Escola de Engenharia
Universidade do Minho

Complementos de Programação de Computadores – Aula 5a

Funções Virtuais e Polimorfismo

Mestrado Integrado em Electrónica Industrial e Computadores

Luís Paulo Reis

lpreis@dsi.uminho.pt

Professor Associado do Departamento de Sistemas de Informação, Escola de Engenharia, Universidade do Minho, Portugal

(Slides Baseados em P.Deitel e H.Deitel 2010)

Escola de Engenharia
Universidade do Minho

Outline

- 20.1 Introduction
- 20.2 Type Fields and switch Statements
- 20.3 Virtual Functions
- 20.4 Abstract Base Classes and Concrete Classes
- 20.5 Polymorphism
- 20.6 New Classes and Dynamic Binding
- 20.7 Virtual Destructors
- 20.8 Case Study: Inheriting Interface and Implementation
- 20.9 Polymorphism, virtual Functions and Dynamic Binding

Escola de Engenharia
Universidade do Minho

Objectives

- To understand the notion of polymorphism
- To understand how to define and use `virtual` functions to effect polymorphism
- To understand the distinction between abstract classes and concrete classes
- To learn how to define pure `virtual` functions to create abstract classes
- To appreciate how polymorphism makes systems extensible and maintainable
- To understand how C++ implements `virtual` functions and dynamic binding “under the hood”

Escola de Engenharia
Universidade do Minho

20.1 Introduction

- **`virtual` functions and polymorphism**
  - Design and implement systems that are more easily extensible
  - Programs written to generically process objects of all existing classes in a hierarchy

Escola de Engenharia
Universidade do Minho

20.2 Type Fields and `switch` Statements

- **`switch` statement**
  - Take an action on a object based on its type
  - A `switch` structure could determine which print function to call based on which type in a hierarchy of shapes

- **Problems with `switch`**
  - Programmer may forget to test all possible cases in a `switch`
    - Tracking this down can be time consuming and error prone
    - Virtual functions and polymorphic programming can eliminate the need for `switch`

Escola de Engenharia
Universidade do Minho

20.3 Virtual Functions

- **`virtual` functions**
  - Used instead of `switch` statements
  - Definition:
    - Keyword `virtual` before function prototype in base class

    ```
    virtual void draw() const;
    ```
  - A base-class pointer to a derived class object will call the correct `draw` function
  - If a derived class does not define a `virtual` function it is inherited from the base class

- **`ShapePtr->Draw();`**
  - Compiler implements dynamic binding
  - Function determined during execution time

- **`ShapeObject.Draw();`**
  - Compiler implements static binding
  - Function determined during compile-time

Escola de Engenharia
Universidade do Minho

20.4 Abstract and Concrete Classes

- **Abstract classes**
  - Sole purpose is to provide a base class for other classes
  - No objects of an abstract base class can be instantiated
    - Too generic to define real objects, i.e. `TwoDimensionalShape`
    - Can have pointers and references
  - Concrete classes - classes that can instantiate objects
    - Provide specifics to make real objects, i.e. `Square`, `Circle`
- **Making abstract classes**
  - Define one or more `virtual` functions as "pure" by initializing the function to zero

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

    - Pure `virtual` function

Escola de Engenharia
Universidade do Minho

20.5 Polymorphism

- **Polymorphism:**
  - Ability for objects of different classes to respond differently to the same function call
  - Base-class pointer (or reference) calls a `virtual` function
    - C++ chooses the correct overridden function in object
  - Suppose `print` not a `virtual` function

```
Employee e, *ePtr = &e;
HourlyWorker h, *hPtr = &h;
ePtr->print();   // call base-class print function
hPtr->print();   // call derived-class print function
ePtr=&h;         // allowable implicit conversion
ePtr->print();   // still calls base-class print
```

Escola de Engenharia
Universidade do Minho

20.6 New Classes and Dynamic Binding

- **Dynamic binding (late binding)**
  - Object's type not needed when compiling `virtual` functions
  - Accommodate new classes that have been added after compilation
  - Important for ISV's (Independent Software Vendors) who do not wish to reveal source code to their customers

Escola de Engenharia
Universidade do Minho

20.7 Virtual Destructors

- **Problem:**
  - If base-class pointer to a derived object is `deleted`, the base-class destructor will act on the object
- **Solution:**
  - Define a `virtual` base-class destructor
  - Now, the appropriate destructor will be called

