CHAPTER
13
Inheritance and Polymorphism

Objectives

• To develop a derived class from a base class through inheritance (§13.2).
• To enable generic programming by passing objects of derived type to a parameter of a base class type (§13.3).
• To understand constructor and destructor chaining (§13.4).
• To know how to invoke the base class’s constructors with arguments (§13.4).
• To redefine functions in the derived class (§13.5).
• To distinguish differences between redefining and overloading (§13.5).
• To understand polymorphism and dynamic binding using virtual functions (§13.6).
• To distinguish differences between redefining and overriding (§13.6).
• To distinguish differences between static matching and dynamic binding (§13.6).
• To access protected members of the base class from derived classes (§13.7).
• To declare abstract classes that contains pure virtual functions (§13.8).
• To cast an object of a base class type to a derived type using the dynamic_cast operator (§13.9).
13.1 Introduction

Object-oriented programming allows you to derive new classes from existing classes. This is called inheritance. This chapter introduces the concept of inheritance. Specifically, it discusses base classes and derived classes, constructor and destructor chaining, protected data members, virtual functions and polymorphism.

13.2 Base classes and Derived classes

In C++ terminology, a class C1 extended from another class C2 is called a derived class, and C2 is called a base class. A base class is also referred to as a parent class, and a derived class as an extended class or a derived class. A derived class inherits accessible data fields and functions from its base class, and may also add new data fields and functions.

Consider geometric objects. Suppose you want to design the classes to model geometric objects like circles and rectangles. Geometric objects have many common properties and behaviors. They can be drawn in a certain color, filled or unfilled. Thus a general class GeometricObject can be used to model all geometric objects. This class contains the properties color and filled and their appropriate get and set functions. Assume that this class also contains the toString() function, which returns a string representation for the object. Since a circle is a special type of geometric object, it shares common properties and functions with other geometric objects. Thus it makes sense to define the Circle class that extends the GeometricObject class. Likewise, Rectangle can also be declared as a derived class of GeometricObject. Figure 13.1 shows the relationship among these classes. An arrow pointing to the base class is used to denote the inheritance relationship between the two classes involved.

<PD: UML Class Diagram>
The GeometricObject class is the base class for Circle and Rectangle.

The Circle class inherits all accessible data fields and functions from the GeometricObject class. In addition, it has a new data field, radius, and its associated get and set functions. It also contains the getArea(), getPerimeter(), and getDiameter() functions for returning the area, perimeter, and diameter of the circle.

The Rectangle class inherits all accessible data fields and functions from the GeometricObject class. In addition, it has data fields width and height, and its associated get and set functions. It also contains the getArea() and getPerimeter() functions for returning the area and perimeter of the rectangle.

The class declaration for GeometricObject is shown in Listing 13.1. The preprocessor directives in lines 2 and 3 set include guard to prevent multiple declarations. The C++ string class header is included in line 3 to support the use of the string class in GeometricObject. The isFilled()
function is the accessor for the filled data field. Since this data field is the bool type, the accessor function is named `isFilled()` by convention.

The `GeometricObject` class is implemented in Listings 13.2. The `toString` function (lines 29-33) returns a string that describes the object. The string operator `+` is used to concatenate two strings and returns a new `string` object.
#include "GeometricObject.h"

GeometricObject::GeometricObject()
{
    color = "white";
    filled = false;
}

GeometricObject::GeometricObject(string color, bool filled)
{
    this->color = color;
    this->filled = filled;
}

string GeometricObject::getColor()
{
    return color;
}

void GeometricObject::setColor(string color)
{
    this->color = color;
}

bool GeometricObject::isFilled()
{
    return filled;
}

void GeometricObject::setFilled(bool filled)
{
    this->filled = filled;
}

string GeometricObject::toString()
{
    return "Geometric object color " + color + " filled " + ((filled) ? "true" : "false");
}

The class declaration for Circle is shown in Listing 13.3. Line 3 declares that the Circle class is derived from the base class GeometricObject. The syntax

class Circle: public GeometricObject
tells the compiler that the class is derived from the base class. So, all public members in GeometricObject are inherited in Circle.

Listing 13.3 DerivedCircle.h (Circle Header)

***PD: Please add line numbers in the following code***
***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.***

.Side Remark line 1: include guard
.Side Remark line 5: extends GeometricObject
.Side Remark line 7: public members
.Side Remark line 17: private members

```c++
#ifndef DERIVEDCIRCLE_H
#define DERIVEDCIRCLE_H
#include "GeometricObject.h"

class Circle: public GeometricObject
{
public:
    Circle();
    Circle(double);
    Circle(double radius, string color, bool filled);
    double getRadius();
    void setRadius(double);
    double getArea();
    double getPerimeter();
    double getDiameter();

private:
    double radius;
}; // Must place semicolon here

#endif
```

The Circle class is implemented in Listings 13.4. Note that the constructor Circle(double radius, string color, bool filled) will be implemented in the next section.

