Chapter 20 - C++ Virtual Functions and Polymorphism

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 “Under the Hood”
Objectives

• In this chapter, you will learn:
  – To understand the notion of polymorphism.
  – To understand how to define and use \texttt{virtual} functions to effect polymorphism.
  – To understand the distinction between abstract classes and concrete classes.
  – To learn how to define pure \texttt{virtual} functions to create abstract classes.
  – To appreciate how polymorphism makes systems extensible and maintainable.
  – To understand how C++ implements \texttt{virtual} functions and dynamic binding “under the hood.”
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
20.2 Type Fields and switch Statements

- **switch** statement
  - Take an action on an 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
20.3 Virtual Functions

• virtual functions
  – Used instead of switch statements
  – Definition:
    • Keyword virtual before function prototype in base class
      ```cpp
      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
20.3 Virtual Functions

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

- **ShapeObject.Draw();**
  - Compiler implements static binding
  - Function determined during compile-time
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`
20.4 Abstract and Concrete Classes

• 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

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
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
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. Shape Definition  
   (abstract base class) 

1. Point Definition  
   (derived class)
#include "shape.h"

class Point : public Shape {

public:
    Point( int = 0, int = 0 ); // default constructor
    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; // x and y coordinates of Point
}; // end class Point

#endif

// Fig. 20.1: point1.cpp
// 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;
} // end function setPoint
```cpp
void Point::print() const
{ cout << '[' << x << ', ' << y << ']'; }

// Fig. 20.1: circle1.h
// 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
}; // end class Circle

#endif
```
// Fig. 20.1: circle1.cpp
// 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 )    // call base-class constructor
    { 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;  
} // end function print
// Fig. 20.1: cylindr1.h
// Definition of class Cylinder

#ifndef CYLINDR1_H
#define CYLINDR1_H

#include "circle1.h"

class Cylinder : public Circle {
public:
    // default constructor
    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; // height of Cylinder
}; // end class Cylinder

#endif
// Fig. 20.1: cylindr1.cpp
// 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 ) // call base-class constructor
    { setHeight( h ); }

void Cylinder::setHeight( double h )
    { height = h > 0 ? h : 0; }

double Cylinder::getHeight() { return height; }

double Cylinder::area() const
    {
        // surface area of Cylinder
        return 2 * Circle::area() +
               2 * 3.14159 * getRadius() * height;
    } // end function area
double Cylinder::volume() const
    { return Circle::area() * height; }

void Cylinder::print() const
    {
    Circle::print();
    cout << "; Height = " << height;
    } // end function print

// Fig. 20.1: fig20_01.cpp
// Driver for shape, point, circle, cylinder hierarchy
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>
using std::ios;
using std::setiosflags;
using std::setprecision;

#include "shape.h"
#include "point1.h"
#include "circle1.h"
#include "cylindr1.h"
```cpp
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
```
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
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
for ( int j = 0; j < 3; j++ )
    virtualViaReference( *arrayOfShapes[ j ] );

return 0;
} // end function main

// 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 = " << baseClassRef->volume() << "\n\n";
} // end function virtualViaPointer

// 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";
} // end function virtualViaReference
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
20.9 Polymorphism, virtual Functions and Dynamic Binding “Under the Hood”

- 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
  - Detailed explanation in Fig. 20.2