Nuevas Tecnologías de la Programación

Algunas cuestiones sobre C++

Andrés Cano Utrera
Dpto. Ciencias de la Computación e I.A
Universidad de Granada

Noviembre de 2008

Índice

1. Funciones virtuales y polimorfismo

1.1. Campos type y sentencias switch

- Una forma de tratar una colección de objetos de diferentes tipos es añadiendo un campo `type` a los objetos y usar una sentencia `switch` basada en este campo.
- Esta forma de construir programas presenta algunos problemas:
  - El programador puede olvidar comprobar todos los posibles casos en `switch`.
  - Si se añade un nuevo tipo de objeto, el programador puede olvidar insertar los nuevos casos en todas las sentencias `switch`.
  - Cada vez que añadimos o borramos un tipo, tenemos que modificar todas las sentencias `switch`
- Las funciones **virtuales** permiten eliminar la necesidad de la lógica con sentencias `switch`.

1.2. Funciones virtuales

- Una función **virtual** se declara precediendo su prototipo con la palabra **virtual** en la *clase base*.
- Por ejemplo, en una clase **Figura** podríamos definir como virtual la función `draw()`, que sería sobreescrita en las subclases **Circulo**, **Triangulo**,etc

```
virtual void draw() const;
```

- Una vez que se declara una función virtual, ésta **permanece virtual** en todas las clases descendientes, incluso si en ellas no se declaran como virtual al sobreescribirla.
- Si la clase derivada no sobreescribe la función virtual, entonces hereda la de la clase base.

- **Poliformismo en tiempo de ejecución o ligadura dinámica**: Si hacemos que un puntero (`shapePtr`) o referencia a la clase base, apunte a un objeto de la clase derivada, y llamamos a la función `draw` (función virtual) con este puntero (`shapePtr->draw()`) o referencia, el programa elegirá automáticamente en tiempo de ejecución la función `draw()` de la subclase del objeto al que apunte el puntero o referencia.
- Si la función virtual es llamada referenciando un objeto específico usando el operador *punto* (`objetoTriangulo.draw()`), la referencia se resuelve en tiempo de ejecución (*ligadura estática*) y se llama a la versión de la clase a la que pertenezca el objeto (`objetoTriangulo`)

1.3. Clases base abstractas

- Las **clases abstractas** son clases de las que no podemos crear ningún objeto.
- Se usan como clases base en una jerarquía de clases, por lo que se les suele llamar *clases base abstractas*.
- Una clase es abstracta si contiene al menos una función **virtual pura**.
- Una función virtual pura es la que contiene un inicializador `=0` en su declaración.

  ```
  virtual double draw() const = 0;
  ```

- Si una clase se deriva de una clase abstracta que contiene una función virtual pura, y no da una definición para ésta, entonces la clase sigue siendo virtual pura en la subclase, y por tanto también abstracta.
- Si que es posible declarar punteros y referencias de una clase abstracta.
- Tales punteros o referencias pueden usarse entonces para realizar manipulaciones polimórficas de objetos de clases derivadas cuando éstos son instanciados con clases derivadas concretas.
- El polimorfismo y las funciones virtuales permiten definir *algoritmos genéricos*.
- El polimorfismo facilita la *extensibilidad* del software.

1.4. Destructores virtuales

- Si un objeto es destruido explícitamente (con un destructor no virtual) aplicando el operador **delete** usando un puntero a su clase base, se llamará la función destructor de la clase base con tal objeto: **Mal funcionamiento**.
- Esto ocurre independientemente del tipo del objeto al que apunte el puntero.
- La solución es declarar el destructor como **virtual** en la clase base.
- Esto hace que todos los destructores de las clases derivadas sean virtuales.
- Por tanto, si una clase tiene funciones virtuales, es conveniente dar un destructor **virtual**.
- Los constructores no pueden declararse como virtuales.

1.5. Ejemplo

Jerarquía de figuras: Fig10.2/shape.h

```
1  // Fig. 10.2: shape.h
2  // Definition of abstract base class Shape
3  #ifndef SHAPE_H
4  #define SHAPE_H
5  class Shape {
6  public:
7     virtual double area() const { return 0.0; }
8     virtual double volume() const { return 0.0; }
9
10    // pure virtual functions overridden in derived classes
11    virtual void printShapeName() const = 0;
12    virtual void print() const = 0;
13 };
14 #endif
```

Jerarquía de figuras: Fig10.2/point1.h

```
1   // Fig. 10.2: point1.h
2  // Definition of class Point
```

```
3  #ifndef POINT1_H
4  #define POINT1_H
5  #include <iostream>
6  using std::cout;
7  #include "shape.h"
8
9  class Point : public Shape {
10 public:
11    Point( int = 0, int = 0 );  // default constructor
12    void setPoint( int, int );
13    int getX() const { return x; }
14    int getY() const { return y; }
15    virtual void printShapeName() const { cout << "Point: "; }
16    virtual void print() const;
17 private:
18    int x, y;   // x and y coordinates of Point
19 };
20 #endif
```

Jerarquía de figuras: Fig10.2/point1.cpp

```
1  / Fig. 10.2: point1.cpp
2  // Member function definitions for class Point
3  #include "point1.h"
4
5  Point::Point( int a, int b ) { setPoint( a, b ); }
6
7  void Point::setPoint( int a, int b )
8  {
9     x = a;
10    y = b;
11 }
12
13 void Point::print() const
14    { cout << '[' << x << ", " << y << ']'; }
```

