9

Object-Oriented Programming: Inheritance

Objectives

- To be able to create classes by inheriting from existing classes.
- To understand how inheritance promotes software reusability.
- To understand the notions of base classes and derived classes.
- To understand the **`protected`** member-access modifier.
- To understand the use of constructors and destructors in inheritance hierarchies.

Say not you know another entirely, till you have divided an inheritance with him.
Johann Kasper Lavater

This method is to define as the number of a class the class of all classes similar to the given class.
Bertrand Russell

A deck of cards was built like the purest of hierarchies, with every card a master to those below it, a lackey to those above it.
Ely Culbertson

Good as it is to inherit a library, it is better to collect one.
Augustine Birrell

Save base authority from others' books.
William Shakespeare

Outline

9.1 Introduction

In this chapter, we begin our discussion of object-oriented programming (OOP) by introducing one of its main features—*inheritance*. Inheritance is a form of software reusability in which programmers create classes that absorb an existing class's data and behaviors and enhance them with new capabilities. Software reusability saves time during program development. It also encourages the reuse of proven and debugged high-quality software, which increases the likelihood that a system will be implemented effectively.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should *inherit* the members of an existing class. This existing class is called the *base class*, and the new class is referred to as the *derived class.* (Other programming languages, such as Java™, refer to the base class as the *superclass* and the derived class as the *subclass*.) A derived class represents a more specialized group of objects. Typically, a derived class contains behaviors inherited from its base class plus additional behaviors. As we will see, a derived class can also customize behaviors inherited from the base class. A *direct base class* is the base class from which a derived class explicitly inherits. An *indirect base class* is inherited from two or more levels up the *class hierarchy*. In the case of *single inheritance,* a class is derived from one base class. C++ also supports *multiple inheritance*, in which a derived class inherits from multiple (possibly unrelated) base classes. Single inheritance is straightforward—we show several examples that should enable the reader to become proficient quickly. Multiple inheritance can be complex and error prone. We cover multiple inheritance in Chapter 22.

C++ offers three kinds of inheritance—**`public`**, **`protected`** and **`private`**. In this chapter, we concentrate on **`public`** inheritance and briefly explain the other two kinds. In Chapter 17, we show how **`private`** inheritance can be used as an alternative to composition. The third form, **`protected`** inheritance, is rarely used. With **`public`** inheritance, every object of a derived class is also an object of that derived class's base class. However, base-class objects are not objects of their derived classes. For example, all cars are vehicles, but not all vehicles are cars. As we continue our study of object-oriented programming in Chapter 9 and Chapter 10, we take advantage of this relationship to perform some interesting manipulations.

Experience in building software systems indicates that significant portions of code deal with closely related special cases. When programmers are preoccupied with special cases, the details can obscure the "big picture." With object-oriented programming, programmers focus on the commonalities among objects in the system, rather than on the special cases. This process is called *abstraction*.

We distinguish between the *"is-a" relationship* and the *"has-a" relationship*. The "is-a" relationship represents inheritance. In an "is-a" relationship, an object of a derived class also can be treated as an object of its base class—for example, a car *is a* vehicle, so any properties and behaviors of a vechicle are also properties of a car. By contrast, the "has-a" relationship stands for composition. (Composition was discussed in Chapter 7.) In a "has-a" relationship, an object contains one or more objects of other classes as members—for example, a car *has a* steering wheel.

Derived-class member functions might require access to base-class data members and member functions. A derived class can access the non-**`private`** members of its base class. Base-class members that should not be accessible to the member functions of derived classes should be declared **`private`** in the base class. A derived class can effect state changes in **`private`** base-class members, but only through non-**`private`** member functions provided in the base class and inherited into the derived class.

Software Engineering Observation 9.1

Member functions of a derived class cannot directly access **`private`** *members of their class's base class.*

Software Engineering Observation 9.2

If a derived class could access its base class's **`private`** *members, classes that inherit from that derived class could access that data as well. This would propagate access to what should be* **`private`** *data, and the benefits of information hiding would be lost.*

One problem with inheritance is that a derived class can inherit data members and member functions it does not need or should not have. It is the class designer's responsibility to ensure that the capabilities provided by a class are appropriate for future derived classes. Even when a base-class member function is appropriate for a derived class, the derived class often requires that member function to behave in a manner specific to the derived class. In such cases, the base-class member function can be redefined in the derived class with an appropriate implementation.

9.2 Base Classes and Derived Classes

Often, an object of one class "is an" object of another class, as well. For example, in geometry, a rectangle *is a* quadrilateral (as are squares, parallelograms and trapezoids). Thus, in C++, class **`Rectangle`** can be said to *inherit* from class **`Quadrilateral`**. In this context, class **`Quadrilateral`** is a base class, and class **`Rectangle`** is a derived class. A rectangle *is a* specific type of quadrilateral, but it is incorrect to claim that a quadrilateral *is a* rectangle—the quadrilateral could be a parallelogram or some other shape. Figure 9.1 lists several simple examples of base classes and derived classes.

Base class	Derived classes
`Student`	`GraduateStudent` `UndergraduateStudent`

`Shape`	`Circle`
	`Triangle`
	`Rectangle`

`Loan`	`CarLoan`
	`HomeImprovementLoan`
	`MortgageLoan`

`Employee`	`Faculty` `Staff`
`Account`	`CheckingAccount` `SavingsAccount`

Fig. 9.1 Inheritance examples.

Because every derived-class object "is an" object of its base class, and one base class can have many derived classes, the set of objects represented by a base class typically is larger than the set of objects represented by any of its derived classes. For example, the base class **`Vehicle`** represents all vehicles, including cars, trucks, boats, bicycles and so on. By contrast, derived class **`Car`** represents a smaller, more-specific subset of all vehicles.

Inheritance relationships form tree-like hierarchical structures. A base class exists in a hierarchical relationship with its derived classes. Although classes can exist independently, once they are employed in inheritance relationships, they become affiliated with other classes. A class becomes either a base class, supplying data and behaviors to other classes, or a derived class, inheriting its data and behaviors from other classes.

Let us develop a simple inheritance hierarchy. A university community has thousands of members. These members consist of employees, students and alumni. Employees are either faculty members or staff members. Faculty members are either administrators (such as deans and department chairpersons) or teachers. This organizational structure yields the inheritance hierarchy depicted in Fig. 9.2. Note that this inheritance hierarchy could contain many other classes. For example, students can be graduate or undergraduate students. Undergraduate students can be freshmen, sophomores, juniors and seniors. Each arrow in the hierarchy represents an "is-a" relationship. For example, as we follow the arrows in this class hierarchy, we can state "an **`Employee`** *is a* **`CommunityMember`**" and "a **`Teacher`** *is a* **`Faculty`** member." **`CommunityMember`** is the *direct base class* of **`Employee`**, **`Student`** and **`Alumnus`**. In addition, **`CommunityMember`** is an *indirect base class* of all the other classes in the diagram. Starting from the bottom of the diagram, the reader can follow the arrows and apply the *is-a* relationship to the topmost base class. For example, an **`Administrator`** *is a* **`Faculty`** member, *is an* **`Employee`** and *is a* **`Community-Member`**. Note that some administrators also teach classes, so we have used multiple inheritance to form class **`AdministratorTeacher`**.

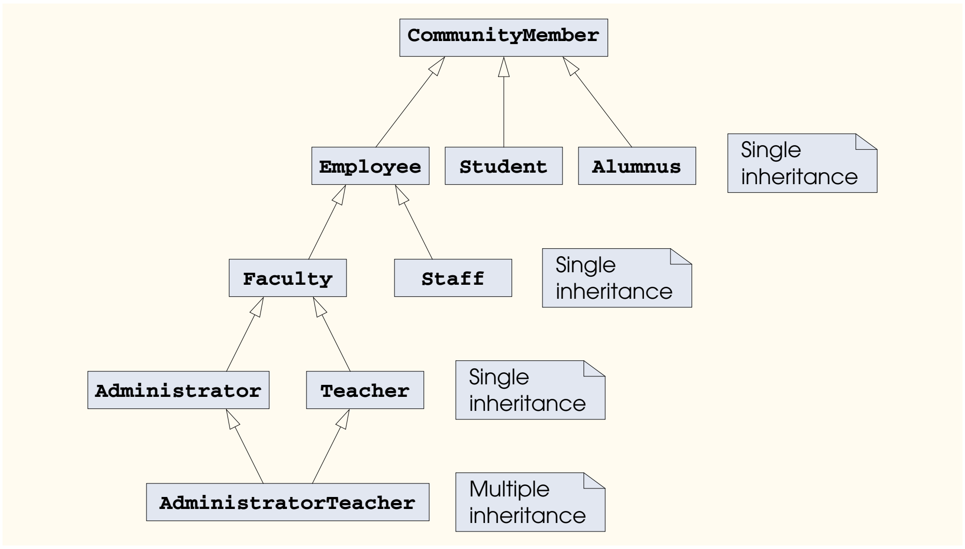

Fig. 9.2 Inheritance hierarchy for university **CommunityMember**s.

Another inheritance hierarchy is the **Shape** hierarchy of Fig. 9.3. To specify that class **TwoDimensionalShape** is derived from (or inherits from) class **Shape**, class **TwoDimensionalShape** could be defined in C++ as follows:

```
class TwoDimensionalShape : public Shape
```

This is an example of ***public** inheritance* and is the most commonly used type of inheritance. We also will discuss ***private** inheritance* and ***protected** inheritance* (Section 9.8). With **public** inheritance, **private** members of a base class are not accessible directly from that class's derived classes, but these **private** base-class members are still inherited. All other base-class members retain their original member access when they become members of the derived class (e.g., **public** members of the base class become **public** members of the derived class, and, as we will soon see, **protected** members of the base class become **protected** members of the derived class). Through these inherited base-class members, the derived class can manipulate **private** members of the base class (if these inherited members provide such functionality in the base class). Note that **friend** functions are not inherited.

Inheritance is not appropriate for every class relationship. In Chapter 7, we discussed the *has-a* relationship, in which classes have members that are objects of other classes. Such relationships create classes by composition of existing classes. For example, given the classes **Employee**, **BirthDate** and **TelephoneNumber**, it is improper to say that an **Employee** *is a* **BirthDate** or that an **Employee** *is a* **TelephoneNumber**. However, it is appropriate to say that an **Employee** *has a* **BirthDate** and that an **Employee** *has a* **TelephoneNumber**.

It is possible to treat base-class objects and derived-class objects similarly; their commonalities are expressed in the members of the base class. Objects of all classes derived from a common base class can be treated as objects of that base class (i.e., such objects have

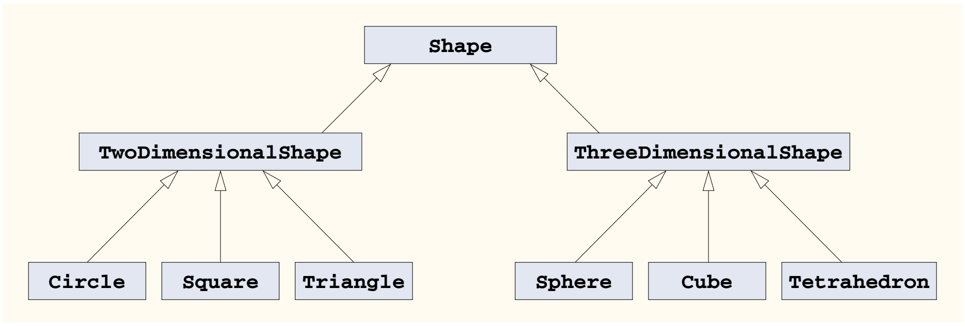

Fig. 9.3 Inheritance hierarchy for **`Shape`**s.

an "is-a" relationship with the base class). In Chapter 10, Object-Oriented Programming: Polymorphism, we consider many examples that take advantage of this relationship.

9.3 `protected` Members

Chapter 7 discussed **`public`** and **`private`** member-access specifiers. A base class's **`public`** members are accessible anywhere that the program has a handle (i.e., a name, reference or pointer) to an object of that base class or one of its derived classes. A base class's **`private`** members are accessible only within the body of that base class and the **`friend`**s of that base class. In this section, we introduce an additional member-access specifier: ***`protected`***.

Using **`protected`** access offers an intermediate level of protection between **`public`** and **`private`** access. A base class's **`protected`** members can be accessed by members and **`friend`**s of that base class and by members and **`friend`**s of any classes derived from that base class.

Derived-class member functions can refer to **`public`** and **`protected`** members of the base class simply by using the member names. When a derived-class member function redefines a base-class member function, the base-class member can be accessed from the derived class by preceding the base-class member name with the base-class name and the binary scope resolution operator (**`::`**). We discuss accessing redefined members of the base class in Section 9.4.

