starting out with >>>

# C++

## From Control Structures through Objects

EIGHTH EDITION

TONY GADDIS

# STARTING OUT WITH

# C++

## From Control Structures through Objects

**EIGHTH EDITION**

*This page intentionally left blank*

CHAPTER 13

# Introduction to Classes

**TOPICS**



## 13.1 Procedural and Object-Oriented Programming

**CONCEPT:** **Procedural programming is a method of writing software. It is a programming practice centered on the procedures or actions that take place in a program. Object-oriented programming is centered around the object. Objects are created from abstract data types that encapsulate data and functions together.**

There are two common programming methods in practice today: procedural programming and object-oriented programming (or OOP). Up to this chapter, you have learned to write procedural programs.

In a procedural program, you typically have data stored in a collection of variables and/or structures, coupled with a set of functions that perform operations on the data. The data and the functions are separate entities. For example, in a program that works with the geometry of a rectangle you might have the variables in Table 13-1:

**Table 13-1**

| Variable Definition | Description |
|---|---|
| `double width;` | Holds the rectangle's width |
| `double length;` | Holds the rectangle's length |

In addition to the variables listed in Table 13-1, you might also have the functions listed in Table 13-2:

**Table 13-2**

| Function Name | Description |
|---|---|
| `setData()` | Stores values in `width` and `length` |
| `displayWidth()` | Displays the rectangle's width |
| `displayLength()` | Displays the rectangle's length |
| `displayArea()` | Displays the rectangle's area |

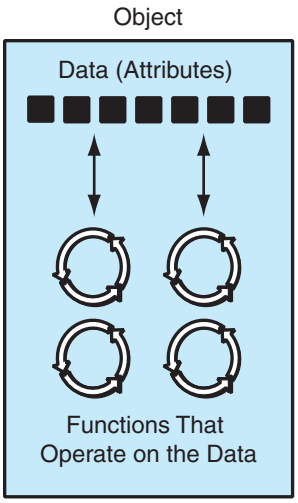
Usually the variables and data structures in a procedural program are passed to the functions that perform the desired operations. As you might imagine, the focus of procedural programming is on creating the functions that operate on the program's data.

Procedural programming has worked well for software developers for many years. However, as programs become larger and more complex, the separation of a program's data and the code that operates on the data can lead to problems. For example, the data in a procedural program are stored in variables, as well as more complex structures that are created from variables. The procedures that operate on the data must be designed with those variables and data structures in mind. But, what happens if the format of the data is altered? Quite often, a program's specifications change, resulting in redesigned data structures. When the structure of the data changes, the code that operates on the data must also change to accept the new format. This results in additional work for programmers and a greater opportunity for bugs to appear in the code.

This problem has helped influence the shift from procedural programming to object-oriented programming (OOP). Whereas procedural programming is centered on creating procedures or functions, object-oriented programming is centered on creating objects. An *object* is a software entity that contains both data and procedures. The data that are contained in an object are known as the object's *attributes*. The procedures that an object performs are called *member functions*. The object is, conceptually, a self-contained unit consisting of attributes (data) and procedures (functions). This is illustrated in Figure 13-1.

OOP addresses the problems that can result from the separation of code and data through encapsulation and data hiding. *Encapsulation* refers to the combining of data and code into a single object. *Data hiding* refers to an object's ability to hide its data from code that

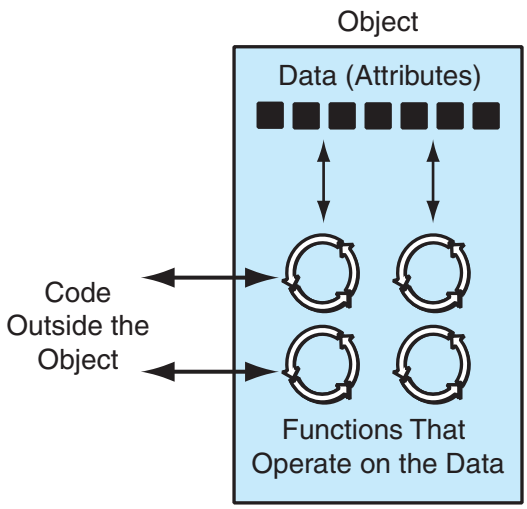
**Figure 13-1**




**NOTE:** In other programming languages, the procedures that an object performs are often called *methods*.

is outside the object. Only the object's member functions may directly access and make changes to the object's data. An object typically hides its data, but allows outside code to access its member functions. As shown in Figure 13-2, the object's member functions provide programming statements outside the object with indirect access to the object's data.

