso creare un oggetto) è detta **astratta** e contiene metodi **vuoti**

Classi Astratte

```
class Figura {
public:
  virtual float perimetro() = 0;
};

class Quadrato : public Figura {
public:
  float perimetro();
};

void Quadrato::perimetro()
{
  …
}

StampaPerimetro(Figura &f) { .... }

...
  Quadrato q(10);
  StampaPerimetro(q);
```