9.4 Relationship between Base Classes and Derived Classes

In this section, we use a point/circle inheritance hierarchy[1] to discuss the relationship between a base class and a derived class. We divide our discussion of the point/circle relationship into several parts. First, we create class **`Point`**, which contains as **`private`** data an *x*–*y* coordinate pair. Then, we create class **`Circle`**, which contains as **`private`** data an *x*–*y* coordinate pair (representing the location of the center of the circle) and a radius. We

1. The point/circle relationship may seem unnatural when we say that a circle "is a" point. This example teaches what is sometimes called *structural inheritance* and focuses on the "mechanics" of inheritance and how a base class and a derived class relate to one another. In the exercises and in Chapter 10, we present more natural inheritance examples.

do not use inheritance to create class **Circle**; rather, we construct the class by writing every line of code the class requires. Next, we create a separate **Circle2** class, which inherits directly from class **Point** (i.e., class **Circle2** "is a" **Point** but also contains a radius) and attempts to access class **Point**'s **private** members—this results in compilation errors, because the derived class does not have access to the base class's **private** data. We then show that if **Point**'s data is declared as **protected**, a **Circle3** class that inherits from class **Point2** can access that data. For this purpose, we define class **Point2** with **protected** data. Both the inherited and noninherited **Circle** classes contain identical functionality, but we show how the inherited **Circle3** class is easier to create and manage. After discussing the convenience of using **protected** data, we set the **Point** data back to **private** in class **Point3** (to enforce good software engineering), then show how a separate **Circle4** class (which inherits from class **Point3**) can use **Point3** member functions to manipulate **Point3**'s **private** data.

Creating a **Point** *Class*

Let us first examine **Point**'s class definition (Fig. 9.4–Fig. 9.5). The **Point** header file (Fig. 9.4) specifies class **Point**'s **public** services, which include a constructor (line 9) and member functions **setX** and **getX** (lines 11–12), **setY** and **getY** (lines 14–15) and **print** (line 17). The **Point** header file specifies data members **x** and **y** as **private** (lines 20–21), so objects of other classes cannot access **x** and **y** directly. Technically, even if **Point**'s data members **x** and **y** were made **public**, **Point** could never maintain an invalid state—a **Point** object's **x** and **y** data members could never contain invalid values, because the *x–y* coordinate plane is infinite in both directions. In general, however, declar-

```
1  // Fig. 9.4: point.h
2  // Point class definition represents an x-y coordinate pair.
3  #ifndef POINT_H
4  #define POINT_H
5
6  class Point {
7
8  public:
9     Point( int = 0, int = 0 ); // default constructor
10
11    void setX( int );          // set x in coordinate pair
12    int getX() const;          // return x from coordinate pair
13
14    void setY( int );          // set y in coordinate pair
15    int getY() const;          // return y from coordinate pair
16
17    void print() const;        // output Point object
18
19 private:
20    int x;  // x part of coordinate pair
21    int y;  // y part of coordinate pair
22
23 }; // end class Point
24
25 #endif
```

Fig. 9.4 **Point** class header file.

ing data members as **`private`** and providing non-**`private`** member functions to manipulate and validate the data members enforces good software engineering. [*Note*: The **`Point`** constructor definition purposely does not use member-initializer syntax in the first several examples of this section, so that we can demonstrate how **`private`** and **`protected`** specifiers affect member access in derived classes. As shown in Fig. 9.5, lines 12–13, we assign values to the data members in the constructor body. Later in this section, we will return to using member-initializer lists in the constructors.]

```
 1  // Fig. 9.5: point.cpp
 2  // Point class member-function definitions.
 3  #include <iostream>
 4
 5  using std::cout;
 6
 7  #include "point.h"   // Point class definition
 8
 9  // default constructor
10  Point::Point( int xValue, int yValue )
11  {
12     x = xValue;
13     y = yValue;
14
15  } // end Point constructor
16
17  // set x in coordinate pair
18  void Point::setX( int xValue )
19  {
20     x = xValue; // no need for validation
21
22  } // end function setX
23
24  // return x from coordinate pair
25  int Point::getX() const
26  {
27     return x;
28
29  } // end function getX
30
31  // set y in coordinate pair
32  void Point::setY( int yValue )
33  {
34     y = yValue; // no need for validation
35
36  } // end function setY
37
38  // return y from coordinate pair
39  int Point::getY() const
40  {
41     return y;
42
43  } // end function getY
44
```

Fig. 9.5 **`Point`** class represents an *x*–*y* coordinate pair. (Part 1 of 2.)

```cpp
// output Point object
void Point::print() const
{
   cout << '[' << x << ", " << y << ']';

} // end function print
```

Fig. 9.5 **`Point`** class represents an *x–y* coordinate pair. (Part 2 of 2.)

Figure 9.6 tests class **`Point`**. Line 12 instantiates object **`point`** of class **`Point`** and passes **`72`** as the *x*-coordinate value and **`115`** as the *y*-coordinate value to the constructor. Lines 15–16 use **`point`**'s **`getX`** and **`getY`** member functions to retrieve these values, then output the values. Lines 18–19 invoke **`point`**'s member functions **`setX`** and **`setY`** to change the values for **`point`**'s **`x`** and **`y`** data members. Line 23 then calls **`point`**'s **`print`** member function to display the new *x*- and *y*-coordinate values.

```cpp
 1  // Fig. 9.6: pointtest.cpp
 2  // Testing class Point.
 3  #include <iostream>
 4
 5  using std::cout;
 6  using std::endl;
 7
 8  #include "point.h"  // Point class definition
 9
10  int main()
11  {
12     Point point( 72, 115 );     // instantiate Point object
13
14     // display point coordinates
15     cout << "X coordinate is " << point.getX()
16          << "\nY coordinate is " << point.getY();
17
18     point.setX( 10 ); // set x-coordinate
19     point.setY( 10 ); // set y-coordinate
20
21     // display new point value
22     cout << "\n\nThe new location of point is ";
23     point.print();
24     cout << endl;
25
26     return 0;  // indicates successful termination
27
28  } // end main
```

```
X coordinate is 72
Y coordinate is 115

The new location of point is [10, 10]
```

Fig. 9.6 **`Point`** class test program.

Creating a* `Circle` *Class Without Using Inheritance

We now discuss the second part of our introduction to inheritance by creating and testing (a completely new) class **`Circle`** (Fig. 9.7–Fig. 9.8), which contains an *x*–*y* coordinate pair (indicating the center of the circle) and a radius. The **`Circle`** header file (Fig. 9.7) specifies class **`Circle`**'s **`public`** services, which include the **`Circle`** constructor (line 11), member functions **`setX`** and **`getX`** (lines 13–14), **`setY`** and **`getY`** (lines 16–17), **`setRadius`** and **`getRadius`** (lines 19–20), **`getDiameter`** (line 22), **`getCircumference`** (line 23), **`getArea`** (line 24) and **`print`** (line 26). Lines 29–31 declare members **`x`**, **`y`** and **`radius`** as **`private`** data. These data members and member functions encapsulate all necessary features of a circle. In Section 9.5, we show how this encapsulation enables us to reuse and extend this class.

Figure 9.9 tests class **`Circle`**. Line 17 instantiates object **`circle`** of class **`Circle`**, passing **`37`** as the *x*-coordinate value, **`43`** as the *y*-coordinate value and **`2.5`** as the radius value to the constructor. Lines 20–22 use member functions **`getX`**, **`getY`** and **`getRadius`**

```
 1  // Fig. 9.7: circle.h
 2  // Circle class contains x-y coordinate pair and radius.
 3  #ifndef CIRCLE_H
 4  #define CIRCLE_H
 5
 6  class Circle {
 7
 8  public:
 9
10     // default constructor
11     Circle( int = 0, int = 0, double = 0.0 );
12
13     void setX( int );            // set x in coordinate pair
14     int getX() const;            // return x from coordinate pair
15
16     void setY( int );            // set y in coordinate pair
17     int getY() const;            // return y from coordinate pair
18
19     void setRadius( double );    // set radius
20     double getRadius() const;    // return radius
21
22     double getDiameter() const;        // return diameter
23     double getCircumference() const;   // return circumference
24     double getArea() const;            // return area
25
26     void print() const;          // output Circle object
27
28  private:
29     int x;          // x-coordinate of Circle's center
30     int y;          // y-coordinate of Circle's center
31     double radius;  // Circle's radius
32
33  }; // end class Circle
34
35  #endif
```

Fig. 9.7 **`Circle`** class header file.

```cpp
// Fig. 9.8: circle.cpp
// Circle class member-function definitions.
#include <iostream>

using std::cout;

#include "circle.h"   // Circle class definition

// default constructor
Circle::Circle( int xValue, int yValue, double radiusValue )
{
   x = xValue;
   y = yValue;
   setRadius( radiusValue );

} // end Circle constructor

// set x in coordinate pair
void Circle::setX( int xValue )
{
   x = xValue; // no need for validation

} // end function setX

// return x from coordinate pair
int Circle::getX() const
{
   return x;

} // end function getX

// set y in coordinate pair
void Circle::setY( int yValue )
{
   y = yValue; // no need for validation

} // end function setY

// return y from coordinate pair
int Circle::getY() const
{
   return y;

} // end function getY

// set radius
void Circle::setRadius( double radiusValue )
{
   radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );

} // end function setRadius

```

Fig. 9.8 **`Circle`** class contains an *x*–*y* coordinate and a radius. (Part 1 of 2.)

```
// return radius
double Circle::getRadius() const
{
   return radius;

} // end function getRadius

// calculate and return diameter
double Circle::getDiameter() const
{
   return 2 * radius;

} // end function getDiameter

// calculate and return circumference
double Circle::getCircumference() const
{
   return 3.14159 * getDiameter();

} // end function getCircumference

// calculate and return area
double Circle::getArea() const
{
   return 3.14159 * radius * radius;

} // end function getArea

// output Circle object
void Circle::print() const
{
   cout << "Center = [" << x << ", " << y << ']'
        << "; Radius = " << radius;

} // end function print
```

Fig. 9.8 **`Circle`** class contains an *x–y* coordinate and a radius. (Part 2 of 2.)

to retrieve **`circle`**'s values, then display. Lines 24–26 invoke **`circle`**'s **`setX`**, **`setY`** and **`setRadius`** member functions to change the *x–y* coordinates and the radius, respectively. Member function **`setRadius`** (Fig. 9.8, lines 47–51) ensures that data member **`radius`** cannot be assigned a negative value (i.e., a circle cannot have a negative radius). Line 30 of Fig. 9.9 calls **`circle`**'s **`print`** member function to display its *x*-coordinate, *y*-coordinate and radius. Lines 36–42 call **`circle`**'s **`getDiameter`**, **`getCircumference`** and **`getArea`** member functions to display **`circle`**'s diameter, circumference and area, respectively.

For class **`Circle`** (Fig. 9.7–Fig. 9.8), note that much of the code is similar, if not identical, to the code in class **`Point`** (Fig. 9.4–Fig. 9.5). For example, the declaration in class **`Circle`** of **`private`** data members **`x`** and **`y`** and member functions **`setX`**, **`getX`**, **`setY`** and **`getY`** are identical to those of class **`Point`**. In addition, the **`Circle`** constructor and member function **`print`** are almost identical to those of class **`Point`**, except that they also manipulate the **radius**. The other additions to class **Circle** are **private** data member **radius** and member functions **setRadius**, **getRadius**, **getDiameter**, **getCircumference** and **getArea**.

```
// Fig. 9.9: circletest.cpp
// Testing class Circle.
#include <iostream>

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

#include <iomanip>

using std::setprecision;

#include "circle.h"  // Circle class definition

int main()
{
   Circle circle( 37, 43, 2.5 );  // instantiate Circle object

   // display point coordinates
   cout << "X coordinate is " << circle.getX()
        << "\nY coordinate is " << circle.getY()
        << "\nRadius is " << circle.getRadius();

   circle.setX( 2 );          // set new x-coordinate
   circle.setY( 2 );          // set new y-coordinate
   circle.setRadius( 4.25 );  // set new radius

   // display new point value
   cout << "\n\nThe new location and radius of circle are\n";
   circle.print();

   // display floating-point values with 2 digits of precision
   cout << fixed << setprecision( 2 );

   // display Circle's diameter
   cout << "\nDiameter is " << circle.getDiameter();

   // display Circle's circumference
   cout << "\nCircumference is " << circle.getCircumference();

   // display Circle's area
   cout << "\nArea is " << circle.getArea();

   cout << endl;

   return 0;  // indicates successful termination

} // end main
```

Fig. 9.9 **Circle** class test program. (Part 1 of 2.)

```
X coordinate is 37
Y coordinate is 43
Radius is 2.5

The new location and radius of circle are
Center = [2, 2]; Radius = 4.25
Diameter is 8.50
Circumference is 26.70
Area is 56.74
```

Fig. 9.9 **`Circle`** class test program. (Part 2 of 2.)