Listing 13.4 DerivedCircle.cpp (Circle Implementation)

***PD: Please add line numbers in the following code***
***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.***

.Side Remark line 1: Circle header
.Side Remark line 4: no-arg constructor
.Side Remark line 10: constructor
.Side Remark line 16: getRadius
.Side Remark line 22: setRadius
.Side Remark line 28: getArea
.Side Remark line 34: getPerimeter
.Side Remark line 40: getDiameter

#include "DerivedCircle.h"

// Construct a default circle object
Circle::Circle()
{
    radius = 1;
}

// Construct a circle object with specified radius
Circle::Circle(double radius)
{
    this->radius = radius;
}

// Return the radius of this circle
double Circle::getRadius()
{
    return radius;
}

// Set a new radius
void Circle::setRadius(double radius)
{
    this->radius = (radius >= 0) ? radius : 0;
}

// Return the area of this circle
double Circle::getArea()
{
    return radius * radius * 3.14159;
}

// Return the perimeter of this circle
double Circle::getPerimeter()
{
    return 2 * radius * 3.14159;
}

// Return the diameter of this circle
double Circle::getDiameter()
{
    return 2 * radius;
}

The class declaration for Rectangle is shown in Listing 13.5. Line 3 declares that the Rectangle class is derived from the base class GeometricObject. So, all public members in GeometricObject are inherited in Rectangle.

Listing 13.5 Rectangle.h (Rectangle Header)

***PD: Please add line numbers in the following code***
***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.***

480
The Rectangle class is implemented in Listings 13.6. Note that the constructor Rectangle(double width, double height, string color, bool filled) will be implemented in the next section.

Listing 13.6 Rectangle.cpp (Rectangle Implementation)

```cpp
#include "Rectangle.h"

// Construct a default rectangle object
Rectangle::Rectangle()
```

width = 1;
height = 1;

// Construct a rectangle object with specified width and height
Rectangle::Rectangle(double width, double height)
{
    this->width = width;
    this->height = height;
}

// Return the width of this rectangle
double Rectangle::getWidth()
{
    return width;
}

// Set a new radius
void Rectangle::setWidth(double width)
{
    this->width = (width >= 0) ? width : 0;
}

// Return the height of this rectangle
double Rectangle::getHeight()
{
    return height;
}

// Set a new height
void Rectangle::setHeight(double height)
{
    this->height = (height >= 0) ? height : 0;
}

// Return the area of this rectangle
double Rectangle::getArea()
{
    return width * height;
}

// Return the perimeter of this rectangle
double Rectangle::getPerimeter()
{
    return 2 * (width + height);
}

Listing 13.7 gives a test program that uses these three classes GeometricObject, Circle, and Rectangles.

Listing 13.7 TestGeometricObject.cpp (Test Inheritance)

***PD: Please add line numbers in the following code***

***Layout: Please layout exactly. Don’t skip the space. This
is true for all source code in the book. Thanks, AU.

#include "GeometricObject.h"
#include "DerivedCircle.h"
#include "Rectangle.h"
#include <iostream>
using namespace std;

int main()
{
    GeometricObject shape;
    shape.setColor("red");
    shape.setFilled(true);
    cout << shape.toString() << endl;

    Circle circle(5);
    circle.setColor("black");
    circle.setFilled(false);
    cout << " Circle radius: " << circle.getRadius() << " area: " << circle.getArea() << " perimeter: " << circle.getPerimeter() << endl;
    cout << circle.toString() << endl;

    Rectangle rectangle(2, 3);
    rectangle.setColor("orange");
    rectangle.setFilled(true);
    cout << " Rectangle width: " << rectangle.getWidth() << " height: " << rectangle.getHeight() << " area: " << rectangle.getArea() << " perimeter: " << rectangle.getPerimeter() << endl;
    cout << rectangle.toString() << endl;

    return 0;
}

<Output>
Geometric object color red filled true
 Circle radius: 5 area: 78.5397 perimeter: 31.4159
Geometric object color black filled false
 Rectangle width: 2 height: 3 area: 6 perimeter: 10
Geometric object color orange filled true
<End Output>

The program creates a GeometricObject and invokes its functions setColor, setFilled, and toString in lines 9-12.

The program creates a Circle object and invokes its
functions `setColor`, `setFilled`, `getRadius`, `getArea`, `getPerimeter`, and `toString` in lines 14-20. Note that the `setColor` and `setFilled` functions are defined in the `GeometricObject` class and inherited in the `Circle` class.

The program creates a `Rectangle` object and invokes its functions `setColor`, `setFilled`, `getRadius`, `getArea`, `getPerimeter`, and `toString` in lines 22-29. Note that the `setColor` and `setFilled` functions are defined in the `GeometricObject` class and inherited in the `Rectangle` class.

**CAUTION**

*<side remark: no blind extension>*

Inheritance is used to model the *is-a* relationship. Do not blindly extend a class just for the sake of reusing functions. For example, it makes no sense for a `Tree` class to extend a `Person` class. A derived class and its base class must have the *is-a* relationship.

### 13.3 Generic Programming

An object of a derived class can be used wherever an object of a based type is required. This enables a function to be used generically for a wide range of object arguments. This is known as *generic programming*. If a function’s parameter type is a base class (e.g., `GeometricObject`), you may pass an object to this function of any of the parameter’s derived classes (e.g., `Circle` or `Rectangle`).