20.8 Case Study: Inheriting Interface and Implementation

- **Re-examine the `Point`, `Circle`, `Cylinder` hierarchy**
  - Use the abstract base class `Shape` to head the hierarchy


```
1  // Fig. 20.1: shape.h
2  // Definition of abstract base class Shape
3  #ifndef SHAPE_H
4  #define SHAPE_H
5
6  class Shape {
7  public:
8       virtual double area() const { return 0.0; }
9       virtual double volume() const { return 0.0; }
10
11 // pure virtual functions overridden in derived classes
12      virtual void printShapeName() const = 0;
13      virtual void print() const = 0;
14 }; // end class Shape
15
16 #endif
17 // Fig. 20.1: point1.h
18 // Definition of class Point
19 #ifndef POINT1_H
20 #define POINT1_H
21
22 #include <iostream>
23
24 using std::cout;
```

1. `Shape` **Definition (abstract base** `class`**)**

---

1. `Point` **Definition (derived** `class`**)**

```
26   #include "shape.h"
27
28   class Point : public Shape {
29   public:
30      Point( int = 0, int = 0 );  // default constructor
31      void setPoint( int, int );
32      int getX() const { return x; }
33      int getY() const { return y; }
34      virtual void printShapeName() const { cout << "Point: ";
     }
35      virtual void print() const;
36   private:
37      int x, y;   // x and y coordinates of Point
38   }; // end class Point
39
40   #endif
41   // Fig. 20.1: point1.cpp
42   // Member function definitions for class Point
43  #include "point1.h"
44
45  Point::Point( int a, int b ) { setPoint( a, b ); }
46
47  void Point::setPoint( int a, int b ) {
48     x = a;  y = b;
49  } // end function setPoint
```

1. `Point` Definition (derived `class`)

1.1 Function Definitions


```
53   void Point::print() const
54      { cout << '[' << x << ", " << y << ']'; }
55   // Fig. 20.1: circle1.h
56   // Definition of class Circle
57   #ifndef CIRCLE1_H
58   #define CIRCLE1_H
59   #include "point1.h"
60
61   class Circle : public Point {
62   public:
63      // default constructor
64      Circle( double r = 0.0, int x = 0, int y = 0 );
65
66      void setRadius( double );
67      double getRadius() const;
68      virtual double area() const;
69      virtual void printShapeName() const { cout << "Circle: ";}
70      virtual void print() const;
71   private:
72      double radius;   // radius of Circle
73   }; // end class Circle
74
75   #endif
```

1. Circle Definition (derived class)

```
76  // Fig. 20.1: circle1.cpp
77  // Member function definitions for class Circle
78  #include <iostream>
79
80  using std::cout;
81
82  #include "circle1.h"
83
84  Circle::Circle( double r, int a, int b )
85     : Point( a, b )  // call base-class constructor
86  { setRadius( r ); }
87
88  void Circle::setRadius( double r ) { radius = r > 0 ? r : 0;
}
89
90  double Circle::getRadius() const { return radius; }
91
92  double Circle::area() const
93     { return 3.14159 * radius * radius; }
94
95  void Circle::print() const
96  {
97     Point::print();
98     cout << "; Radius = " << radius;
99  } // end function print
```

1.1 Function Definitions


```
100 // Fig. 20.1: cylindr1.h
101 // Definition of class Cylinder
102 #ifndef CYLINDR1_H
103 #define CYLINDR1_H
104 #include "circle1.h"
105
106 class Cylinder : public Circle {
107 public:
108    // default constructor
109    Cylinder( double h = 0.0, double r = 0.0,
110              int x = 0, int y = 0 );
111
112    void setHeight( double );
113    double getHeight();
114    virtual double area() const;
115    virtual double volume() const;
116    virtual void printShapeName() const { cout << "Cylinder: ";}
117    virtual void print() const;
118 private:
119    double height;   // height of Cylinder
120 }; // end class Cylinder
121
122 #endif
```

1. `Cylinder` **Definition (derived** `class`**)**

```
123 // Fig. 20.1: cylindr1.cpp
124 // Member and friend function definitions for class Cylinder
125 #include <iostream>
126
127 using std::cout;
128
129 #include "cylindr1.h"
130
131 Cylinder::Cylinder( double h, double r, int x, int y )
132    : Circle( r, x, y )  // call base-class constructor
133 { setHeight( h ); }
134
135 void Cylinder::setHeight( double h )
136    { height = h > 0 ? h : 0; }
137
138 double Cylinder::getHeight() { return height; }
139
140 double Cylinder::area() const
141 {
142    // surface area of Cylinder
143    return 2 * Circle::area() +
144           2 * 3.14159 * getRadius() * height;
145 } // end function area
146
```

1.1 Function Definitions


```
147 double Cylinder::volume() const
148    { return Circle::area() * height; }
149
150 void Cylinder::print() const
151 {
152    Circle::print();
153    cout << "; Height = " << height;
154 } // end function print
155 // Fig. 20.1: fig20_01.cpp
156 // Driver for shape, point, circle, cylinder hierarchy
157 #include <iostream>
158
159 using std::cout;
160 using std::endl;
161
162 #include <iomanip>
163
164 using std::ios;
165 using std::setiosflags;
166 using std::setprecision;
167
168 #include "shape.h"
169 #include "point1.h"
170 #include "circle1.h"
171 #include "cylindr1.h"
172
```