It appears that we literally copied code from class **`Point`**, pasted this code into class **`Circle`**, then modified class **`Circle`** to include a radius and member functions that manipulate the radius. This “copy-and-paste” approach is often error prone and time consuming. Worse yet, it can result in many physical copies of the code existing throughout a system, creating a code-maintenance nightmare. Is there a way to “absorb” the attributes and behaviors of one class in a way that makes them part of other classes without duplicating code? In the next several examples, we answer that question using a more elegant class construction approach emphasizing the benefits of inheritance.

***`Point`*/*`Circle`* Hierarchy Using Inheritance**

Now we create and test class **`Circle2`** (Fig. 9.10–Fig. 9.11), which inherits data members **`x`** and **`y`** and member functions **`setX`**, **`getX`**, **`setY`** and **`getY`** from class **`Point`** (Fig. 9.4–Fig. 9.5). An object of class **`Circle2`** “is a” **`Point`** (because inheritance absorbs the capabilities of class **`Point`**), but, as evidenced by the class **`Circle2`** header file, also contains data member **`radius`** (Fig. 9.10, line 25). The colon (**`:`**) in line 8 of the class definition indicates inheritance. Keyword **`public`** indicates the type of inheritance. As a derived class (formed with **`public`** inheritance), **`Circle2`** inherits all the members of class **`Point`**, except for the constructor. Thus, the public services of **`Circle2`** include the **`Circle2`** constructor (line 13)—each class provides its own constructors that are specific to the class—the **`public`** member functions inherited from class **`Point`**; member functions **`setRadius`** and **`getRadius`** (lines 15–16); and member functions **`getDiameter`**, **`getCircumference`**, **`getArea`** and **`print`** (lines 18–22).

```
1   // Fig. 9.10: circle2.h
2   // Circle2 class contains x-y coordinate pair and radius.
3   #ifndef CIRCLE2_H
4   #define CIRCLE2_H
5
6   #include "point.h"  // Point class definition
7
8   class Circle2 : public Point {
9
10  public:
11
12     // default constructor
13     Circle2( int = 0, int = 0, double = 0.0 );
```

Fig. 9.10 **`Circle2`** class header file. (Part 1 of 2.)

```

   void setRadius( double );   // set radius
   double getRadius() const;   // return radius

   double getDiameter() const;       // return diameter
   double getCircumference() const;  // return circumference
   double getArea() const;           // return area

   void print() const;         // output Circle2 object

private:
   double radius;  // Circle2's radius

}; // end class Circle2

#endif
```

Fig. 9.10 **Circle2** class header file. (Part 2 of 2.)

Figure 9.11 shows the member-function implementations for class **Circle2**. The constructor (lines 10–16) should set the *x*–*y* coordinate to a specific value, so lines 12–13 attempt to assign parameter values to **x** and **y** directly. The compiler generates syntax errors for lines 12 and 13 (and line 56, where **Circle2**'s **print** member function attempts to use the values of **x** and **y** directly), because the derived class **Circle2** is not allowed to access base class **Point**'s **private** data members **x** and **y**. As you can see, C++ rigidly enforces restrictions on accessing **private** data members, so that even a derived class (which is closely related to its base class) cannot access the base class's **private** data.

```
 1  // Fig. 9.11: circle2.cpp
 2  // Circle2 class member-function definitions.
 3  #include <iostream>
 4
 5  using std::cout;
 6
 7  #include "circle2.h"   // Circle2 class definition
 8
 9  // default constructor
10  Circle2::Circle2( int xValue, int yValue, double radiusValue )
11  {
12     x = xValue;
13     y = yValue;
14     setRadius( radiusValue );
15
16  } // end Circle2 constructor
17
18  // set radius
19  void Circle2::setRadius( double radiusValue )
20  {
21     radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );
22
23  } // end function setRadius
```

Fig. 9.11 Private base-class data cannot be accessed from derived class. (Part 1 of 3.)

```
24
25 // return radius
26 double Circle2::getRadius() const
27 {
28    return radius;
29
30 } // end function getRadius
31
32 // calculate and return diameter
33 double Circle2::getDiameter() const
34 {
35    return 2 * radius;
36
37 } // end function getDiameter
38
39 // calculate and return circumference
40 double Circle2::getCircumference() const
41 {
42    return 3.14159 * getDiameter();
43
44 } // end function getCircumference
45
46 // calculate and return area
47 double Circle2::getArea() const
48 {
49    return 3.14159 * radius * radius;
50
51 } // end function getArea
52
53 // output Circle2 object
54 void Circle2::print() const
55 {
56    cout << "Center = [" << x << ", " << y << ']'
57         << "; Radius = " << radius;
58
59 } // end function print
```

```
C:\cpphtp4\examples\ch09\CircleTest\circle2.cpp(12) : error C2248:
'x' : cannot access private member declared in class 'Point'
        C:\cpphtp4\examples\ch09\circletest\point.h(20) :
        see declaration of 'x'

C:\cpphtp4\examples\ch09\CircleTest\circle2.cpp(13) : error C2248:
'y' : cannot access private member declared in class 'Point'
        C:\cpphtp4\examples\ch09\circletest\point.h(21) :
        see declaration of 'y'

C:\cpphtp4\examples\ch09\CircleTest\circle2.cpp(56) : error C2248:
'x' : cannot access private member declared in class 'Point'
        C:\cpphtp4\examples\ch09\circletest\point.h(20) :
        see declaration of 'x'
```

(continued next page)

Fig. 9.11 Private base-class data cannot be accessed from derived class. (Part 2 of 3.)

```
C:\cpphtp4\examples\ch09\CircleTest\circle2.cpp(56) : error C2248:
'y' : cannot access private member declared in class 'Point'
        C:\cpphtp4\examples\ch09\circletest\point.h(21) :
        see declaration of 'y'
```

Fig. 9.11 Private base-class data cannot be accessed from derived class. (Part 3 of 3.)

Point/Circle Hierarchy Using protected Data

To enable class **Circle2** to access **Point** data members **x** and **y** directly, we can declare those members as **protected** in the base class. As we discussed in Section 9.3, a base class's **protected** members can be accessed by members and **friend**s of the base class and by members and **friend**s of any classes derived from that base class. Class **Point2** (Fig. 9.12–Fig. 9.13) is a modification of class **Point** (Fig. 9.4–Fig. 9.5) that declares data members **x** and **y** as **protected** (Fig. 9.12, lines 19–21) rather than **private**. Other than the class name change (and, hence, the constructor name change) to **Point2**, the member-function implementations in Fig. 9.13 are identical to those in Fig. 9.5.

```
 1  // Fig. 9.12: point2.h
 2  // Point2 class definition represents an x-y coordinate pair.
 3  #ifndef POINT2_H
 4  #define POINT2_H
 5
 6  class Point2 {
 7
 8  public:
 9     Point2( int = 0, int = 0 ); // default constructor
10
11     void setX( int );     // set x in coordinate pair
12     int getX() const;     // return x from coordinate pair
13
14     void setY( int );     // set y in coordinate pair
15     int getY() const;     // return y from coordinate pair
16
17     void print() const;   // output Point2 object
18
19  protected:
20     int x;   // x part of coordinate pair
21     int y;   // y part of coordinate pair
22
23  }; // end class Point2
24
25  #endif
```

Fig. 9.12 **Point2** class header file.

```
// Fig. 9.13: point2.cpp
// Point2 class member-function definitions.
#include <iostream>
```

Fig. 9.13 **Point2** class represents an *x-y* coordinate pair as **protected** data. (Part 1 of 2.)

```

using std::cout;

#include "point2.h"   // Point2 class definition

// default constructor
Point2::Point2( int xValue, int yValue )
{
   x = xValue;
   y = yValue;

} // end Point2 constructor

// set x in coordinate pair
void Point2::setX( int xValue )
{
   x = xValue; // no need for validation

} // end function setX

// return x from coordinate pair
int Point2::getX() const
{
   return x;

} // end function getX

// set y in coordinate pair
void Point2::setY( int yValue )
{
   y = yValue; // no need for validation

} // end function setY

// return y from coordinate pair
int Point2::getY() const
{
   return y;

} // end function getY

// output Point2 object
void Point2::print() const
{
   cout << '[' << x << ", " << y << ']';

} // end function print
```

Fig. 9.13 **`Point2`** class represents an *x–y* coordinate pair as **`protected`** data. (Part 2 of 2.)

Class **`Circle3`** (Fig. 9.14–Fig. 9.15) is a modification of class **`Circle2`** (Fig. 9.10–Fig. 9.11) that inherits from class **`Point2`** rather than from class **`Point`**. Because class **`Circle3`** inherits from class **`Point2`**, objects of class **`Circle3`** can access inherited data members that were declared **protected** in class **Point2** (i.e., data members **x** and **y**). As a result, the compiler does not generate errors when compiling the **Circle3** constructor and **print** member function definitions in Fig. 9.15 (lines 10–16 and 54–59, respectively). This shows the special privileges that a derived class is granted to access **protected** base-class data members. Objects of a derived class also can access **protected** members in any of that derived class's indirect base classes.

```cpp
// Fig. 9.14: circle3.h
// Circle3 class contains x-y coordinate pair and radius.
#ifndef CIRCLE3_H
#define CIRCLE3_H

#include "point2.h"  // Point2 class definition

class Circle3 : public Point2 {

public:

   // default constructor
   Circle3( int = 0, int = 0, double = 0.0 );

   void setRadius( double );   // set radius
   double getRadius() const;   // return radius

   double getDiameter() const;       // return diameter
   double getCircumference() const;  // return circumference
   double getArea() const;           // return area

   void print() const;         // output Circle3 object

private:
   double radius;  // Circle3's radius

}; // end class Circle3

#endif
```

Fig. 9.14 **Circle3** class header file.

```cpp
1  // Fig. 9.15: circle3.cpp
2  // Circle3 class member-function definitions.
3  #include <iostream>
4
5  using std::cout;
6
7  #include "circle3.h"   // Circle3 class definition
8
9  // default constructor
10 Circle3::Circle3( int xValue, int yValue, double radiusValue )
11 {
12    x = xValue;
```

Fig. 9.15 **Circle3** class that inherits from class **Point2**. (Part 1 of 2.)

```
13       y = yValue;
14       setRadius( radiusValue );
15
16    } // end Circle3 constructor
17
18    // set radius
19    void Circle3::setRadius( double radiusValue )
20    {
21       radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );
22
23    } // end function setRadius
24
25    // return radius
26    double Circle3::getRadius() const
27    {
28       return radius;
29
30    } // end function getRadius
31
32    // calculate and return diameter
33    double Circle3::getDiameter() const
34    {
35       return 2 * radius;
36
37    } // end function getDiameter
38
39    // calculate and return circumference
40    double Circle3::getCircumference() const
41    {
42       return 3.14159 * getDiameter();
43
44    } // end function getCircumference
45
46    // calculate and return area
47    double Circle3::getArea() const
48    {
49       return 3.14159 * radius * radius;
50
51    } // end function getArea
52
53    // output Circle3 object
54    void Circle3::print() const
55    {
56       cout << "Center = [" << x << ", " << y << ']'
57            << "; Radius = " << radius;
58
59    } // end function print
```

Fig. 9.15 **Circle3** class that inherits from class **Point2**. (Part 2 of 2.)

Class **Circle3** does not inherit class **Point2**'s constructor. However, class **Circle3**'s constructor (lines 10–16) calls class **Point2**'s constructor implicitly. In fact, the first task of any derived-class constructor is to call its direct base class's constructor, either implicitly or explicitly. (The syntax for calling a base-class constructor is discussed later in this section.) If the code does not include an explicit call to the base-class constructor, an implicit call is made to the base class's default constructor. Even though lines 12–13 set **x** and **y** values explicitly, the constructor first calls the **Point2** default constructor, which initializes these data members to their default **0** values. Thus, **x** and **y** each are initialized twice. We will fix this performance problem in the next examples.