For example, if you declare a function as follows:

```cpp
void displayGeometricObject(GeometricObject shape)
{
    cout << shape.toString() << endl;
}
```

The parameter type is `GeometricObject`. You can invoke this function in the following code:

```cpp
displayGeometricObject(GeometricObject("black", true));
displayGeometricObject(Circle(5));
displayGeometricObject(Rectangle(2, 3));
```

Each statement creates an anonymous object and passes it to invoke `displayGeometricObject`. Since `Circle` and `Rectangle` are derived from `GeometricObject`, you can pass a `Circle` object or a `Rectangle` object to the `GeometricObject` parameter type in the `displayGeometricObject` function.

### 13.4 Constructors and Destructors

484
A derived class inherits accessible data fields and functions from its base class. Does it inherit constructors or destructors? Can base class constructors and destructors be invoked from derived classes? This section addresses these questions and their ramification.

13.4.1 Calling Base Class Constructors

A constructor is used to construct an instance of a class. Unlike data fields and functions, the constructors of a base class are not inherited in the derived class. They can only be invoked from the constructors of the derived classes to initialize the data fields in the base class. The syntax to invoke it is as follows:

```
DerivedClass(parameterList): BaseClass()
// Perform initialization
```

Or

```
DerivedClass(parameterList): BaseClass(argumentList)
// Perform initialization
```

The former invokes the no-arg constructor of its base class, and the latter invokes the base class constructor with the specified arguments.

You can only invoke the base class’s constructor in the class implementation, not in the class declaration. For example, you can now add the following constructor in the Circle class implementation:

```cpp
// Construct a circle object with specified radius, color and filled
Circle::Circle(double radius, string color, bool filled)
    : GeometricObject(color, filled)
    // this->radius = radius;
```

This implements the Circle class’s constructor to invoke the GeometricObject class’s constructor with the specified color and filled values.

**NOTE**
A constructor in a derived class must always
invoke a constructor in its base class. If a base constructor is not invoked explicitly, the base class’s no-arg constructor is invoked by default. For example,

\[
\text{public Circle()}
\begin{aligned}
\{ \\
\text{radius} = 1;
\}
\end{aligned}
\text{ is equivalent to}
\begin{aligned}
\text{public Circle(): GeometricObject()}
\{ \\
\text{radius} = 1;
\}
\end{aligned}
\]

\[
\text{public Circle(double radius)}
\begin{aligned}
\{ \\
\text{this->radius} = \text{radius};
\}
\end{aligned}
\text{ is equivalent to}
\begin{aligned}
\text{public Circle(double radius): GeometricObject()}
\{ \\
\text{this->radius} = \text{radius};
\}
\end{aligned}
\]

13.4.2 Constructor and Destructor Chaining
Constructing an instance of a class invokes the constructors of all the base classes along the inheritance chain. A base class’s constructor is called before the derived class’s constructor. Conversely, the destructors are automatically invoked in reverse order, with the derived class’s destructor invoked first. This is called constructor and destructor chaining.

Consider the following code:

Listing 13.8 ChainingDemo.cpp (Constructor and Destructor Chaining)

```cpp
#include <iostream>

using namespace std;

class Person
{
public:
  Person()
  {
    cout << "Person's constructor is invoked" << endl;
  }

  ~Person()
  {
    cout << "Person's destructor is invoked" << endl;
  }
};

class Employee: public Person
```

**PD: Please add line numbers in the following code***
**Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.**

<Side Remark line 4: Person class>
<Side Remark line 18: Employee class>
<Side Remark line 32: Faculty class>
<Side Remark line 48: create a Faculty>
The program creates an instance of Faculty in line 48. Since Faculty is derived from Employee and Employee is derived from Person, Person’s constructor is called first, then Employee’s, and finally Faculty’s. When the program exits, the Faculty object is destroyed. So the Faculty’s destructor is called, then Employee’s, and finally Person’s.

CAUTION

.Side Remark: no-arg constructor

If a class is designed to be extended, it is better to provide a no-arg constructor to avoid
programming errors. Consider the following code:

```cpp
***PD: Please add line numbers in the following code***

```class Fruit{
|  
| public:
|     Fruit(int id)
|     {}  
|}
```

```cpp
class Apple: public Fruit{
|  
| public:
|     Apple()
|     {}  
|}
```

Since no constructor is explicitly defined in Apple, Apple’s default no-arg constructor is declared implicitly. Since Apple is a derived class of Fruit, Apple’s default constructor automatically invokes Fruit’s no-arg constructor. However, Fruit does not have a no-arg constructor because Fruit has an explicit constructor defined. Therefore, the program cannot be compiled.

***End of CAUTION***

13.5 Redefining Functions
The toString() function is defined in the GeometricObject class to return a string description of a GeometricObject. You can redefine this function in the Circle and Rectangle classes to return a more specific description that is tailored to a Circle or a Rectangle object.

To redefine a base class’s function in the derived class, you need to add the function’s prototype in the derived class’s header file, and provide a new implementation for the function in the derived class’s implementation file.

For example, to redefine toString in Circle, add the function’s prototype to the Circle declaration and implement it as follows.

```cpp
// Redefine the toString function
string Circle::toString()
{
    return "Circle object";
}
```
So the following code

```cpp
***PD: Please add line numbers in the following code***
***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.