**Figure 13-2**



When an object's internal data are hidden from outside code, and access to that data is restricted to the object's member functions, the data are protected from accidental corruption. In addition, the programming code outside the object does not need to know about the format or internal structure of the object's data. The code only needs to interact with the object's functions. When a programmer changes the structure of an object's internal data, he or she also modifies the object's member functions so they may properly operate on the data. The way in which outside code interacts with the member functions, however, does not change.

An everyday example of object-oriented technology is the automobile. It has a rather simple interface that consists of an ignition switch, steering wheel, gas pedal, brake pedal, and a gear shift. Vehicles with manual transmissions also provide a clutch pedal. If you want to drive an automobile (to become its user), you only have to learn to operate these elements of its interface. To start the motor, you simply turn the key in the ignition switch. What happens internally is irrelevant to the user. If you want to steer the auto to the left, you rotate the steering wheel left. The movements of all the linkages connecting the steering wheel to the front tires occur transparently.

Because automobiles have simple user interfaces, they can be driven by people who have no mechanical knowledge. This is good for the makers of automobiles because it means more people are likely to become customers. It's good for the users of automobiles because they can learn just a few simple procedures and operate almost any vehicle.

These are also valid concerns in software development. A real-world program is rarely written by only one person. Even the programs you have created so far weren't written entirely by you. If you incorporated C++ library functions, or objects like `cin` and `cout`, you used code written by someone else. In the world of professional software development, programmers commonly work in teams, buy and sell their code, and collaborate on projects. With OOP, programmers can create objects with powerful engines tucked away "under the hood," protected by simple interfaces that safeguard the object's algorithms.

## Object Reusability

In addition to solving the problems of code/data separation, the use of OOP has also been encouraged by the trend of *object reusability*. An object is not a stand-alone program, but is used by programs that need its service. For example, Sharon is a programmer who has developed an object for rendering 3D images. She is a math whiz and knows a lot about computer graphics, so her object is coded to perform all the necessary 3D mathematical operations and handle the computer's video hardware. Tom, who is writing a program for an architectural firm, needs his application to display 3D images of buildings. Because he is working under a tight deadline and does not possess a great deal of knowledge about computer graphics, he can use Sharon's object to perform the 3D rendering (for a small fee, of course!).

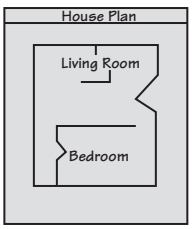
## Classes and Objects

Now let's discuss how objects are created in software. Before an object can be created, it must be designed by a programmer. The programmer determines the attributes and functions that are necessary and then creates a class. A *class* is code that specifies the attributes

and member functions that a particular type of object may have. Think of a class as a "blueprint" that objects may be created from. It serves a similar purpose as the blueprint for a house. The blueprint itself is not a house, but is a detailed description of a house. When we use the blueprint to build an actual house, we could say we are building an instance of the house described by the blueprint. If we so desire, we can build several identical houses from the same blueprint. Each house is a separate instance of the house described by the blueprint. This idea is illustrated in Figure 13-3.

**Figure 13-3**

Blueprint that describes a house.

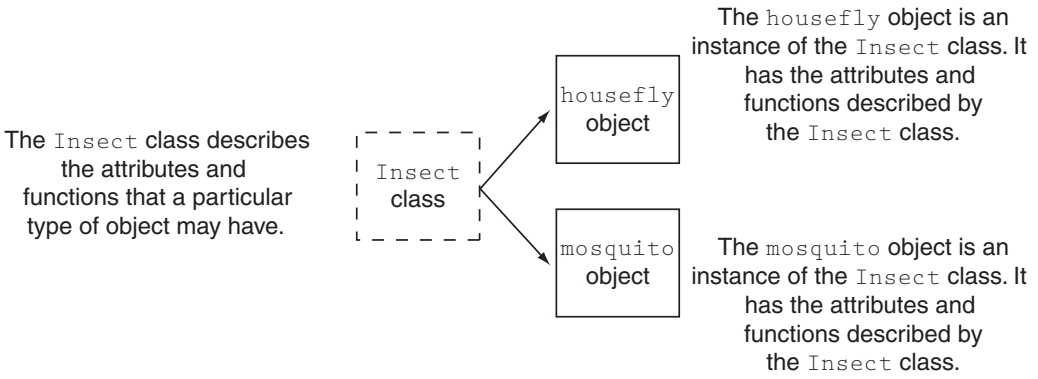



Instances of the house described by the blueprint.

So, a class is not an object, but it is a description of an object. When the program is running, it uses the class to create, in memory, as many objects of a specific type as needed. Each object that is created from a class is called an *instance* of the class.