Figure 9.16 performs identical tests on class **Circle3** as those that Fig. 9.9 performed on class **Circle** (Fig. 9.7–Fig. 9.8). Note that the outputs of the two programs are identical. We created class **Circle** without using inheritance and created class **Circle3** using inheritance; however, both classes provide the same functionality. Note that the code listing for class **Circle3** (i.e., the header and implementation files), which is 88 lines, is considerably shorter than the code listing for class **Circle**, which is 122 lines, because class **Circle3** absorbs part of its functionality from **Point2**, whereas class **Circle** does not absorb any functionality. Also, there is now only one copy of the point functionality mentioned in class **Point2**. This makes the code easier to debug, maintain and modify, because the point-related code exists only in the files of Fig. 9.12–Fig. 9.13.

```
// Fig. 9.16: circletest3.cpp
// Testing class Circle3.
#include <iostream>

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

#include <iomanip>

using std::setprecision;

#include "circle3.h"  // Circle3 class definition

int main()
{
   Circle3 circle( 37, 43, 2.5 ); // instantiate Circle3 object

   // display point coordinates
   cout << "X coordinate is " << circle.getX()
        << "\nY coordinate is " << circle.getY()
        << "\nRadius is " << circle.getRadius();

   circle.setX( 2 );          // set new x-coordinate
   circle.setY( 2 );          // set new y-coordinate
   circle.setRadius( 4.25 );  // set new radius

   // display new point value
   cout << "\n\nThe new location and radius of circle are\n";
   circle.print();

   // display floating-point values with 2 digits of precision
   cout << fixed << setprecision( 2 );

```

Fig. 9.16 Protected base-class data can be accessed from derived class. (Part 1 of 2.)

```
   // display Circle3's diameter
   cout << "\nDiameter is " << circle.getDiameter();

   // display Circle3's circumference
   cout << "\nCircumference is " << circle.getCircumference();

   // display Circle3's area
   cout << "\nArea is " << circle.getArea();

   cout << endl;

   return 0;  // indicates successful termination

} // end main
```

```
X coordinate is 37
Y coordinate is 43
Radius is 2.5

The new location and radius of circle are
Center = [2, 2]; Radius = 4.25
Diameter is 8.50
Circumference is 26.70
Area is 56.74
```

Fig. 9.16 Protected base-class data can be accessed from derived class. (Part 2 of 2.)

In this example, we declared base-class data members as **`protected`**, so that derived classes could modify their values directly. The use of **`protected`** data members allows for a slight increase in performance, because we avoid incurring the overhead of a call to a *set* or *get* member function. However, such performance increases are often negligible compared to the optimizations compilers can perform. It is better to use **`private`** data to encourage proper software engineering. Your code will be easier to maintain, modify and debug.

Using **`protected`** data members creates two major problems. First, the derived-class object does not have to use a member function to set the value of the base-class's **`protected`** data member. Therefore, a derived-class object easily can assign an illegal value to the **`protected`** data member, thus leaving the object in an invalid state. For example, if we were to declare **`Circle3`**'s data member **`radius`** as **`protected`**, a derived-class object (e.g., **`Cylinder`**) could then assign a negative value to **`radius`**. The second problem with using **`protected`** data members is that derived-class member functions are more likely to be written to depend on the base-class implementation. In practice, derived classes should depend only on the base-class services (i.e., non-**`private`** member functions) and not on the base-class implementation. With **`protected`** data members in the base class, if the base-class implementation changes, we may need to modify all derived classes of that base class. For example, if for some reason we were to change the names of data members **`x`** and **`y`** to **`xCoordinate`** and **`yCoordinate`**, then we would have to do so for all occurrences in which a derived class references these base-class data members directly. In such a case, the software is said to be *fragile* or *brittle*, because a small change in the base class can "break" derived-class implementation. The programmer should be able to change the base-class implementation freely, while still providing the same services to derived classes. (Of course, if the base-class services change, we must reimplement our derived classes, but good object-oriented design attempts to prevent this.)

Software Engineering Observation 9.3

It is appropriate to use the **`protected`** *access specifier when a base class should provide a service (i.e., a member function) only to its derived classes and should not provide the service to other clients.*

Software Engineering Observation 9.4

Declaring base-class data members **`private`** *(as opposed to declaring them* **`protected`***) enables programmers to change the base-class implementation without having to change derived-class implementations.*

Testing and Debugging Tip 9.1

When possible, avoid including **`protected`** *data members in a base class. Rather, include non-***`private`** *member functions that access* **`private`** *data members, ensuring that the object maintains a consistent state.*

`Point`/`Circle` Hierarchy Using `private` Data

We now reexamine our point/circle hierarchy example once more; this time, attempting to use the best software-engineering practices. Class **`Point3`** (Fig. 9.17–Fig. 9.18) declares data members **`x`** and **`y`** as **`private`** (Fig. 9.17, lines 19–21) and exposes member functions **`setX`**, **`getX`**, **`setY`**, **`getY`** and **`print`** for manipulating these values. In the constructor implementation (Fig. 9.18, lines 10–15), note that member initializers are used (line 11) to specify the values of members **`x`** and **`y`**. We show how derived-class **`Circle4`** (Fig. 9.19–Fig. 9.20) can invoke non-**`private`** base-class member functions (**`setX`**, **`getX`**, **`setY`** and **`getY`**) to manipulate these data members.

Software Engineering Observation 9.5

When possible, use member functions to alter and obtain the values of data members, even if those values can be modified directly. A set *member function can prevent attempts to assign inappropriate values to the data member, and a* get *member function can help control the presentation of the data to clients.*

Performance Tip 9.1

Using a member function to access a data member's value can be slightly slower than accessing the data directly. However, attempting to optimize programs by referencing data directly often is unnecessary, because the compiler optimizes the programs implicitly. Today's so-called "optimizing compilers" are carefully designed to perform many optimizations implicitly, even if the programmer does not write what appears to be the most optimal code. A good rule is, "Do not second-guess the compiler."

```
1  // Fig. 9.17: point3.h
2  // Point3 class definition represents an x-y coordinate pair.
3  #ifndef POINT3_H
4  #define POINT3_H
5
6  class Point3 {
7
```

Fig. 9.17 **`Point3`** class header file. (Part 1 of 2.)

```
public:
   Point3( int = 0, int = 0 ); // default constructor

   void setX( int );     // set x in coordinate pair
   int getX() const;     // return x from coordinate pair

   void setY( int );     // set y in coordinate pair
   int getY() const;     // return y from coordinate pair

   void print() const;   // output Point3 object

private:
   int x;  // x part of coordinate pair
   int y;  // y part of coordinate pair

}; // end class Point3

#endif
```

Fig. 9.17 **Point3** class header file. (Part 2 of 2.)

```
// Fig. 9.18: point3.cpp
// Point3 class member-function definitions.
#include <iostream>

using std::cout;

#include "point3.h"   // Point3 class definition

// default constructor
Point3::Point3( int xValue, int yValue )
   : x( xValue ), y( yValue )
{
   // empty body

} // end Point3 constructor

// set x in coordinate pair
void Point3::setX( int xValue )
{
   x = xValue; // no need for validation

} // end function setX

// return x from coordinate pair
int Point3::getX() const
{
   return x;

} // end function getX

```

Fig. 9.18 **Point3** class uses member functions to manipulate its **private** data. (Part 1 of 2.)

```cpp
// set y in coordinate pair
void Point3::setY( int yValue )
{
   y = yValue; // no need for validation

} // end function setY

// return y from coordinate pair
int Point3::getY() const
{
   return y;

} // end function getY

// output Point3 object
void Point3::print() const
{
   cout << '[' << getX() << ", " << getY() << ']';

} // end function print
```

Fig. 9.18 **Point3** class uses member functions to manipulate its **private** data. (Part 2 of 2.)

Class **Circle4** (Fig. 9.19–Fig. 9.20) has several changes to its member function implementations (Fig. 9.20) that distinguish it from class **Circle3** (Fig. 9.14–Fig. 9.15). Class **Circle4**'s constructor (lines 10–15) introduces *base-class initializer syntax* (line 11), which uses a member initializer to pass arguments to the base-class (**Point3**) constructor. C++ actually requires a derived-class constructor to call its base-class constructor to initialize the base-class data members that are inherited into the derived class. Line 11 accomplishes this task by invoking the **Point3** constructor by name. Values **xValue** and **yValue** are passed from the **Circle4** constructor to the **Point3** constructor to initialize base-class members **x** and **y**. If the **Circle** constructor did not invoke the **Point** constructor explicitly, the default **Point** constructor would be invoked implicitly with the default values for **x** and **y** (i.e., 0 and 0). If class **Point3** did not provide a default constructor, the compiler would issue a syntax error.

Common Programming Error 9.1

It is a syntax error if a derived-class constructor calls one of its base-class constructors with arguments that do not match exactly the number and types of parameters specified in one of the base-class constructor definitions.

In Fig. 9.15, class **Circle3**'s constructor actually initialized base-class members **x** and **y** twice. First, class **Point2**'s constructor was called implicitly with the default values **x** and **y**, then class **Circle3**'s constructor assigned values to **x** and **y** in its body.

```cpp
// Fig. 9.19: circle4.h
// Circle4 class contains x-y coordinate pair and radius.
#ifndef CIRCLE4_H
#define CIRCLE4_H
```

Fig. 9.19 **Circle4** class header file. (Part 1 of 2.)

```cpp

#include "point3.h"  // Point3 class definition

class Circle4 : public Point3 {

public:

   // default constructor
   Circle4( int = 0, int = 0, double = 0.0 );

   void setRadius( double );   // set radius
   double getRadius() const;   // return radius

   double getDiameter() const;       // return diameter
   double getCircumference() const;  // return circumference
   double getArea() const;           // return area

   void print() const;         // output Circle4 object

private:
   double radius;  // Circle4's radius

}; // end class Circle4

#endif
```

Fig. 9.19 **Circle4** class header file. (Part 2 of 2.)

```cpp
// Fig. 9.20: circle4.cpp
// Circle4 class member-function definitions.
#include <iostream>

using std::cout;

#include "circle4.h"   // Circle4 class definition

// default constructor
Circle4::Circle4( int xValue, int yValue, double radiusValue )
   : Point3( xValue, yValue )  // call base-class constructor
{
   setRadius( radiusValue );

} // end Circle4 constructor

// set radius
void Circle4::setRadius( double radiusValue )
{
   radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );

} // end function setRadius

```

Fig. 9.20 **Circle4** class that inherits from class **Point3**, which does not provide **protected** data. (Part 1 of 2.)

```
24  // return radius
25  double Circle4::getRadius() const
26  {
27     return radius;
28
29  } // end function getRadius
30
31  // calculate and return diameter
32  double Circle4::getDiameter() const
33  {
34     return 2 * getRadius();
35
36  } // end function getDiameter
37
38  // calculate and return circumference
39  double Circle4::getCircumference() const
40  {
41     return 3.14159 * getDiameter();
42
43  } // end function getCircumference
44
45  // calculate and return area
46  double Circle4::getArea() const
47  {
48     return 3.14159 * getRadius() * getRadius();
49
50  } // end function getArea
51
52  // output Circle4 object
53  void Circle4::print() const
54  {
55     cout << "Center = ";
56     Point3::print();      // invoke Point3's print function
57     cout << "; Radius = " << getRadius();
58
59  } // end function print
```

Fig. 9.20 **`Circle4`** class that inherits from class **`Point3`**, which does not provide **`protected`** data. (Part 2 of 2.)

Performance Tip 9.2

In a derived-class constructor, initializing member objects and invoking base-class constructors explicitly in the member initializer list can prevent duplicate initialization in which a default constructor is called, then data members are modified again in the body of the derived-class constructor.

In addition to the changes discusses so far, member functions **`getDiameter`** (Fig. 9.20, lines 32–36), **`getArea`** (lines 46–50) and **`print`** (lines 53–59) each invoke member function **`getRadius`** to obtain the radius value, rather than accessing the **`radius`** directly. If we decide to rename data member **`radius`**, only the bodies of functions **`setRadius`** and **`getRadius`** will need to change.

Class **`Circle4`**'s **`print`** function (Fig. 9.20, lines 53–59) redefines class **`Point3`**'s **`print`** member function (Fig. 9.18, lines 46–50). Class **`Circle4`**'s version displays the

private data members **x** and **y** of class **Point3** by calling base-class **Point3**'s **print** function with the expression **Point3::print()** (line 56). Note the syntax used to invoke a redefined base-class member function from a derived class—place the base-class name and the binary scope-resolution operator (**::**) before the base-class member-function name. This member-function invocation is a good software engineering practice: Recall that *Software Engineering Observation* 6.19 stated that, if an object's member function performs the actions needed by another object, call that member function rather than duplicating its code body. By having **Circle4**'s **print** function invoke **Point3**'s **print** function to perform part of the task of printing a **Circle4** object (i.e., to display the *x*- and *y*-coordinate values), we avoid duplicating code and reduce code-maintenance problems.