.Side Remark line 1: create GeometricObject
.Side Remark line 2: invoke toString
.Side Remark line 4: create Circle
.Side Remark line 5: invoke toString

GeometricObject shape;
cout << "shape.toString() returns " << shape.toString() << endl;

Circle circle(5);
cout << "circle.toString() returns " << circle.toString() << endl;
```
displays:

```cpp
shape.toString() returns Geometric object color white filled false
circle.toString() returns Circle object
```

The code creates a `GeometricObject` in line 1. The `toString` function defined in `GeometricObject` is invoked in line 2, since `shape`’s type is `GeometricObject`.

The code creates a `Circle` object in line 3. The `toString` function defined in `Circle` is invoked in line 5, since `circle`’s type is `Circle`.

.Side Remark: Invoke function in the base
If you wish to invoke the `toString` function defined in the `GeometricObject` class on the calling object `circle`, use the scope resolution operator (::) with the base class name as follows:

```cpp
circle.GeometricObject::toString()
```

NOTE
.Side Remark: redefining vs. overloading
You have learned about overloading functions in §5.7, “Overloading Functions.” Overloading a function is a way to provide more than one function with the same name but with different signatures to distinguish them. To redefine a function, the function must be defined in the derived class using the same signature and same return type as in its base class.
13.6 Polymorphism and Virtual Functions
Before introducing polymorphism, let us begin with an example in Listing 13.9 to demonstrate the need for polymorphism.

Listing 13.9 WhyPolymorphismDemo.cpp (Why Need Polymorphism?)

***PD: Please add line numbers in the following code***
***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.***

```cpp
#include <iostream>
using namespace std;

class C
{
public:
    string toString()
    {
        return "class C";
    }
};

class B: public C
{
    string toString()
    {
        return "class B";
    }
};

class A: public B
{
    string toString()
    {
        return "class A";
    }
};

displayObject(C x)
```

490
```
cout << x.toString() << endl;
}

int main()
{
    displayObject(A());
    displayObject(B());
    displayObject(C());
    return 0;
}

<Output>
class C
class C
class C
<End Output>

The program declares three classes A, B, and C. A is derived from B, and B is derived from C. The displayObject function is invoked in lines 36–38 by passing anonymous objects A(), B(), and C().

The argument of the displayObject function (lines 29–32) is an object of the C class or C’s derived classes. The function invokes the toString() function on the object (line 31). The toString() function defined in class C is invoked. So, the output is same for the three function calls.

It would be nice to invoke the toString() function defined in A when invoking displayObject(A()), the toString() function defined in B when invoking displayObject(B()), and the toString() function defined in C when invoking displayObject(C()). This can be done by modifying Listing 13.9 using virtual functions and pointer variables. The new program is shown in Listing 13.10.

Listing 13.10 PolymorphismDemo.cpp (Why Need Polymorphism?)

***PD: Please add line numbers in the following code***
***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.***
<Side Remark line 4: class C>
<Side Remark line 7: virtual function toString>
<Side Remark line 13: class B>
<Side Remark line 15: override toString function>
<Side Remark line 21: class A>
<Side Remark line 25: override toString function>
<Side Remark line 29: displayObject>
<Side Remark line 31: invoke toString>
<Side Remark line 36: invoke displayObject>
<Side Remark line 37: invoke displayObject>
<Side Remark line 38: invoke displayObject>
```cpp
#include <iostream>
using namespace std;

class C
{
public:
    virtual string toString()
    {
        return "class C";
    }
};

class B: public C
{
    string toString()
    {
        return "class B";
    }
};

class A: public B
{
    string toString()
    {
        return "class A";
    }
};

void displayObject(C *p)
{
    cout << p->toString() << endl;
}

int main()
{
    displayObject(&A());
    displayObject(&B());
    displayObject(&C());
    return 0;
}
```

<Output>
class A
class B
class C
<End Output>

<Side Remark: virtual>
<Side Remark: pointer>
<Side Remark: dynamic binding>
<Side Remark: polymorphism>
Line 7 defines **toString** to be a virtual function using the **virtual** keyword.
keyword `virtual`. Line 29 defines a pointer parameter `p` whose type is class `C`. When invoking `displayObject(&A())` in line 36, the address of object `A()` is passed to `p`. When invoking `p->toString()` in line 31, C++ dynamically determines which `toString()` to use. Since `p` points to an object of class `A`, the `toString` function defined in class `A` is invoked. The capability of determining which function to invoke at runtime is known as `dynamic binding`. It is also commonly known as polymorphism (from a Greek word meaning “many forms”) because one function has many implementations.

**NOTE**

*Side Remark: redefining vs. overriding*

In C++, redefining a virtual function in a derived class is called *overriding a function*.

To enable dynamic binding for a function, you need to two things:

- The function must be declared `virtual` in the base class.
- The variable that references the object for the function must contain the address of the object.

Listing 13.10 passes the address of the object to a pointer (lines 29-40); alternatively, you can rewrite lines 29-40 by passing the object to a base class reference, as follows:

```cpp
void displayObject(C &p)
{
    cout << p.toString() << endl;
}

int main()
{
    displayObject(A());
    displayObject(B());
    displayObject(C());
    return 0;
}
```

**NOTE**

*Side Remark: virtual*

If a function is defined `virtual` in a base class, it is automatically `virtual` in all its derived classes. It is not necessary to add the keyword `virtual` in the function declaration in the derived class.