Driver

1. Load headers

1.1 Function prototypes

```
173  void virtualViaPointer( const Shape * );
174  void virtualViaReference( const Shape & );
175
176  int main()
177  {
178     cout << setiosflags( ios::fixed | ios::showpoint )
179          << setprecision( 2 );
180
181     Point point( 7, 11 );                  // create a Point
182     Circle circle( 3.5, 22, 8 );           // create a Circle
183     Cylinder cylinder( 10, 3.3, 10, 10 );  // create a Cylinder
184
185     point.printShapeName();    // static binding
186     point.print();             // static binding
187     cout << '\n';
188
189     circle.printShapeName();   // static binding
190     circle.print();            // static binding
191     cout << '\n';
192
193     cylinder.printShapeName(); // static binding
194     cylinder.print();          // static binding
195     cout << "\n\n";
196
197     Shape *arrayOfShapes[ 3 ];  // array of base-class pointers
198
```

1.2 Initialize objects

2. Function calls


```
199     // aim arrayOfShapes[0] at derived-class Point object
200     arrayOfShapes[ 0 ] = &point;
201
202     // aim arrayOfShapes[1] at derived-class Circle object
203     arrayOfShapes[ 1 ] = &circle;
204
205     // aim arrayOfShapes[2] at derived-class Cylinder object
206     arrayOfShapes[ 2 ] = &cylinder;
207
208     // Loop through arrayOfShapes and call virtualViaPointer
209     // to print the shape name, attributes, area, and volume
210     // of each object using dynamic binding.
211     cout << "Virtual function calls made off "
212          << "base-class pointers\n";
213
214     for ( int i = 0; i < 3; i++ )
215        virtualViaPointer( arrayOfShapes[ i ] );
216
217     // Loop through arrayOfShapes and call virtualViaReference
218     // to print the shape name, attributes, area, and volume
219     // of each object using dynamic binding.
220     cout << "Virtual function calls made off "
221          << "base-class references\n";
222
```

2. Function calls

```
223    for ( int j = 0; j < 3; j++ )
224       virtualViaReference( *arrayOfShapes[ j ] );
225
226    return 0;
227 } // end function main
228
229 // Make virtual function calls off a base-class pointer
230 // using dynamic binding.
231 void virtualViaPointer( const Shape *baseClassPtr )
232 {
233    baseClassPtr->printShapeName();
234    baseClassPtr->print();
235    cout << "\nArea = " << baseClassPtr->area()
236         << "\nVolume = " << baseClassPtr->volume() << "\n\n";
237 } // end function virtualViaPointer
238
239 // Make virtual function calls off a base-class reference
240 // using dynamic binding.
241 void virtualViaReference( const Shape &baseClassRef )
242 {
243    baseClassRef.printShapeName();
244    baseClassRef.print();
245    cout << "\nArea = " << baseClassRef.area()
246         << "\nVolume = " << baseClassRef.volume() << "\n\n";
247 } // end function virtualViaReference
```

3. Function Definitions


```
Point: [7, 11]
Circle: [22, 8]; Radius = 3.50
Cylinder: [10, 10]; Radius = 3.30; Height = 10.00

Virtual function calls made off base-class pointers
Point: [7, 11]
Area = 0.00
Volume = 0.00

Circle: [22, 8]; Radius = 3.50
Area = 38.48
Volume = 0.00

Cylinder: [10, 10]; Radius = 3.30; Height = 10.00
Area = 275.77
Volume = 342.12

Virtual function calls made off base-class references
Point: [7, 11]
Area = 0.00
Volume = 0.00

Circle: [22, 8]; Radius = 3.50
Area = 38.48
Volume = 0.00

Cylinder: [10, 10]; Radius = 3.30; Height = 10.00
Area = 275.77
Volume = 342.12
```

Program Output

Escola de Engenharia
Universidade do Minho

20.9 Polymorphism, `virtual` Functions and Dynamic Binding

- **When to use polymorphism**
  - Polymorphism has a lot of overhead
- **`virtual` function table (vtable)**
  - Every class with a `virtual` function has a vtable
  - For every `virtual` function, vtable has a pointer to the proper function
    - If a derived class has the same function as a base class, then the function pointer points to the base-class function

Escola de Engenharia
Universidade do Minho

Complementos de Programação de Computadores – Aula 5a

Funções Virtuais e Polimorfismo

Mestrado Integrado em Electrónica Industrial e Computadores

Luís Paulo Reis

lpreis@dsi.uminho.pt

Professor Associado do Departamento de Sistemas de Informação, Escola de Engenharia, Universidade do Minho, Portugal

(Slides Baseados em P.Deitel e H.Deitel 2010)