Common Programming Error 9.2

When a base-class member function is redefined in a derived class, the derived-class version often calls the base-class version to do additional work. Failure to use the **::** *reference (prefixed with the name of the base class) when referencing the base class's member function causes infinite recursion, because the derived-class member function would then call itself.*

Common Programming Error 9.3

Including a base-class member function with a different signature in the derived class hides the base-class version of the function. Attempts to call the base-class version through the **public** *interface of a derived-class object result in compilation errors.*

Figure 9.21 performs identical manipulations on a **Circle4** object as did Fig. 9.9 and Fig. 9.16 on objects of classes **Circle** and **Circle3**, respectively. Although each "circle" class behaves identically, class **Circle4** is the best engineered. Using inheritance, we have efficiently and effectively constructed a well-engineered class.

```cpp
// Fig. 9.21: circletest4.cpp
// Testing class Circle4.
#include <iostream>

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

#include <iomanip>

using std::setprecision;

#include "circle4.h"  // Circle4 class definition

int main()
{
   Circle4 circle( 37, 43, 2.5 ); // instantiate Circle4 object

   // display point coordinates
   cout << "X coordinate is " << circle.getX()
        << "\nY coordinate is " << circle.getY()
        << "\nRadius is " << circle.getRadius();
```

Fig. 9.21 Base-class **private** data is accessible to a derived class via **public** or **protected** member function inherited by the derived class. (Part 1 of 2.)

```

   circle.setX( 2 );           // set new x-coordinate
   circle.setY( 2 );           // set new y-coordinate
   circle.setRadius( 4.25 );   // set new radius

   // display new circle value
   cout << "\n\nThe new location and radius of circle are\n";
   circle.print();

   // display floating-point values with 2 digits of precision
   cout << fixed << setprecision( 2 );

   // display Circle4's diameter
   cout << "\nDiameter is " << circle.getDiameter();

   // display Circle4's circumference
   cout << "\nCircumference is " << circle.getCircumference();

   // display Circle4's area
   cout << "\nArea is " << circle.getArea();

   cout << endl;

   return 0;  // indicates successful termination

} // end main
```

```
X coordinate is 37
Y coordinate is 43
Radius is 2.5

The new location and radius of circle are
Center = [2, 2]; Radius = 4.25
Diameter is 8.50
Circumference is 26.70
Area is 56.74
```

Fig. 9.21 Base-class **private** data is accessible to a derived class via **public** or **protected** member function inherited by the derived class. (Part 2 of 2.)

9.5 Case Study: Three-Level Inheritance Hierarchy

Let us consider a more substantial inheritance example involving a three-level point/circle–cylinder hierarchy. In Section 9.4, we developed classes **Point3** (Fig. 9.17–Fig. 9.18) and **Circle4** (Fig. 9.19–Fig. 9.20). Now, we present an example in which we derive class **Cylinder** from class **Circle4**.

The first class that we use in our case study is class **Point3** (Fig. 9.17–Fig. 9.18). We declared **Point3**'s data members as **private**. Class **Point3** also contains member functions **setX**, **getX**, **setY** and **getY** for accessing **x** and **y**, and member function **print** for displaying the *x–y* coordinate pair on the standard output.

We also use class **Circle4** (Fig. 9.19–Fig. 9.20), which inherits from class **Point3**. Class **Circle4** contains functionality from class **Point3** and provides member function **setRadius**, which ensures that the **radius** data member cannot hold a negative value, and member functions **getRadius**, **getDiameter**, **getCircumference**, **getArea** and **print**. Derived classes of class **Circle4** (such as class **Cylinder**, which we introduce momentarily) should redefine these member functions as necessary to provide implementations specific to the derived class. For example, a circle has an area that is calculated by the formula, πr^2, in which *r* represents the circle's radius. However, a cylinder has a surface area that is calculated by the formula, $(2\pi r^2) + (2\pi rh)$, in which *r* represents the cylinder's radius and *h* represents the cylinder's height. Therefore, class **Cylinder** should redefine member function **getArea** to include this calculation.

Figure 9.22–Fig. 9.23 present class **Cylinder**, which inherits from class **Circle4**. The **Cylinder** header file (Fig. 9.22) specifies that a **Cylinder** has a **height** (line 23) and specifies class **Cylinder**'s **public** services, which include inherited **Circle4** member functions (line 8) **setRadius**, **getRadius**, **getDiameter**, **getCircumference**, **getArea** and **print**; indirectly inherited **Point3** member functions **setX**, **getX**, **setY** and **getY**; the **Cylinder** constructor (line 13); and **Cylinder** member functions **setHeight**, **getHeight**, **getArea**, **getVolume** and **print** (lines 15–20). Member functions **getArea** and **print** redefine the member functions with the same names that are inherited from class **Circle4**.

Figure 9.23 shows class **Cylinder**'s member-function implementations. Member function **getArea** (lines 33–38) redefines member function **getArea** of class **Circle4** to calculate surface area. Member function **print** (lines 48–53) redefines member function **print** of class **Circle4** to display the text representation of the cylinder to the standard

```
 1  // Fig. 9.22: cylinder.h
 2  // Cylinder class inherits from class Circle4.
 3  #ifndef CYLINDER_H
 4  #define CYLINDER_H
 5
 6  #include "circle4.h"  // Circle4 class definition
 7
 8  class Cylinder : public Circle4 {
 9
10  public:
11
12     // default constructor
13     Cylinder( int = 0, int = 0, double = 0.0, double = 0.0 );
14
15     void setHeight( double );  // set Cylinder's height
16     double getHeight() const;  // return Cylinder's height
17
18     double getArea() const;    // return Cylinder's area
19     double getVolume() const;  // return Cylinder's volume
20     void print() const;        // output Cylinder
21
22  private:
23     double height;  // Cylinder's height
24
25  }; // end class Cylinder
26
27  #endif
```

Fig. 9.22 **Cylinder** class header file.

output. Class **Cylinder** also includes member function **getVolume** (lines 41–45) to calculate the cylinder's volume.

Figure 9.24 is a **CylinderTest** application that tests class **Cylinder**. Line 18 instantiates a **Cylinder** object called **cylinder**. Lines 21–24 use **cylinder**'s

```
 1  // Fig. 9.23: cylinder.cpp
 2  // Cylinder class inherits from class Circle4.
 3  #include <iostream>
 4
 5  using std::cout;
 6
 7  #include "cylinder.h"   // Cylinder class definition
 8
 9  // default constructor
10  Cylinder::Cylinder( int xValue, int yValue, double radiusValue,
11     double heightValue )
12     : Circle4( xValue, yValue, radiusValue )
13  {
14     setHeight( heightValue );
15
16  } // end Cylinder constructor
17
18  // set Cylinder's height
19  void Cylinder::setHeight( double heightValue )
20  {
21     height = ( heightValue < 0.0 ? 0.0 : heightValue );
22
23  } // end function setHeight
24
25  // get Cylinder's height
26  double Cylinder::getHeight() const
27  {
28     return height;
29
30  } // end function getHeight
31
32  // redefine Circle4 function getArea to calculate Cylinder area
33  double Cylinder::getArea() const
34  {
35     return 2 * Circle4::getArea() +
36        getCircumference() * getHeight();
37
38  } // end function getArea
39
40  // calculate Cylinder volume
41  double Cylinder::getVolume() const
42  {
43     return Circle4::getArea() * getHeight();
44
45  } // end function getVolume
46
```

Fig. 9.23 **Cylinder** class inherits from class **Circle4** and redefines member function **getArea**. (Part 1 of 2.)

```
47  // output Cylinder object
48  void Cylinder::print() const
49  {
50     Circle4::print();
51     cout << "; Height = " << getHeight();
52
53  } // end function print
```

Fig. 9.23 **`Cylinder`** class inherits from class **`Circle4`** and redefines member function **`getArea`**. (Part 2 of 2.)

member functions **getX**, **getY**, **getRadius** and **getHeight** to obtain information about **cylinder**, because **CylinderTest** cannot reference the **private** data members of class **Cylinder** directly. Lines 26–29 use member functions **setX**, **setY**, **setRadius** and **setHeight** to reset **cylinder**'s *x–y* coordinates (we assume the cylinder's *x–y* coordinates specify the position of the center of its bottom on the *x–y* plane), radius and height. Class **Cylinder** can use class **Point3**'s **setX**, **getX**, **setY** and **getY** member functions, because class **Cylinder** inherits them indirectly from class **Point3**. (Class **Cylinder** inherits member functions **setX**, **getX**, **setY** and **getY** directly from class **Circle4**, which inherited them directly from class **Point3**.) Line 33 invokes **cylinder**'s **print** member function to display the text representation of object **cylinder**. Lines 39 and 43 invoke member functions **getDiameter** and **getCircumference** of the **cylinder** object—because class **Cylinder** inherits these functions from class **Circle4**, these member functions, exactly as defined in **Circle4**, are invoked. Lines 46 and 49 invoke member functions **getArea** and **getVolume** to determine the surface area and volume of **cylinder**.

```
1   // Fig. 9.24: cylindertest.cpp
2   // Testing class Cylinder.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7   using std::fixed;
8
9   #include <iomanip>
10
11  using std::setprecision;
12
13  #include "cylinder.h"  // Cylinder class definition
14
15  int main()
16  {
17     // instantiate Cylinder object
18     Cylinder cylinder( 12, 23, 2.5, 5.7 );
19
20     // display point coordinates
21     cout << "X coordinate is " << cylinder.getX()
22          << "\nY coordinate is " << cylinder.getY()
```

Fig. 9.24 **`Point`/`Circle`/`Cylinder`** hierarchy test program. (Part 1 of 2.)

```
         << "\nRadius is " << cylinder.getRadius()
         << "\nHeight is " << cylinder.getHeight();

    cylinder.setX( 2 );          // set new x-coordinate
    cylinder.setY( 2 );          // set new y-coordinate
    cylinder.setRadius( 4.25 );  // set new radius
    cylinder.setHeight( 10 );    // set new height

    // display new cylinder value
    cout << "\n\nThe new location and radius of circle are\n";
    cylinder.print();

    // display floating-point values with 2 digits of precision
    cout << fixed << setprecision( 2 );

    // display cylinder's diameter
    cout << "\n\nDiameter is " << cylinder.getDiameter();

    // display cylinder's circumference
    cout << "\nCircumference is "
         << cylinder.getCircumference();

    // display cylinder's area
    cout << "\nArea is " << cylinder.getArea();

    // display cylinder's volume
    cout << "\nVolume is " << cylinder.getVolume();

    cout << endl;

    return 0;  // indicates successful termination

} // end main
```

```
X coordinate is 12
Y coordinate is 23
Radius is 2.5
Height is 5.7

The new location and radius of circle are
Center = [2, 2]; Radius = 4.25; Height = 10

Diameter is 8.50
Circumference is 26.70
Area is 380.53
Volume is 567.45
```

Fig. 9.24 **`Point`/`Circle`/`Cylinder`** hierarchy test program. (Part 2 of 2.)

Using the point/circle/cylinder example, we have shown the use and benefits of inheritance. We were able to develop classes **`Circle4`** and **`Cylinder`** much more quickly by using inheritance than if we had developed these classes “from scratch.” Inheritance avoids duplicating code and the associated code-maintenance problems.

9.6 Constructors and Destructors in Derived Classes

As we explained in the previous section, instantiating a derived-class object begins a chain of constructor calls in which the derived-class constructor, before performing its own tasks, invokes its direct base class's constructor either explicitly or implicitly. Similarly, if the base class were derived from another class, the base-class constructor would be required to invoke the constructor of the next class up in the hierarchy, and so on. The last constructor called in the chain is defined in the class at the base of the inheritance hierarchy (for example, class **`Point3`**, in the **`Point3`**/**`Circle4`**/**`Cylinder`** hierarchy), whose body actually finishes executing first. The original derived-class constructor's body finishes executing last. Each base-class constructor initializes the base-class data members that the derived-class object inherits. For example, again consider the **`Point3`**/**`Circle4`**/**`Cylinder`** hierarchy from Fig. 9.18, Fig. 9.20 and Fig. 9.23. When a program creates a **`Cylinder`** object, the **`Cylinder`** constructor is called. That constructor calls **`Circle4`**'s constructor, which in turn calls **`Point3`**'s constructor. The **`Point3`** constructor initializes the *x*–*y* coordinates of the **`Cylinder`** object. When **`Point3`**'s constructor completes execution, it returns control to **`Circle4`**'s constructor, which initializes the **`Cylinder`** object's radius. When **`Circle4`**'s constructor completes execution, it returns control to **`Cylinder`**'s constructor, which initializes the **`Cylinder`** object's height.