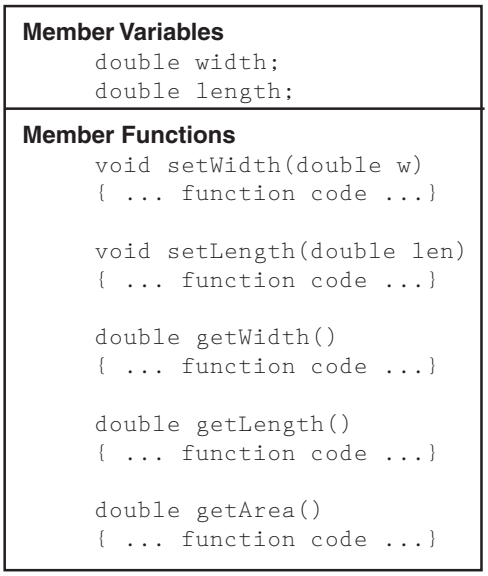
For example, Jessica is an entomologist (someone who studies insects), and she also enjoys writing computer programs. She designs a program to catalog different types of insects. As part of the program, she creates a class named `Insect`, which specifies attributes and member functions for holding and manipulating data common to all types of insects. The `Insect` class is not an object, but a specification that objects may be created from. Next, she writes programming statements that create a `housefly` object, which is an instance of the `Insect` class. The `housefly` object is an entity that occupies computer memory and stores data about a housefly. It has the attributes and member functions specified by the `Insect` class. Then she writes programming statements that create a `mosquito` object. The `mosquito` object is also an instance of the `Insect` class. It has its own area in memory and stores data about a mosquito. Although the `housefly` and `mosquito` objects are two separate entities in the computer's memory, they were both created from the `Insect` class. This means that each of the objects has the attributes and member functions described by the `Insect` class. This is illustrated in Figure 13-4.

**Figure 13-4**




At the beginning of this section we discussed how a procedural program that works with rectangles might have variables to hold the rectangle's width and length and separate functions to do things like store values in the variables and make calculations. The program would pass the variables to the functions as needed. In an object-oriented program, we would create a `Rectangle` class, which would encapsulate the data (width and length) and the functions that work with the data. Figure 13-5 shows a representation of such a class.

**Figure 13-5**




In the object-oriented approach, the variables and functions are all members of the `Rectangle` class. When we need to work with a rectangle in our program, we create a `Rectangle` object, which is an instance of the `Rectangle` class. When we need to perform an operation on the `Rectangle` object's data, we use that object to call the appropriate member function. For example, if we need to get the area of the rectangle, we use the object to call the `getArea` member function. The `getArea` member function would be designed to calculate the area of that object's rectangle and return the value.

## Using a Class You Already Know

Before we go any further, let's review the basics of a class that you have already learned something about: the `string` class. First, recall that you must have the following `#include` directive in any program that uses the `string` class:

```
#include <string>
```

This is necessary because the `string` class is declared in the `string` header file. Next, you can define a `string` object with a statement such as

```
string cityName;
```

This creates a `string` object named `cityName`. The `cityName` object is an instance of the `string` class.

Once a `string` object has been created, you can store data in it. Because the `string` class is designed to work with the assignment operator, you can assign a string literal to a `string` object. Here is an example:

```
cityName = "Charleston";
```

After this statement executes, the string `"Charleston"` will be stored in the `cityName` object. `"Charleston"` will become the object's data.

The `string` class specifies numerous member functions that perform operations on the data that a `string` object holds. For example, it has a member function named `length`, which returns the length of the string stored in a `string` object. The following code demonstrates:

```
string cityName;              // Create a string object named cityName
int strSize;                  // To hold the length of a string
cityName = "Charleston";      // Assign "Charleston" to cityName
strSize = cityName.length(); // Store the string length in strSize
```

The last statement calls the `length` member function, which returns the length of a string. The expression `cityName.length()` returns the length of the string stored in the `cityName` object. After this statement executes, the `strSize` variable will contain the value 10, which is the length of the string `"Charleston"`.

The `string` class also specifies a member function named `append`, which appends an additional string onto the string already stored in an object. The following code demonstrates.

```
string cityName;
cityName = "Charleston";
cityName.append(" South Carolina");
```

In the second line, the string `"Charleston"` is assigned to the `cityName` object. In the third line, the `append` member function is called and `" South Carolina"` is passed as an argument. The argument is appended to the string that is already stored in `cityName`. After this statement executes, the `cityName` object will contain the string `"Charleston South Carolina"`.