Jerarquía de figuras: Fig10.2/circle1.h

```
1  // Fig. 10.2: circle1.h
2  // Definition of class Circle
3  #ifndef CIRCLE1_H
4  #define CIRCLE1_H
5  #include "point1.h"
6
7  class Circle : public Point {
8  public:
9     Circle( double r = 0.0, int x = 0, int y = 0 );// default constructor
10    void setRadius( double );
11    double getRadius() const;
12    virtual double area() const;
13    virtual void printShapeName() const { cout << "Circle: "; }
14    virtual void print() const;
15 private:
16    double radius;   // radius of Circle
17 };
18 #endif
```

Jerarquía de figuras: Fig10.2/circle1.cpp

```
1  // Fig. 10.2: circle1.cpp
2  // Member function definitions for class Circle
3  #include <iostream>
4  using std::cout;
5  #include "circle1.h"
6
7  Circle::Circle( double r, int a, int b )
8     : Point( a, b )  // call base-class constructor
9  { setRadius( r ); }
10
11 void Circle::setRadius( double r ) { radius = r > 0 ? r : 0; }
12 double Circle::getRadius() const { return radius; }
13 double Circle::area() const
14    { return 3.14159 * radius * radius; }
15 void Circle::print() const
16 {
17    Point::print();
18    cout << "; Radius = " << radius;
19 }
```

Jerarquía de figuras: Fig10.2/cylindr1.h

```
1  // Fig. 10.2: cylindr1.h
2  // Definition of class Cylinder
3  #ifndef CYLINDR1_H
4  #define CYLINDR1_H
5  #include "circle1.h"
6
7  class Cylinder : public Circle {
8  public:
9     // default constructor
10    Cylinder( double h = 0.0, double r = 0.0,
11              int x = 0, int y = 0 );
12
13    void setHeight( double );
14    double getHeight();
15    virtual double area() const;
16    virtual double volume() const;
17    virtual void printShapeName() const {cout << "Cylinder: ";}
18    virtual void print() const;
19 private:
20    double height;   // height of Cylinder
21 };
22 #endif
```

Jerarquía de figuras: Fig10.2/cylindr1.cpp

```
1  Cylinder::Cylinder( double h, double r, int x, int y )
2     : Circle( r, x, y )  // call base-class constructor
3  { setHeight( h ); }
4
5  void Cylinder::setHeight( double h )
6     { height = h > 0 ? h : 0; }
7
8  double Cylinder::getHeight() { return height; }
9
10 double Cylinder::area() const
11 {
12    // surface area of Cylinder
13    return 2 * Circle::area() +
14           2 * 3.14159 * getRadius() * height;
15 }
16
17 double Cylinder::volume() const
18    { return Circle::area() * height; }
19
20 void Cylinder::print() const
21 {
22    Circle::print();
23    cout << "; Height = " << height;
24 }
```

Jerarquía de figuras: Fig10.2/fig10-02.cpp

```
1  // Fig. 10.2: fig10_02.cpp
2  #include <iostream>
3  using std::cout;
4  using std::endl;
5  #include <iomanip>
6  using std::ios;
7  using std::setiosflags;
8  using std::setprecision;
9  #include "shape.h"
10 #include "point1.h"
11 #include "circle1.h"
12 #include "cylindr1.h"
13 void virtualViaPointer( const Shape * );
14 void virtualViaReference( const Shape & );
15 int main()
16 {
17    cout << setiosflags( ios::fixed | ios::showpoint )
18         << setprecision( 2 );
19    Point point( 7, 11 );                  // create a Point
20    Circle circle( 3.5, 22, 8 );           // create a Circle
21    Cylinder cylinder( 10, 3.3, 10, 10 );  // create a Cylinder
22
23    point.printShapeName();    // static binding
24    point.print();             // static binding
25    cout << '\n';
26
27    circle.printShapeName();   // static binding
28    circle.print();            // static binding
29    cout << '\n';
30
31    cylinder.printShapeName(); // static binding
32    cylinder.print();          // static binding
33    cout << "\n\n";
34
35    Shape *arrayOfShapes[ 3 ];  // array of base-class pointers
36
37    // aim arrayOfShapes[0] at derived-class Point object
38    arrayOfShapes[ 0 ] = &point;
39
40    // aim arrayOfShapes[1] at derived-class Circle object
41    arrayOfShapes[ 1 ] = &circle;
```

```
42
43    // aim arrayOfShapes[2] at derived-class Cylinder object
44    arrayOfShapes[ 2 ] = &cylinder;
45
46    // Loop through arrayOfShapes and call virtualViaPointer
47    // to print the shape name, attributes, area, and volume
48    // of each object using dynamic binding.
49    cout << "Virtual function calls made off "
50         << "base-class pointers\n";
51
52    for ( int i = 0; i < 3; i++ )
53       virtualViaPointer( arrayOfShapes[ i ] );
54
55    // Loop through arrayOfShapes and call virtualViaReference
56    // to print the shape name, attributes, area, and volume
57    // of each object using dynamic binding.
58    cout << "Virtual function calls made off "
59         << "base-class references\n";
60    for ( int j = 0; j < 3; j++ )
61       virtualViaReference( *arrayOfShapes[ j ] );
62    return 0;
63 }
64
65 // Make virtual function calls off a base-class pointer
66 // using dynamic binding.
67 void virtualViaPointer( const Shape *baseClassPtr )
68 {
69    baseClassPtr->printShapeName();
70    baseClassPtr->print();
71    cout << "\nArea = " << baseClassPtr->area()
72         << "\nVolume = " << baseClassPtr->volume() << "\n\n";
73 }
74
75 // Make virtual function calls off a base-class reference
76 // using dynamic binding.
77 void virtualViaReference( const Shape &baseClassRef )
78 {
79    baseClassRef.printShapeName();
80    baseClassRef.print();
81    cout << "\nArea = " << baseClassRef.area()
82         << "\nVolume = " << baseClassRef.volume() << "\n\n";
83 }
```