Software Engineering Observation 9.6

When a program creates a derived-class object, the derived-class constructor immediately calls the base-class constructor, the base-class constructor's body executes, then the derived-class constructor's body executes.

When a derived-class object is destroyed, the program then calls that object's destructor. This begins a chain of destructor calls in which the derived-class destructor and the destructors of the direct and indirect base classes execute in reverse of the order in which the constructors executed. When a derived-class object's destructor is called, the destructor performs its task, then invokes the destructor of the next base class in the hierarchy. This process repeats until the destructor of the final base class at the top of the hierarchy is called. Then the object is removed from memory.

Software Engineering Observation 9.7

Suppose that we create an object of a derived class where both the base class and the derived class contain objects of other classes. When an object of that derived class is created, first the constructors for the base class's member objects execute, then the base-class constructor executes, then the constructors for the derived class's member objects execute, then the derived class's constructor executes. Destructors are called in the reverse of the order in which their corresponding constructors are called.

Base-class constructors, destructors and assignment operators are not inherited by derived classes. Derived-class constructors and assignment operators, however, can call base-class constructors and assignment operators.

Our next example revisits the point/circle hierarchy by defining class **`Point4`** (Fig. 9.25–Fig. 9.26) and class **`Circle5`** (Fig. 9.27–Fig. 9.28) that contain constructors and destructors, each of which prints a message when it is invoked.

Class **`Point4`** (Fig. 9.25–Fig. 9.26) contains the features from class **`Point`** (Fig. 9.4–Fig. 9.5). We modified the constructor (lines 11–18 of Fig. 9.26) and included a destructor (lines 21–27), each of which outputs a line of text upon its invocation.

```
// Fig. 9.25: point4.h
// Point4 class definition represents an x-y coordinate pair.
#ifndef POINT4_H
#define POINT4_H

class Point4 {

public:
   Point4( int = 0, int = 0 ); // default constructor
   ~Point4();                  // destructor

   void setX( int );     // set x in coordinate pair
   int getX() const;     // return x from coordinate pair

   void setY( int );     // set y in coordinate pair
   int getY() const;     // return y from coordinate pair

   void print() const;   // output Point3 object

private:
   int x;  // x part of coordinate pair
   int y;  // y part of coordinate pair

}; // end class Point4

#endif
```

Fig. 9.25 **`Point4`** class header file.

```
// Fig. 9.26: point4.cpp
// Point4 class member-function definitions.
#include <iostream>

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

#include "point4.h"   // Point4 class definition

// default constructor
Point4::Point4( int xValue, int yValue )
   : x( xValue ), y( yValue )
{
   cout << "Point4 constructor: ";
   print();
   cout << endl;

} // end Point4 constructor

// destructor
Point4::~Point4()
{
   cout << "Point4 destructor: ";
```

Fig. 9.26 **`Point4`** base class contains a constructor and a destructor. (Part 1 of 2.)

```
   print();
   cout << endl;

} // end Point4 destructor

// set x in coordinate pair
void Point4::setX( int xValue )
{
   x = xValue; // no need for validation

} // end function setX

// return x from coordinate pair
int Point4::getX() const
{
   return x;

} // end function getX

// set y in coordinate pair
void Point4::setY( int yValue )
{
   y = yValue; // no need for validation

} // end function setY

// return y from coordinate pair
int Point4::getY() const
{
   return y;

} // end function getY

// output Point4 object
void Point4::print() const
{
   cout << '[' << getX() << ", " << getY() << ']';

} // end function print
```

Fig. 9.26 **`Point4`** base class contains a constructor and a destructor. (Part 2 of 2.)

Class **`Circle5`** (Fig. 9.27–Fig. 9.28) contains features from class **`Circle4`** (Fig. 9.19–Fig. 9.20). We modified the constructor (lines 11–20 of Fig. 9.28) and included a destructor (lines 23–29), each of which outputs a line of text upon its invocation.

```
// Fig. 9.27: circle5.h
// Circle5 class contains x-y coordinate pair and radius.
#ifndef CIRCLE5_H
#define CIRCLE5_H
```

Fig. 9.27 **`Circle5`** class header file. (Part 1 of 2.)

```

#include "point4.h"  // Point4 class definition

class Circle5 : public Point4 {

public:

   // default constructor
   Circle5( int = 0, int = 0, double = 0.0 );

   ~Circle5();                  // destructor
   void setRadius( double );    // set radius
   double getRadius() const;    // return radius

   double getDiameter() const;       // return diameter
   double getCircumference() const;  // return circumference
   double getArea() const;           // return area

   void print() const;          // output Circle5 object

private:
   double radius;  // Circle5's radius

}; // end class Circle5

#endif
```

Fig. 9.27 **Circle5** class header file. (Part 2 of 2.)

```
// Fig. 9.28: circle5.cpp
// Circle5 class member-function definitions.
#include <iostream>

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

#include "circle5.h"   // Circle5 class definition

// default constructor
Circle5::Circle5( int xValue, int yValue, double radiusValue )
   : Point4( xValue, yValue )  // call base-class constructor
{
   setRadius( radiusValue );

   cout << "Circle5 constructor: ";
   print();
   cout << endl;

} // end Circle5 constructor
```

Fig. 9.28 **Circle5** class inherits from class **Point4**. (Part 1 of 2.)

```

// destructor
Circle5::~Circle5()
{
   cout << "Circle5 destructor: ";
   print();
   cout << endl;

} // end Circle5 destructor

// set radius
void Circle5::setRadius( double radiusValue )
{
   radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );

} // end function setRadius

// return radius
double Circle5::getRadius() const
{
   return radius;

} // end function getRadius

// calculate and return diameter
double Circle5::getDiameter() const
{
   return 2 * getRadius();

} // end function getDiameter

// calculate and return circumference
double Circle5::getCircumference() const
{
   return 3.14159 * getDiameter();

} // end function getCircumference

// calculate and return area
double Circle5::getArea() const
{
   return 3.14159 * getRadius() * getRadius();

} // end function getArea

// output Circle5 object
void Circle5::print() const
{
   cout << "Center = ";
   Point4::print();      // invoke Point4's print function
   cout << "; Radius = " << getRadius();

} // end function print
```

Fig. 9.28 **Circle5** class inherits from class **Point4**. (Part 2 of 2.)

Figure 9.29 demonstrates the order in which constructors and destructors are called for objects of classes that are part of an inheritance hierarchy. Function **`main`** (lines 11–29) begins by instantiating a **`Point4`** object (line 15) in a separate block inside **`main`** (lines 13–17). The object goes in and out of scope immediately (the end of the block is reached as soon

```cpp
 1  // Fig. 9.29: fig09_29.cpp
 2  // Display order in which base-class and derived-class
 3  // constructors are called.
 4  #include <iostream>
 5
 6  using std::cout;
 7  using std::endl;
 8
 9  #include "circle5.h"  // Circle5 class definition
10
11  int main()
12  {
13     { // begin new scope
14
15        Point4 point( 11, 22 );
16
17     } // end scope
18
19     cout << endl;
20     Circle5 circle1( 72, 29, 4.5 );
21
22     cout << endl;
23     Circle5 circle2( 5, 5, 10 );
24
25     cout << endl;
26
27     return 0;  // indicates successful termination
28
29  } // end main
```

```
Point4 constructor: [11, 22]
Point4 destructor: [11, 22]

Point4 constructor: [72, 29]
Circle5 constructor: Center = [72, 29]; Radius = 4.5

Point4 constructor: [5, 5]
Circle5 constructor: Center = [5, 5]; Radius = 10

Circle5 destructor: Center = [5, 5]; Radius = 10
Point4 destructor: [5, 5]
Circle5 destructor: Center = [72, 29]; Radius = 4.5
Point4 destructor: [72, 29]
```

Fig. 9.29 Constructor and destructor call order.

as the object is created), so both the **`Point4`** constructor and destructor are called. Next, line 20 instantiates **`Circle5`** object **`circle1`**. This invokes the **`Point4`** constructor to perform output with values passed from the **`Circle5`** constructor, then performs the output specified in the **`Circle5`** constructor. Line 23 then instantiates **`Circle5`** object **`circle2`**. Again, the **`Point4`** and **`Circle5`** constructors are both called. Note that, in each case, the body of the **`Point4`** constructor is executed before the body of the **`Circle5`** constructor executes. When the end of **`main`** is reached, the destructors are called for objects **`circle1`** and **`circle2`**. But, because destructors are called in the reverse order of their corresponding constructors, the **`Circle5`** destructor and **`Point4`** destructor are called (in that order) for object **`circle2`**, then the **`Circle5`** and **`Point4`** destructors are called (in that order) for object **`circle1`**.

9.7 "Uses A" and "Knows A" Relationships

Inheritance and composition encourage software reuse by creating classes that take advantage of functionality and data defined in existing classes. There are other ways to use the services of classes. Although a person object is not a car and a person object does not contain a car, a person object certainly *uses a* car. A function *uses* an object simply by calling a non-**`private`** member function of that object using a pointer, reference or the object name itself.

An object can be *aware of* another object. Knowledge networks frequently have such relationships. One object can contain a pointer handle or a reference handle to another object to be aware of that object. In this case, one object is said to have a *knows a* relationship with the other object; this is sometimes called an *association*.

9.8 `public`, `protected` and `private` Inheritance

When deriving a class from a base class, the base class may be inherited through **`public`**, **`protected`** or **`private`** inheritance. Use of **`protected`** and **`private`** inheritance is rare and each should be used only with great care; we normally use **`public`** inheritance in this book. (Chapter 17 demonstrates **`private`** inheritance as an alternative to composition.) Figure 9.30 summarizes for each type of inheritance the accessibility of base-class members in a derived class. The first column contains the base-class member-access specifiers.

When deriving a class from a **`public`** base class, **`public`** members of the base class become **`public`** members of the derived class and **`protected`** members of the base class become **`protected`** members of the derived class. A base class's **`private`** members are never accessible directly from a derived class, but can be accessed through calls to the **`public`** and **`protected`** members of the base class.

When deriving from a **`protected`** base class, **`public`** and **`protected`** members of the base class become **`protected`** members of the derived class. When deriving from a **`private`** base class, **`public`** and **`protected`** members of the base class become **`private`** members (e.g., the functions become utility functions) of the derived class. **`Private`** and **`protected`** inheritance are not *is-a* relationships.

Base-class member-access specifier	Type of inheritance		
	public inheritance	**protected** inheritance	**private** inheritance
public	**public** in derived class. Can be accessed directly by non-**static** member functions, **friend** functions and nonmember functions.	**protected** in derived class. Can be accessed directly by non-**static** member functions and **friend** functions.	**private** in derived class. Can be accessed directly by non-**static** member functions and **friend** functions.
protected	**protected** in derived class. Can be accessed directly by non-**static** member functions and **friend** functions.	**protected** in derived class. Can be accessed directly by non-**static** member functions and **friend** functions'.	**private** in derived class. Can be accessed directly by non-**static** member functions and **friend** functions.
private	Hidden in derived class. Can be accessed by non-**static** member functions and **friend** functions through **public** or **protected** member functions of the base class.	Hidden in derived class. Can be accessed by non-**static** member functions and **friend** functions through **public** or **protected** member functions of the base class.	Hidden in derived class. Can be accessed by non-**static** member functions and **friend** functions through **public** or **protected** member functions of the base class.

Fig. 9.30 Summary of base-class member accessibility in a derived class.

9.9 Software Engineering with Inheritance

In this section, we discuss the use of inheritance to customize existing software. When we use inheritance to create a new class from an existing one, the new class inherits the data members and member functions of the existing class. We can customize the new class to meet our needs by including additional members and by redefining base-class members. This is done in C++ without the derived-class programmer accessing the base class's source code. The derived class must be able to link to the base class's object code. This powerful capability is attractive to independent software vendors (ISVs). ISVs can develop proprietary classes for sale or license and make these classes available to users in object-code format. Users then can derive new classes from these library classes rapidly and without accessing the ISVs' proprietary source code. All the ISVs need to supply with the object code are the header files.

Sometimes, it is difficult for students to appreciate the scope of problems faced by designers who work on large-scale software projects in industry. People experienced with such projects say that effective software reuse improves the software-development process. Object-oriented programming facilitates software reuse, thus shortening development times.