**NOTE**

*Side Remark: static matching vs. dynamic binding*
Matching a function signature and binding a function implementation are two separate issues. The *declared type* of the variable decides which function to match at compile time. The compiler finds a matching function according to parameter type, number of parameters, and order of the parameters at compile time. A virtual function may be implemented in several derived classes. C++ dynamically binds the implementation of the function at runtime, decided by the *actual class* of the object referenced by the variable.

**TIP**

*Side Remark: use virtual functions?*

If a function defined in a base class needs to be redefined in its derived classes, you should declare it virtual to avoid confusions and mistakes. On the other hand, if a function will not be redefined, it is more efficient without declaring it virtual, because it takes more time and system resource to bind virtual functions dynamically at runtime.

### 13.7 The protected Keyword

So far you have used the private and public keywords to specify whether data fields and functions can be accessed from the outside of the class. Private members can only be accessed from the inside of the class and public members can be accessed from any other classes. A protected data field or a protected function in a base class can be accessed by name in its derived classes.

The keywords *private*, *protected*, and *public* are known as visibility or accessibility keywords because they specify how class and class members are accessed. The visibility of these modifiers increases in this order:

```
Visibility increases

private, protected, public
```

Listing 13.11 demonstrates how to use protected keywords.

**Listing 13.11 VisibilityDemo.cpp (private, protected, public)**

***PD: Please add line numbers in the following code***

***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.***

*Side Remark line 7: public*

*Side Remark line 9: protected*

*Side Remark line 12: private*
```cpp
#include <iostream>
using namespace std;

class B
{
public:
    int i;

protected:
    int j;

private:
    int k;
};

class A: public B
{
public:
    void display()
    {
        cout << i << endl; // Fine, can access it
        cout << j << endl; // Fine, can access it
        cout << k << endl; // Wrong, cannot access it
    }

int main()
{
    A a;
    cout << a.i << endl; // Fine, can access it
    cout << a.j << endl; // Wrong, cannot access it
    cout << a.k << endl; // Wrong, cannot access it
    return 0;
}
```

Since A is derived from B and j is protected, j can be accessed from class A in line 22. Since k is private, k cannot be accessed from class A.

Since i is public, i can be accessed from a.i in line 30. Since j and k are not public, they cannot be accessed from the object a in lines 31-32.

13.8 Abstract Classes and Pure Virtual Functions

Side Remark: abstract class

In the inheritance hierarchy, classes become more specific and concrete with each new derived class. If you move from a derived class back up to its parent and ancestor classes, the classes become more general and less specific. Class design should ensure that a base class contains common features of its derived classes. Sometimes a base class is
so abstract that it cannot have any specific instances. Such a class is referred to as an abstract class.

.Side Remark: abstract function
GeometricObject was declared as the base class for Circle and Rectangle in §13.2, “Base classes and Derived classes.” GeometricObject models common features of geometric objects. Both Circle and Rectangle contain the getArea() and getPerimeter() functions for computing the area and perimeter of a circle and a rectangle. Since you can compute areas and perimeters for all geometric objects, it is better to declare the getArea() and getPerimeter() functions in the GeometricObject class. However, these functions cannot be implemented in the GeometricObject class because their implementation is dependent on the specific type of geometric object. Such functions are referred to as abstract functions. After you declare the functions in GeometricObject, GeometricObject becomes an abstract class. The new GeometricObject class is shown in Figure 13.2. In UML graphic notation, the names of abstract classes and their abstract functions are italicized, as shown in Figure 13.2.

.PD: UML Class Diagram>
The new `GeometricObject` class contains abstract functions.

**Side Remark: abstract function**

In C++, abstract functions are called pure virtual functions. A class that contains pure virtual functions becomes an abstract class. A pure virtual function is declared in a manner similar to the following:

```cpp
virtual double getArea() = 0;
```

The `= 0` notation indicates that `getArea` is a pure virtual function. A pure virtual function does not have a body or implementation in the base class.

Listing 13.12 defines the new abstract `GeometricObject` class with two pure virtual functions in lines 16-17.

**Listing 13.12 AbstractGeometricObject.h (Abstract Class Declaration)**

***PD: Please add line numbers in the following code***

***Layout: Please layout exactly. Don’t skip the space. This***
is true for all source code in the book. Thanks, AU.

GeometricObject is just like a regular class except that you cannot create objects from GeometricObject because it is an abstract class. If you attempt to create an object from GeometricObject, the compiler will generate an error.

Listing 13.13 gives an implementation of the GeometricObject class.

Listing 13.13 AbstractGeometricObject.cpp (Abstract Class Implementation)

**PD: Please add line numbers in the following code**

**Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.**

Listing line 1: include header

```cpp
#include "AbstractGeometricObject.h"
```

// Same as lines 3-39 in GeometricObject.cpp in Listing 13.2

Listings 13.14, 13.15, and 13.16 show the files for the new Circle and Rectangle classes derived from the abstract GeometricObject.

Listing 13.14 DerivedCircle2.h (New Derived Circle Declaration)

**PD: Please add line numbers in the following code**
You may be wondering whether the abstract functions `getArea` and `getPerimeter` should be removed from the `GeometricObject`
class. The following example shows the benefits of retaining them in the GeometricObject class.

