Polimorfismo

Introduzione al polimorfismo

- Il polimorfismo è la possibilità di utilizzare una unica interfaccia per più metodi

Polimorfismo

- Il polimorfismo al momento della compilazione si ottiene con con l'overloading
- Il polimorfismo al momento dell'esecuzione si ottiene con l'ereditarietà e le funzioni virtuali

Esempio di uso

- classe base Forma
- classe derivata Punto
- classe derivata Cerchio
- classe derivata Cilindro

- Si vuole poter invocare una unica funzione **`disegna`** per tutte le classi derivate
- In questo modo un vettore che contiene puntatori a oggetti di diverse classi può chiamare lo stesso metodo **`(*ptr).disegna`** su ogni elemento

Metodo alternativo

- Un modo alternativo per selezionare una azione dato un oggetto di tipo diverso è utilizzare la commutazione (switch)
- Svantaggi:
  - si deve prevedere esplicitamente il test per ogni tipo
  - si devono testare tutti i casi possibili
  - quando si aggiunge un tipo nuovo vanno modificati gli switch in tutto il programma
- La programmazione polimorfica elimina questi svantaggi e dà al programma un aspetto semplificato (meno diramazioni e più codice sequenziale)

Utilità

- Si crea una gerarchia di classi
- Dal caso più generale a quello più specifico
- Tutte le funzionalità e i metodi comuni di interfacciamento sono definiti all'interno di una classe base
- Quando i metodi possono essere implementati solo dalle classi derivate si utilizzano le funzioni virtuali per specificare l'interfaccia che deve essere garantita

Funzioni virtuali

- I metodi di una classe base possono essere dichiarati **`virtual`**

```
virtual retType funcName(parType);
```

- Una classe che eredita una funzione virtual può ridefinirla
- Ereditare una funzione virtual è diverso dal *overriding* di funzione
- La differenza si nota quando si utilizzano i puntatori alla classe base per riferirsi agli oggetti delle classi derivate

Esempio non-virtual

```
class Base{
public:
   Base(int i=0){b=i;}
   void print(){cout<<“Base:”<<b;}
private:
   int b;
};
class Deriv:public Base{
public:
   Deriv(int i=0, int z=0):Base(i){d=z;}
   void print(){cout<<“Deriv:”<<d;}
private:
   int d;
};

void main(){
   Deriv dObj(10,20);
   Base* objPtr=&dObj;
   objPtr->print();//Stampa: Base:10
}
```

Esempio virtual

```
class Base{
public:
   Base(int i=0){b=i;}
   virtual void print(){cout<<“Base:”<<b;}
private:
   int b;
};
class Deriv:public Base{
public:
   Deriv(int i=0, int z=0):Base(i){d=z;}
   void print(){cout<<“Deriv:”<<d;}
private:
   int d;
};

void main(){
   Deriv dObj(10,20);
   Base* objPtr=&dObj;
   objPtr->print(); //Stampa: Deriv:20
}
```

Funzioni virtuali

- In pratica accedendo ad un oggetto di una classe derivata tramite un puntatore di tipo classe base
  - nel caso ordinario si accede ai membri della classe base
  - nel caso virtual si accede ai membri della classe derivata

Funzioni virtuali pure

- Talvolta non è possibile definire un comportamento significativo per una funzione in una classe base
- Ex: la classe base Forma può avere un metodo Stampa ma questo è definibile con precisione solo dalle classi derivate Cerchio e Cilindro che specificano i propri attributi

Funzioni virtuali pure

- Se si vuole specificare che le classi che erditano devono *necessariamente* definire una funzione

  allora si rende tale funzione una

  funzione virtuale pura

- Sintassi:

```
virtual retType FuncName(argType)=0;
```

Classi astratte

- Una classe che contiene una o più funzioni astratte pure è una *classe astratta*

- Non è possibile istanziare alcun oggetto di una classe astratta
- Infatti esiste almeno un metodo che non è possibile definire!!

Classi astratte

- Una gerarchia non deve necessariamente contenere classi astratte
- Tuttavia il livello più alto (o i primi livelli) generalmente è realizzato come classe astratta
- Si specifica così l'interfaccia necessaria per tutte le classi derivate

Distruttore da un puntatore a classe base

- Cosa accade quando si invoca il distruttore tramite un puntatore ad una classe base?
- Se si allocano dinamicamente oggetti con il `new` e si deallocano tramite l'operatore `delete`...
- ...se si sta usando un puntatore alla classe base per riferirsi ad un oggetto di una classe derivata...
- ..allora viene chiamato il distruttore della classe base e non quello della classe derivata

- per chiamare correttamente il distruttore della classe derivata si deve dichiarare il distruttore come **`virtual`**

Distruttori virtuali

- Per classi con funzioni virtuali si consiglia di creare sempre distruttori virtuali anche se non strettamente necessari.
- In questo modo le classi derivate invocheranno i distruttori in modo appropriato