The availability of substantial and useful class libraries delivers the maximum benefits of software reuse through inheritance. Interest in class libraries is growing exponentially. Just as shrink-wrapped software produced by independent software vendors became an explosive-growth industry with the arrival of the personal computer, so, too, is the creation and sale of class libraries. Application designers build their applications with these libraries, and library designers are being rewarded by having their libraries included with the applications. The standard C++ libraries that are shipped with C++ compilers tend to be rather general purpose and limited in scope. However, there is massive worldwide commitment to the development of class libraries for a huge variety of applications arenas.

Software Engineering Observation 9.8

At the design stage in an object-oriented system, the designer often determines that certain classes are closely related. The designer should "factor out" common attributes and behaviors and place these in a base class. Then use inheritance to form derived classes, endowing them with capabilities beyond those inherited from the base class.

Software Engineering Observation 9.9

The creation of a derived class does not affect its base class's source code. Inheritance preserves the integrity of a base class.

Software Engineering Observation 9.10

Just as designers of non-object-oriented systems should avoid proliferation of functions, designers of object-oriented systems should avoid proliferation of classes. Proliferation of classes creates management problems and can hinder software reusability, because it becomes difficult for a client to locate the most appropriate class of a huge class library. The alternative is to create fewer classes that provide more substantial functionality, but such classes might provide too much functionality.

Performance Tip 9.3

If classes produced through inheritance are larger than they need to be (i.e., contain too much functionality), memory and processing resources might be wasted. Inherit from the class whose functionality is "closest" to what is needed.

Reading derived-class definitions can be confusing, because inherited members are not shown physically in the derived class, but nevertheless are present in the derived classes. A similar problem exists when documenting derived-class members.

In this chapter, we introduced inheritance—the ability to create classes by absorbing an existing class's data members and member functions, and embellishing these with new capabilities. In Chapter 10, we build upon our discussion of inheritance by introducing *polymorphism*—an object-oriented technique that enables us to write programs that handle, in a more general manner, a wide variety of classes related by inheritance. After studying Chapter 10, you will be familiar with classes, encapsulation, inheritance and polymorphism—the most crucial aspects of object-oriented programming.

9.10 (Optional Case Study) Thinking About Objects: Incorporating Inheritance into the Elevator Simulation

We now examine our simulation design to decide whether it might benefit from inheritance. In the previous "Thinking About Objects" sections, we have been treating **`ElevatorButton`** and **`FloorButton`** as separate classes. In fact, these classes have much in common; each is a *kind of* a button. To apply inheritance, we first look for commonality between these classes. We then extract this commonality, place it into base class **`Button`** and derive classes **`ElevatorButton`** and **`FloorButton`** from **`Button`**.

Let us now examine the similarities between classes **ElevatorButton** and **FloorButton**. Figure 9.31 shows the attributes and operations of both classes, as declared in their header files from Chapter 7 (Fig. 7.37 and Fig. 7.39, respectively). The classes have in common one attribute (**pressed**) and two operations (**pressButton** and **resetButton**). We place these three elements in base-class **Button**, then **ElevatorButton** and **FloorButton** inherit the attributes and operations of **Button**. In our previous implementation, **ElevatorButton** and **FloorButton** each declared a reference to an object of class **Elevator**—class **Button** also should contain this reference.

Figure 9.32 shows our modified elevator simulator design, which incorporates inheritance. Class **Floor** is composed of one object of class **FloorButton** and one object of class **Light**. In addition, class **Elevator** is composed of one object of class **ElevatorButton**, one object of class **Door** and one object of class **Bell**. A solid line with a hollow arrowhead extends from each of the derived classes to the base class—this line indicates that classes **FloorButton** and **ElevatorButton** inherit from class **Button**.

One question remains: Should the derived classes redefine any of the base-class member functions? If we compare the public member functions of each class (Fig. 7.38 and Fig. 7.40), we notice that the **resetButton** member function is identical for both classes. This function does not need to be redefined. However, the implementation of member function **pressButton** differs for each class. Class **ElevatorButton** contains the **pressButton** code

```
pressed = true;
cout << "elevator button tells elevator to prepare to leave"
     << endl;
elevatorRef.prepareToLeave( true );
```

whereas class **FloorButton** contains this different **pressButton** code

```
pressed = true;
cout << "floor " << floorNumber
     << " button summons elevator" << endl;
elevatorRef.summonElevator( floorNumber );
```

The first line of each block of code is identical, but the remaining sections are different. Therefore, each derived class must redefine the base-class **Button** member function **pressButton**.

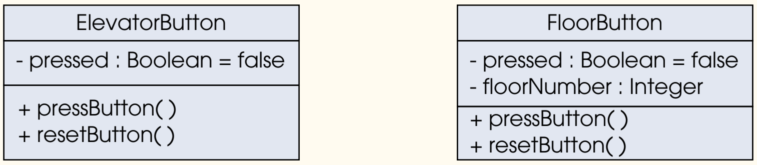

Fig. 9.31 Attributes and operations of classes **ElevatorButton** and **FloorButton**.

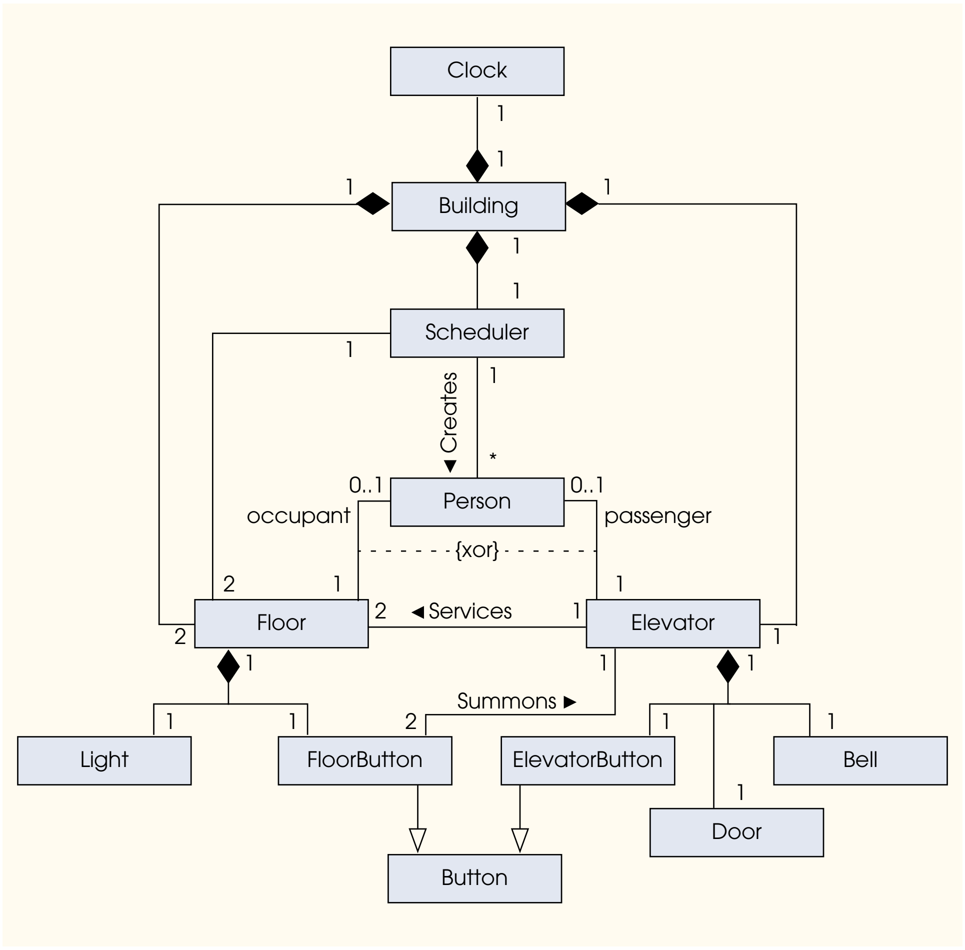

Fig. 9.32 Class diagram incorporating inheritance into the elevator-simulator.

Figure 9.33 lists the header file for the base class **Button**.[2] We declare **public** member functions **pressButton** and **resetButton** (lines 13–14) and **private** data member **pressed** of type **bool** (line 22). Notice the declaration of the reference to an **Elevator** object in line 19 and the corresponding parameter to the constructor in line 11. We show how to initialize the reference when we discuss the code for the derived classes.

The derived classes perform two different actions. Class **ElevatorButton** invokes the **prepareToLeave** member function of class **Elevator**; class **FloorButton** invokes the **summonElevator** member function. Thus, both classes need access to the

2. The benefit of encapsulation is that no other files in our elevator simulation need to be changed. We simply substitute the new **elevatorButton** and **floorButton** header and implementation files for the old ones and add the files for class **Button**.

elevatorRef data member of the base class; however, this data member should not be available to non-**Button** objects. Therefore, we place the **elevatorRef** data member in the **protected** section of **Button**. Only base-class member functions directly manipulate data member **pressed**, so we declare this data member as **private**. Derived classes do not need to access **pressed** directly.

Figure 9.34 lists the implementation file for class **Button**. Line 12 in the constructor initializes the reference to the elevator. The constructor and destructor display messages indicating that they are running, and the **pressButton** and **resetButton** member functions manipulate **private** data member **pressed**.

```
// Fig. 9.33: button.h
// Definition for class Button.
#ifndef BUTTON_H
#define BUTTON_H

class Elevator;                       // forward declaration

class Button {

public:
   Button( Elevator & );              // constructor
   ~Button();                         // destructor
   void pressButton();                // sets button on
   void resetButton();                // resets button off

protected:

   // reference to button's elevator
   Elevator &elevatorRef;

private:
   bool pressed;                      // state of button

}; // end class Button

#endif // BUTTON_H
```

Fig. 9.33 **Button** class header file.

```
1   // Fig. 9.34: button.cpp
2   // Member function definitions for class Button.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   #include "button.h"  // Button class definition
9
```

Fig. 9.34 **Button** class implementation file—base class for **ElevatorButton** and **FloorButton**. (Part 1 of 2.)

```
10  // constructor
11  Button::Button( Elevator &elevatorHandle )
12     : elevatorRef( elevatorHandle ), pressed( false )
13  {
14     cout << "button constructed" << endl;
15
16  } // end Button constructor
17
18  // destructor
19  Button::~Button()
20  {
21     cout << "button destructed" << endl;
22
23  } // end Button destructor
24
25  // press button
26  void Button::pressButton()
27  {
28     pressed = true;
29
30  } // end function pressButton
31
32  // reset button
33  void Button::resetButton()
34  {
35     pressed = false;
36
37  } // end function resetButton
```

Fig. 9.34 **Button** class implementation file—base class for **ElevatorButton** and **FloorButton**. (Part 2 of 2.)

Figure 9.35 contains the header file for class **ElevatorButton**. Line 8 indicates that the class inherits from class **Button**. This inheritance means that class **ElevatorButton** contains the protected **elevatorRef** data member and the public **pressButton** and **resetButton** member functions of the base class. In line 13, we provide a function prototype for **pressButton** to signal our intent to redefine that member function in the **.cpp** file. We discuss the **pressButton** implementation momentarily.

The constructor takes as a parameter a reference to class **Elevator** (line 11). We discuss the necessity for this parameter when we discuss the class's implementation. Notice, however, that we do not need to include a forward declaration of class **Elevator** in the derived class, because the base-class header file contains the forward reference.

```
1   // Fig. 9.35: elevatorButton.h
2   // ElevatorButton class definition.
3   #ifndef ELEVATORBUTTON_H
4   #define ELEVATORBUTTON_H
5
6   #include "button.h"  // Button class definition
```

Fig. 9.35 **ElevatorButton** class header file. (Part 1 of 2.)

```

class ElevatorButton : public Button {

public:
   ElevatorButton( Elevator & );  // constructor
   ~ElevatorButton();             // destructor
   void pressButton();            // press the button

}; // end class ElevatorButton

#endif // ELEVATORBUTTON_H
```

Fig. 9.35 **`ElevatorButton`** class header file. (Part 2 of 2.)