This example in Listing 13.18 presents a program that creates two geometric objects, a circle and a rectangle, invokes the equalArea function to check whether the two objects have equal areas, and invokes the displayGeometricObject function to display the objects.

Listing 13.18 TestGeometricObject2.cpp (Using Abstract Classes)

```cpp
#include "AbstractGeometricObject.h"
#include "DerivedCircle2.h"
#include "Rectangle2.h"
#include <iostream>
using namespace std;

// A function for comparing the areas of two geometric objects
bool equalArea(GeometricObject &object1,
               GeometricObject &object2)
{
    return object1.getArea() == object2.getArea();
}

// A function for displaying a geometric object
void displayGeometricObject(GeometricObject &object)
{
    cout << "The area is " << object.getArea() << endl;
    cout << "The perimeter is " << object.getPerimeter() << endl;
}

int main()
{
    Circle circle(5);
    Rectangle rectangle(5, 3);

    cout << "Circle info: " << endl;
    displayGeometricObject(circle);

    cout << "\nRectangle info: " << endl;
    displayGeometricObject(rectangle);

    cout << "\nThe two objects have the same area? " <<
    (equalArea(circle, rectangle) ? "Yes" : "No") << endl;

    return 0;
}
```
The program creates a Circle object and a Rectangle object in lines 23-24.

The pure virtual functions getArea() and getPerimeter() defined in the GeometricObject class are overridden in the Circle class and the Rectangle class.

When invoking displayGeometricObject(circle1) (line 27), the functions getArea and getPerimeter defined in the Circle class are used, and when invoking displayGeometricObject(rectangle) (line 29), the functions getArea and getPerimeter defined in the Rectangle class are used. C++ dynamically determines which of these functions to invoke at runtime, depending on the type of object.

Similarly, when invoking equalArea(object1, object2) (line 33), the getArea function defined in the Circle class is used for object1.getArea(), since object1 is a circle, and the getArea function defined in the Rectangle class is used for object2.getArea(), since object2 is a rectangle.

Note that if the getArea and getPerimeter functions were not defined in GeometricObject, you cannot define the equalArea and displayObject functions in this program. So, you now see the benefits of defining the abstract functions in GeometricObject.

13.9 Dynamic Casting

<Side Remark: dynamic cast operator>

The header for the displayGeometricObject function in Listing 13.18 is

void displayGeometricObject(GeometricObject &object)

Suppose you wish to modify this function to display radius, diameter, area, and perimeter if the object is a circle. How can this be done?

You can use the dynamic_cast operator to cast a parameter of the GeometricObject type into a Circle type, and then invoke the getRadius() and getDiameter() functions defined in the Circle class, as shown in the following code:

***PD: Please add line numbers in the following code***
***Layout: Please layout exactly. Don’t skip the space. This
GeometricObject *p = &object;
Circle *p1 = dynamic_cast<Circle*>(p);

if (p1 != 0)
  
  cout << "The radius is " << p1->getRadius() << endl;
  cout << "The diameter is " << p1->getDiameter() << endl;

Line 1 creates a pointer for the `object` parameter. The `dynamic_cast` operator (line 2) checks whether pointer `p` points to a `Circle` object. If so, the object’s address is assigned to `p1`, otherwise `p1` is 0. If `p1` is not 0, the `getRadius()` and `getDiameter()` functions of the `Circle` object (pointed by `p1`) are invoked in lines 6-7.

NOTE

Assigning a pointer of a derived class type to a pointer of its base class type is called **upcasting** and assigning a pointer of a base class type to a pointer of its derived class type is called **downcasting**. Upcasting can be performed implicitly without using the `dynamic_cast` operator. For example, the following code is correct:

```cpp
GeometricObject *p = new Circle(1);
Circle *p1 = new Circle(2);
p = p1;
```

However, downcasting must be performed explicitly. For example, to assign `p` to `p1`, you have to use

```cpp
p1 = dynamic_cast<Circle*>(p);
```

***END NOTE***

Listing 13.19 gives a complete program that modifies the `displayGeometricObject` function to display information for `Circle` and `Rectangle` objects.

Listing 13.19 DynamicCastingDemo.cpp (Using Dynamic Casting)

***PD: Please add line numbers in the following code***

***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.***

<Side Remark line 1: include header file>
<Side Remark line 14: casting to Circle>
<Side Remark line 15: casting to Rectangle>
#include "AbstractGeometricObject.h"
#include "DerivedCircle2.h"
#include "Rectangle2.h"
#include <iostream>
using namespace std;

// A function for displaying a geometric object
void displayGeometricObject(GeometricObject &object)
{
    cout << "The area is " << object.getArea() << endl;
    cout << "The perimeter is " << object.getPerimeter() << endl;

    GeometricObject *p = &object;
    Circle *p1 = dynamic_cast<Circle*>(p);
    Rectangle *p2 = dynamic_cast<Rectangle*>(p);

    if (p1 != 0)
    {
        cout << "The radius is " << p1->getRadius() << endl;
        cout << "The diameter is " << p1->getDiameter() << endl;
    }

    if (p2 != 0)
    {
        cout << "The width is " << p2->getWidth() << endl;
        cout << "The height is " << p2->getHeight() << endl;
    }
}

int main()
{
    Circle circle(5);
    Rectangle rectangle(5, 3);

    cout << "Circle info: " << endl;
    displayGeometricObject(circle);

    cout << "\nRectangle info: " << endl;
    displayGeometricObject(rectangle);

    return 0;
}

<Output>
Circle info:
The area is 78.5397
The perimeter is 31.4159
The radius is 5
The diameter is 10