- **NOTA:** i costruttori non possono essere virtuali

Esempio

```
// Definition of abstract base class Shape
#ifndef SHAPE_H
#define SHAPE_H

class Shape {
public:
   virtual double area() const { return 0.0; }
   virtual double volume() const { return 0.0; }

   // pure virtual functions overridden in derived classes
   virtual void printShapeName() const = 0;
   virtual void print() const = 0;
};

#endif
```

```
// Definition of class Point
#ifndef POINT1_H
#define POINT1_H
#include <iostream>
#include "shape.h"

class Point : public Shape {
public:
   Point( int = 0, int = 0 );
   void setPoint( int, int );
   int getX() const { return x; }
   int getY() const { return y; }
   virtual void printShapeName() const {cout << "Point: "; }
   virtual void print() const;
private:
   int x, y;
};

#endif
```

```
// Member function definitions for class Point
#include "point1.h"

Point::Point( int a, int b ) { setPoint( a, b ); }

void Point::setPoint( int a, int b )
{
   x = a;
   y = b;
}

void Point::print() const
   { cout << '[' << x << ", " << y << ']'; }
```

```
// Definition of class Circle
#ifndef CIRCLE1_H
#define CIRCLE1_H
#include "point1.h"

class Circle : public Point {
public:
   // default constructor
   Circle( double r = 0.0, int x = 0, int y = 0 );
   void setRadius( double );
   double getRadius() const;
   virtual double area() const;
   virtual void printShapeName() const{ cout << "Circle: ";
}
   virtual void print() const;
private:
   double radius;   // radius of Circle
};

#endif
```

```
// Member function definitions for class Circle
#include <iostream>
using std::cout;
#include "circle1.h”

Circle::Circle( double r, int a, int b ): Point( a, b )
{ setRadius( r ); }

void Circle::setRadius( double r ){ radius = r > 0 ? r : 0; }

double Circle::getRadius() const { return radius; }

double Circle::area()const{return 3.14159 * radius * radius;}

void Circle::print() const{
   Point::print();
   cout << "; Radius = " << radius;
}
```

```
// Definition of class Cylinder
#ifndef CYLINDR1_H
#define CYLINDR1_H
#include "circle1.h"

class Cylinder : public Circle {
public:
   Cylinder(double h=0.0,double r=0.0,int x=0, int y=0 );
   void setHeight( double );
   double getHeight();
   virtual double area() const;
   virtual double volume() const;
   virtual void printShapeName() const {cout << "Cylinder: ";}
   virtual void print() const;
private:
   double height;};

#endif
```

```
// Member and friend function definitions for class Cylinder
#include <iostream>
using std::cout;
#include "cylindr1.h"
Cylinder::Cylinder( double h, double r, int x, int y )
   :Circle(r,x,y) { setHeight( h ); }
void Cylinder::setHeight( double h ){ height = h > 0 ? h : 0; }
double Cylinder::getHeight() { return height; }
double Cylinder::area() const{
   return 2 * Circle::area() +
          2 * 3.14159 * getRadius() * height;}
double Cylinder::volume() const { return Circle::area() *
   height; }
void Cylinder::print() const{
    Circle::print();
    cout << "; Height = " << height;
}
```

```
// Driver for shape, point, circle, cylinder hierarchy
#include <iostream>
#include <iomanip>

#include "shape.h"
#include "point1.h"
#include "circle1.h"
#include "cylindr1.h"

void virtualViaPointer( const Shape * );
void virtualViaReference( const Shape & );

int main()
{
   cout << setiosflags( ios::fixed | ios::showpoint )
        << setprecision( 2 );

   Point point( 7, 11 );                    // create a Point
   Circle circle( 3.5, 22, 8 );             // create a Circle
   Cylinder cylinder( 10, 3.3, 10, 10 );  // create a Cylinder
```

```
point.printShapeName();    // static binding
 point.print();             // static binding
 cout << '\n';

 circle.printShapeName();   // static binding
 circle.print();            // static binding
 cout << '\n';

 cylinder.printShapeName(); // static binding
 cylinder.print();          // static binding
 cout << "\n\n";

 Shape *arrayOfShapes[ 3 ];  // array of base-class pointers

 // aim arrayOfShapes[0] at derived-class Point object
 arrayOfShapes[ 0 ] = &point;

 // aim arrayOfShapes[1] at derived-class Circle object
 arrayOfShapes[ 1 ] = &circle;

 // aim arrayOfShapes[2] at derived-class Cylinder object
 arrayOfShapes[ 2 ] = &cylinder;
```

```
 // Loop through arrayOfShapes and call virtualViaPointer
    // to print the shape name, attributes, area, and volume
    // of each object using dynamic binding.
    cout << "Virtual function calls made off "
         << "base-class pointers\n";

    for ( int i = 0; i < 3; i++ )
       virtualViaPointer( arrayOfShapes[ i ] );

    // Loop through arrayOfShapes and call virtualViaReference
    // to print the shape name, attributes, area, and volume
    // of each object using dynamic binding.
    cout << "Virtual function calls made off "
         << "base-class references\n";

    for ( int j = 0; j < 3; j++ )
       virtualViaReference( *arrayOfShapes[ j ] );

    return 0;
}
```

```
// Make virtual function calls off a base-class pointer
// using dynamic binding.
void virtualViaPointer( const Shape *baseClassPtr )
{
   baseClassPtr->printShapeName();
   baseClassPtr->print();
   cout << "\nArea = " << baseClassPtr->area()
        << "\nVolume = " << baseClassPtr->volume() << "\n\n";
}

// Make virtual function calls off a base-class reference
// using dynamic binding.
void virtualViaReference( const Shape &baseClassRef )
{
   baseClassRef.printShapeName();
   baseClassRef.print();
   cout << "\nArea = " << baseClassRef.area()
        << "\nVolume = " << baseClassRef.volume() << "\n\n";
}
```