Figure 9.36 lists the implementation file of class **`ElevatorButton`**. The class constructors and destructors display messages to indicate that these functions are executing. Line 13 passes the **`Elevator`** reference to the base-class constructor.

```
 1  // Fig. 9.36: elevatorButton.cpp:
 2  // Member-function definitions for class ElevatorButton.
 3  #include <iostream>
 4
 5  using std::cout;
 6  using std::endl;
 7
 8  #include "elevatorButton.h"  // ElevatorButton class definition
 9  #include "elevator.h"        // Elevator class definition
10
11  // constructor
12  ElevatorButton::ElevatorButton( Elevator &elevatorHandle )
13     : Button( elevatorHandle )
14  {
15     cout << "elevator button constructed" << endl;
16
17  } // end ElevatorButton constructor
18
19  // destructor
20  ElevatorButton::~ElevatorButton()
21  {
22     cout << "elevator button destructed" << endl;
23
24  } // end ~ElevatorButton destructor
25
26  // press button and signal elevator to prepare to leave floor
27  void ElevatorButton::pressButton()
28  {
29     Button::pressButton();
30     cout << "elevator button tells elevator to prepare to leave"
31          << endl;
32     elevatorRef.prepareToLeave( true );
33
34  } // end function pressButton
```

Fig. 9.36 **`ElevatorButton`** class member-function definitions.

Member function **pressButton** first calls the **pressButton** member function (line 29) in base class **Button**; this call sets to **true** the **pressed** attribute of class **Button**. Line 32 notifies the elevator to move to the other floor by passing **true** to member function **prepareToLeave**.

Figure 9.37 lists the header file for class **FloorButton**. The only difference between this file and the header file for class **ElevatorButton** is the addition in line 16 of the **floorNumber** data member. We use this data member to distinguish the floors in the simulation output messages. The constructor declaration includes a parameter of type **int** (line 11), so the **FloorButton** object can initialize attribute **floorNumber**.

Figure 9.38 shows the implementation of class **FloorButton**. Lines 13–14 pass the **Elevator** reference to the base-class constructor and initialize the **floorNumber** data member. The constructor (lines 12–19) and destructor (lines 22–27) output appropriate messages, using data member **floorNumber**. The redefined **pressButton** member function (lines 30–39) first calls member function **pressButton** (line 32) in the base class, then invokes the elevator's **summonElevator** member function (line 37), passing **floorNumber** to indicate the floor that summoned the elevator.

```
1  // Fig. 9.37: floorButton.h
2  // FloorButton class definition.
3  #ifndef FLOORBUTTON_H
4  #define FLOORBUTTON_H
5
6  #include "button.h"  // Button class definition
7
8  class FloorButton : public Button {
9
10 public:
11     FloorButton( int, Elevator & );  // constructor
12    ~FloorButton();                   // destructor
13    void pressButton();               // press the button
14
15 private:
16    const int floorNumber;  // button's floor number
17
18 }; // end class FloorButton
19
20 #endif // FLOORBUTTON_H
```

Fig. 9.37 **FloorButton** class header file.

```
1  // Fig. 9.38: floorButton.cpp
2  // Member-function definitions for class FloorButton.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include "floorButton.h"
9  #include "elevator.h"
```

Fig. 9.38 **FloorButton** class member-function definitions. (Part 1 of 2.)

```

// constructor
FloorButton::FloorButton( int floor, Elevator &elevatorHandle )
   : Button( elevatorHandle ),
     floorNumber( floor )
{
   cout << "floor " << floorNumber << " button constructed"
        << endl;

} // end FloorButton constructor

// destructor
FloorButton::~FloorButton()
{
   cout << "floor " << floorNumber << " button destructed"
        << endl;

} // end ~FloorButton destructor

// press the button
void FloorButton::pressButton()
{
   Button::pressButton();
   cout << "floor " << floorNumber
        << " button summons elevator" << endl;

   // call elevator to this floor
   elevatorRef.summonElevator( floorNumber );

} // end function pressButton
```

Fig. 9.38 **FloorButton** class member-function definitions. (Part 2 of 2.)

We now have completed the implementation for the elevator-simulator case study that we have been developing since Chapter 2. One significant architectural opportunity remains. You might have noticed that classes **Button**, **Door** and **Light** have much in common. Each of these classes contains a "state" attribute and corresponding "set on" and "set off" operations. Class **Bell** also bears some similarity to these other classes. Object-oriented thinking tells us that we should place commonalities in one or more base classes, from which we should then use inheritance to form appropriate derived classes. We leave this implementation to the reader as an exercise. We suggest that you begin by modifying the class diagram in Fig. 9.32. [*Hint*: **Button**, **Door** and **Light** are essentially "toggle" classes—they each have "state," "set on" and "set off" capabilities; **Bell** is a "thinner" class, with only a single operation and no state.]

We sincerely hope that this elevator simulation case study was a challenging and meaningful experience for you. We employed a carefully developed, incremental object-oriented process to produce a UML-based design for our elevator simulator. From this design, we produced a substantial working C++ implementation using key programming notions, including classes, objects, encapsulation, visibility, composition and inheritance.

In the remaining chapters of the book, we present many additional key C++ technologies. We would be grateful if you would take a moment to send your comments, criticisms and suggestions for improving this case study to us at **`deitel@deitel.com`**.

SUMMARY

- Software reuse reduces program-development time.
- The direct base class of a derived class is the base class from which the derived class inherits (specified by the class name to the right of the **`:`** in the first line of a class definition). An indirect base class of a derived class is two or more levels up the class hierarchy from that derived class.
- With single inheritance, a class is derived from one base class. With multiple inheritance, a class is derived from more than one direct base class.
- A derived class can include its own data members and member functions, so a derived class is often larger than its base class.
- A derived class is more specific than its base class and represents a smaller group of objects.
- Every object of a derived class is also an object of that class's base class. However, a base-class object is not an object of that class's derived classes.
- Derived-class member functions can access protected base-class members directly.
- An "is-a" relationship represents inheritance. In an "is-a" relationship, an object of a derived class also can be treated as an object of its base class.
- A "has-a" relationship represents composition. In a "has-a" relationship, a class object contains one or more objects of other classes as members.
- A derived class cannot access the private members of its base class directly; allowing this would violate the encapsulation of the base class. A derived class can, however, access the public and protected members of its base class directly.
- When a base-class member function is inappropriate for a derived class, that member function can be redefined in the derived class with an appropriate implementation.
- Single-inheritance relationships form tree-like hierarchical structures—a base class exists in a hierarchical relationship with its derived classes.
- It is possible to treat base-class objects and derived-class objects similarly; the commonality shared between the object types is expressed in the data members and member functions of the base class.
- A base class's public members are accessible anywhere that the program has a handle to an object of that base class or to an object of one of that base class's derived classes.
- A base class's private members are accessible only within the definition of that base class or from friends of that class.
- A base class's protected members have an intermediate level of protection between public and private access. A base class's protected members can be accessed by members and friends of that base class and by members and friends of any classes derived from that base class.
- Unfortunately, protected data members often yield two major problems. First, the derived-class object does not have to use a *set* function to change the value of the base-class's protected data. Second, derived-class member functions are more likely to depend on base-class implementation details.
- When a derived-class member function redefines a base-class member function, the base-class member function can be accessed from the derived class by preceding the base-class member function name with the base-class name and the scope resolution operator (**`::`**).

- When an object of a derived class is instantiated, the base class's constructor is called immediately (either explicitly or implicitly) to initialize the base-class data members in the derived-class object (before the derived-class data members are initialized).
- Declaring data members **`private`**, while providing non-private member functions to manipulate and perform validation checking on this data, enforces good software engineering.
- When a derived-class object is destroyed, the destructors are called in the reverse order of the constructors—first the derived-class destructor is called, then the base-class destructor is called.
- When deriving a class from a base class, the base class may be declared as either **`public`**, **`protected`** or **`private`**.
- When deriving a class from a **`public`** base class, **`public`** members of the base class become **`public`** members of the derived class, and **`protected`** members of the base class become **`protected`** members of the derived class.
- When deriving a class from a **`protected`** base class, **`public`** and **`protected`** members of the base class become **`protected`** members of the derived class.
- When deriving a class from a **`private`** base class, **`public`** and **`protected`** members of the base class become **`private`** members of the derived class.
- "Knows a" relationships are examples of objects containing pointers or references to other objects so they can be aware of those objects.

TERMINOLOGY

abstraction
association
base class
base-class constructor
base-class default constructor
base-class destructor
base-class initializer
class hierarchy
composition
customize software
derived class
derived-class constructor
derived-class destructor
direct base class
`friend` of a base class
`friend` of a derived class
has-a relationship
hierarchical relationship
indirect base class
infinite recursion error
inheritance
is-a relationship
knows-a relationship
member access control
member class
member object
multiple inheritance
object-oriented programming (OOP)
`private` base class
`private` inheritance
`protected` base class
`protected` inheritance
`protected` keyword
`protected` member of a class
`public` base class
`public` inheritance
redefine a base-class member function
single inheritance
software reusability
uses-a relationship

SELF-REVIEW EXERCISES

9.1 Fill in the blanks in each of the following statements:

a) __________ is a form of software reusability in which new classes absorb the data and behaviors of existing classes and embellish these classes with new capabilities.

b) A base class's __________ members can be accessed only in the base-class definition or in derived-class definitions.

c) In a(n) __________ relationship, an object of a derived class also can be treated as an object of its base class.
d) In a(n) __________ relationship, a class object has one or more objects of other classes as members.
e) In single inheritance, a class exists in a(n) __________ relationship with its derived classes.
f) A base class's __________ members are accessible anywhere that the program has a handle to an object of that base class or to an object of one of its derived classes.
g) A base class's **`protected`** access members have a level of protection between those of **`public`** and __________ access.
h) C++ provides for __________, which allows a derived class to inherit from many base classes, even if these base classes are unrelated.
i) When an object of a derived class is instantiated, the base class's __________ is called implicitly or explicitly to do any necessary initialization of the base-class data members in the derived-class object.
j) When deriving a class from a base class with **`public`** inheritance, **`public`** members of the base class become __________ members of the derived class, and **`protected`** members of the base class become __________ members of the derived class.
k) When deriving a class from a base class with **`protected`** inheritance, **`public`** members of the base class become __________ members of the derived class, and **`protected`** members of the base class become __________ members of the derived class.

9.2 State whether each of the following is *true* or *false*. If *false*, explain why.
a) It is possible to treat base-class objects and derived-class objects similarly.
b) Base-class constructors are not inherited by derived classes.
c) A "has-a" relationship is implemented via inheritance.
d) A **`Car`** class has an "is a" relationship with its **`SteeringWheel`** and **`Brakes`**.
e) Inheritance encourages the reuse of proven high-quality software.

ANSWERS TO SELF-REVIEW EXERCISES

9.1 a) Inheritance. b) **`protected`**. c) "is-a" or inheritance. d) "has-a" or composition or aggregation. e) hierarchical. f) **`public`**. g) **`private`**. h) multiple inheritance. i) constructor. j) **`public`**, **`protected`**. k) **`protected`**, **`protected`**.

9.2 a) True. b) True. c) False. A "has-a" relationship is implemented via composition. An "is-a" relationship is implemented via inheritance. d) False. This is an example of a "has–a" relationship. Class **`Car`** has an "is–a" relationship with class **`Vehicle`**. e) True.

EXERCISES

9.3 Many programs written with inheritance could be written with composition instead, and vice versa. Rewrite classes **`Point3`**, **`Circle4`** and **`Cylinder`** to use composition, rather than inheritance. After you do this, assess the relative merits of the two approaches for the **`Point3`**, **`Circle4`**, **`Cylinder`** problem, as well as for object-oriented programs in general. Which approach is more natural, why?

9.4 Some programmers prefer not to use **`protected`** access because it breaks the encapsulation of the base class. Discuss the relative merits of using **`protected`** access vs. using **`private`** access in base classes.

9.5 Rewrite the case study in Section 9.5 as a **`Point`**, **`Square`**, **`Cube`** program. Do this two ways—once via inheritance and once via composition.

9.6 Write an inheritance hierarchy for class **`Quadrilateral`**, **`Trapezoid`**, **`Parallelogram`**, **`Rectangle`** and **`Square`**. Use **`Quadrilateral`** as the base class of the hierarchy. Make the hierarchy as deep (i.e., as many levels) as possible. The **`private`** data of **`Quadrilateral`** should be the x–y coordinate pairs for the four endpoints of the **`Quadrilateral`**.

9.7 Modify classes **`Point3`**, **`Circle4`** and **`Cylinder`** to contain destructors. Then modify the program of Fig. 9.29 to demonstrate the order in which constructors and destructors are invoked in this hierarchy.

9.8 Write down all the shapes you can think of—both two dimensional and three dimensional—and form those shapes into a shape hierarchy. Your hierarchy should have base class **`Shape`** from which class **`TwoDimensionalShape`** and class **`ThreeDimensionalShape`** are derived. Once you have developed the hierarchy, define each of the classes in the hierarchy. We will use this hierarchy in the exercises of Chapter 10 to process all shapes as objects of base-class **`Shape`**. (This technique, called polymorphism, is the subject of Chapter 10.)