Rectangle info:
The area is 15
The perimeter is 16
The width is 5
The program invokes the `displayGeometricObject` function to display a `Circle` object in line 36 and a `Rectangle` object in line 39. The function casts the `object` parameter into a `Circle` pointer `p1` in line 14 and a `Rectangle` pointer `p2` in line 15. If it is a `Circle` object, the object’s `getRadius()` and `getDiameter()` functions are invoked in lines 19-20. If it is a `Rectangle` object, the object’s `getWidth()` and `getHeight()` functions are invoked in lines 25-26.

The function also invokes `GeometricObject`’s `getArea()` and `getPerimeter()` functions in lines 10-11. Since these two functions are defined in the `GeometricObject` class, there is no need to downcast the object parameter to `Circle` or `Rectangle` in order to invoke these two functions.

**TIP**

*Side Remark: typeid operator*

Occasionally, it is useful to obtain the information about the class of the object. You can use the `typeid` operator to return a reference to an object of class `type_info`. For example, you can use the following statement to display the class name for object `x`.

```cpp
string x;
cout << typeid(x).name() << endl;
```

It displays `string`, because `x` is an object of the `string` class.

***END NOTE***

**Key Terms**

***PD: Please place terms in two columns same as in intro5e.***

- abstract class
- abstract function
- base class 288
- constructor chaining 290
- derived class 288
- destructor chaining 290
- downcasting
- dynamic binding 295
- generic programming 296
• inheritance 288
• is-a relationship
• override 291
• polymorphism 295
• protected 301
• pure virtual function
• redefine
• upcasting
• virtual function

Chapter Summary
• You can derive a new class from an existing class. This is known as class inheritance. The new class is called a derived class, child class, or extended class. The existing class is called a base class or parent class.
• An object of a derived class can be used wherever an object of a based type is required. This enables a function to be used generically for a wide range of object arguments. This is known as generic programming.
• A constructor is used to construct an instance of a class. Unlike data fields and functions, the constructors of a base class are not inherited in the derived class. They can only be invoked from the constructors of the derived classes to initialize the data fields in the base class.
• A child class constructor always invokes its base class’s constructor in the class implementation, not in the class declaration. If a base constructor is not invoked explicitly, the base class’s no-arg constructor is invoked by default.
• Constructing an instance of a class invokes the constructors of all the base classes along the inheritance chain. A base class’s constructor is called before the derived class’s constructor. Conversely, the destructors are automatically invoked in reverse order, with the derived class’s destructor invoked first. This is called constructor and destructor chaining.
• A function defined in the base class may be redefined in the derived class. A redefined function must match the signature and return type of the function in the base class.
• A virtual function enables dynamic binding. A virtual function is often redefined in the derived classes. The compiler decides which function implementation to use dynamically at runtime.
• If a function defined in a base class needs to be redefined in its derived classes, you should declare it virtual to avoid confusions and mistakes. On the other hand, if a function will not be redefined, it is more
efficient without declaring it virtual, because it
takes more time and system resource to bind virtual
functions dynamically at runtime.

- A protected data field or a protected function in a
  base class can be accessed by name in its derived
classes.
- A pure virtual function is also called an abstract
  function. If a class contains a pure virtual function,
  the class is called an abstract class. You cannot
  create instances from an abstract class, but abstract
  classes can be used as data types for parameters in a
  function to enable generic programming.
- You can use the `dynamic_cast` operator to cast an object
  of a base class type to a pointer of a derived class
  type in order to invoke the functions defined in the
  derived classes.

**Review Questions**

*Sections 13.2-13.5*

13.1
What is the printout of running the program in (a)? What
problem arises in compiling the program in (b)?

```cpp
#include <iostream>
using namespace std;

class A
{
public:
    A()
    {
        cout <<
            "A's no-arg constructor is invoked";
    }
};

class B: public A
{
};

int main()
{
    B b;
}
```

13.2
True or false? (1) A derived class is a subset of a base
class? (2) When invoking a constructor from a
derived class, its base class’s no-arg constructor
is always invoked? (3) You can override a private
function defined in a base class? (4) You can
override a static function defined in a base
class? (5) Can you override a constructor?

13.3
Identify the problems in the following classes:
**class Circle**

**public:**
- **Circle(double radius);**
- 
- **double getRadius();**
  - return radius;
  -
- **double getArea();**
  - return radius * radius * 3.14159;
  -

**private:**
- double radius;

**class B : Circle**

**public:**
- **B(double radius, double length): Circle(radius)**
  - length = length;
  -

  /** Returns Circle’s getArea * length */
- **double getArea();**
  - return getArea() * length;
  -

**private:**
- double length;

13.4

Explain the difference between function overloading and function overriding.

13.5

Show the output of the following code.

```cpp
#include <iostream>
using namespace std;

class A
1
```
```cpp
public:
    A()
    {
        cout << "A's no-arg constructor is invoked" << endl;
    }

    ~A()
    {
        cout << "A's destructor is invoked" << endl;
    }

class B: public A
{
    public:
    B()
    {
        cout << "B's no-arg constructor is invoked" << endl;
    }

    ~B()
    {
        cout << "B's destructor is invoked" << endl;
    }

    int main()
    {
        B b1;
        B b2;
        return 0;
    }
}
```

**Section 13.6 Polymorphism and Virtual Functions**

13.6

Show the output of the following code.

```cpp
#include <iostream>
using namespace std;

class A
{
public:
    void f()
    {
        cout << "invoke f from A" << endl;
    }

class B: public A
{
public:
    void f()
    {
```
Show the output of the following code.

```cpp
#include <iostream>
using namespace std;

class A
{
public:
    virtual void f()
    {
        cout << "invoke f from A" << endl;
    }
};

class B: public A
{
public:
    void f()
    {
        cout << "invoke f from B" << endl;
    }
};

void p(A a)
{
    a.f();
}

int main()
{
    A a;
    a.f();
    p(a);
    B b;
    b.f();
    p(b);
    return 0;
}
```

509
If you replace `p(A a)` by `p(A &a)`, what will be the output?

13.8
Is declaring virtual functions enough to enable dynamic binding?

13.9
Is it a good practice to declare all functions virtual?

Sections 13.7 The protected Keyword

13.10
If a member is declared private in a class, can it be accessed from other classes? If a member is declared protected in a class, can it be accessed from other classes? If a member is declared public in a class, can it be accessed from other classes?

Sections 13.8 Abstract Classes and Pure Virtual Functions

13.11
How do you declare a pure virtual function?

13.12
What is wrong in the following code?

```cpp
class A
{
public:
    virtual void f() = 0;
};

int main()
{
    A a;
    return 0;
}
```

13.13
Can you compile and run the following code? What will be the output?

```cpp
#include <iostream>
using namespace std;

class A
```
```cpp
// class A
public:
    virtual void f() = 0;
};

class B: public A
{
    public:
        void f()
        {
            cout << "invoke f from B" << endl;
        }
};

class C: public B
{
    public:
        virtual void m() = 0;
};

class D: public C
{
    public:
        virtual void m()
        {
            cout << "invoke m from D" << endl;
        }
};

void p(A &a)
{
    a.f();
}

int main()
{
    D d;
    p(d);
    d.m();
    return 0;
}
```

13.14

The `getArea` and `getPerimeter` methods may be removed from the `GeometricObject` class. What are the benefits of defining `getArea` and `getPerimeter` as abstract methods in the `GeometricObject` class?

13.15

What is upcasting? What is downcasting?
13.16
When do you need to downcast an object from a base class type to a derived class type?

13.17
What will be the value in p1 after the following statements?

```cpp
GeometricObject *p = new Rectangle(2, 3);
Circle *p1 = new Circle(2);
p1 = dynamic_cast<Circle*>(p);
```

Programming Exercises

13.1
(The Triangle class) Design a class named Triangle that extends GeometricObject. The class contains:
- Three double data fields named side1, side2, and side3 with default values 1.0 to denote three sides of the triangle.
- A no-arg constructor that creates a default triangle.
- A constructor that creates a rectangle with the specified side1, side2, and side3.
- The accessor functions for all three data fields.
- A function named getArea() that returns the area of this triangle.
- A function named getPerimeter() that returns the perimeter of this triangle.
- A function named toString() that returns a string description for the triangle.

For the formula to compute the area of a triangle, see Exercise 5.20. The toString() function is implemented as follows:

```cpp
return "Triangle: side1 = " + side1 + " side2 = " + side2 + " side3 = " + side3;
```

Draw the UML diagram that involving the classes Triangle and GeometricObject. Implement the class. Write a test program that creates a Triangle object with sides 1, 1.5, 1, setting color yellow and filled true, and displaying the area, perimeter, color, and whether filled or not.

13.2
(The Person, Student, Employee, Faculty, and Staff classes) Design a class named Person and its two derived classes
named Student and Employee. Make Faculty and Staff derived classes of Employee. A person has a name, address, phone number, and e-mail address. A student has a class status (freshman, sophomore, junior, or senior). Define the status as a constant. An employee has an office, salary, and date-hired. Define a class named MyDate that contains the fields year, month, and day. A faculty member has office hours and a rank. A staff member has a title. Override the toString function in each class to display the class name and the person's name.

Draw the UML diagram for the classes. Implement the classes. Write a test program that creates a Person, Student, Employee, Faculty, and Staff, and invokes their toString() functions.

13.3
(Derived classes of Account) In Exercise 7.3, the Account class was created to model a bank account. An account has the properties account number, balance, and annual interest rate, date created, and functions to deposit and withdraw. Create two derived classes for checking and saving accounts. A checking account has an overdraft limit, but a savings account cannot be overdrawn.

Draw the UML diagram for the classes. Implement the classes. Write a test program that creates objects of Account, SavingsAccount, and CheckingAccount, invokes their toString() functions.

13.4
(Implementing a stack class using inheritance) In Listing 13.8, StackOfInteger is implemented using composition. Create a new stack class that extends vector.

Draw the UML diagram for the classes. Implement it.