## 13.2 Introduction to Classes

**CONCEPT:** In C++, the class is the construct primarily used to create objects.

A *class* is similar to a structure. It is a data type defined by the programmer, consisting of variables and functions. Here is the general format of a class declaration:


VideoNote
**Writing a Class**

```
class ClassName
{
   declaration;
   // ... more declarations
   // may follow...
};
```

The declaration statements inside a class declaration are for the variables and functions that are members of that class. For example, the following code declares a class named `Rectangle` with two member variables: `width` and `length`.

```
class Rectangle
{
   double width;
   double length;
};                    // Don't forget the semicolon.
```

There is a problem with this class, however. Unlike structures, the members of a class are *private* by default. Private class members cannot be accessed by programming statements outside the class. So, no statements outside this `Rectangle` class can access the `width` and `length` members.

Recall from our earlier discussion on object-oriented programming that an object can perform data hiding, which means that critical data stored inside the object are protected from code outside the object. In C++, a class's private members are hidden and can be accessed only by functions that are members of the same class. A class's *public* members may be accessed by code outside the class.

### Access Specifiers

C++ provides the key words `private` and `public`, which you may use in class declarations. These key words are known as *access specifiers* because they specify how class members may be accessed. The following is the general format of a class declaration that uses the `private` and `public` access specifiers.

```
class ClassName
{
   private:
      // Declarations of private
      // members appear here.
   public:
      // Declarations of public
      // members appear here.
};
```

Notice that the access specifiers are followed by a colon (:) and then followed by one or more member declarations. In this general format, the `private` access specifier is used first. All of the declarations that follow it, up to the `public` access specifier, are for private members. Then, all of the declarations that follow the `public` access specifier are for public members.

## Public Member Functions

To allow access to a class's private member variables, you create public member functions that work with the private member variables. For example, consider the `Rectangle` class. To allow access to a `Rectangle` object's `width` and `length` member variables, we will add the member functions listed in Table 13-3.

**Table 13-3**

| Member Function | Description |
|---|---|
| `setWidth` | This function accepts an argument, which is assigned to the `width` member variable. |
| `setLength` | This function accepts an argument, which is assigned to the `length` member variable. |
| `getWidth` | This function returns the value stored in the `width` member variable. |
| `getLength` | This function returns the value stored in the `length` member variable. |
| `getArea` | This function returns the product of the `width` member variable multiplied by the `length` member variable. This value is the area of the rectangle. |

For the moment we will not actually define the functions described in Table 13-3. We leave that for later. For now we will only include declarations, or prototypes, for the functions in the class declaration:

```
class Rectangle
{
   private:
     double width;
     double length;
   public:
     void setWidth(double);
     void setLength(double);
     double getWidth() const;
     double getLength() const;
     double getArea() const;
};
```

In this declaration, the member variables `width` and `length` are declared as `private`, which means they can be accessed only by the class's member functions. The member functions, however, are declared as `public`, which means they can be called from statements outside the class. If code outside the class needs to store a width or a length in a `Rectangle` object, it must do so by calling the object's `setWidth` or `setLength` member functions. Likewise, if code outside the class needs to retrieve a width or length stored in a `Rectangle` object, it must do so with the object's `getWidth` or `getLength` member functions. These public functions provide an interface for code outside the class to use `Rectangle` objects.

**NOTE:** Even though the default access of a class is private, it's still a good idea to use the `private` key word to explicitly declare private members. This clearly documents the access specification of all the members of the class.

## Using `const` with Member Functions

Notice that the key word `const` appears in the declarations of the `getWidth`, `getLength`, and `getArea` member functions, as shown here:

```
double getWidth() const;
double getLength() const;
double getArea() const;
```

When the key word `const` appears after the parentheses in a member function declaration, it specifies that the function will not change any data stored in the calling object. If you inadvertently write code in the function that changes the calling object's data, the compiler will generate an error. As you will see momentarily, the `const` key word must also appear in the function header.

## Placement of `public` and `private` Members

There is no rule requiring you to declare private members before public members. The `Rectangle` class could be declared as follows:

```
class Rectangle
{
   public:
      void setWidth(double);
      void setLength(double);
      double getWidth() const;
      double getLength() const;
      double getArea() const;
   private:
      double width;
      double length;
};
```

In addition, it is not required that all members of the same access specification be declared in the same place. Here is yet another declaration of the `Rectangle` class.

```
class Rectangle
{
   private:
      double width;
   public:
      void setWidth(double);
      void setLength(double);
      double getWidth() const;
      double getLength() const;
      double getArea() const;
   private:
      double length;
};
```

Although C++ gives you freedom in arranging class member declarations, you should adopt a consistent standard. Most programmers choose to group member declarations of the same access specification together.

**NOTE:** Notice in our example that the first character of the class name is written in uppercase. This is not required, but serves as a visual reminder that the class name is not a variable name.

## Defining Member Functions

The `Rectangle` class declaration contains declarations or prototypes for five member functions: `setWidth`, `setLength`, `getWidth`, `getLength`, and `getArea`. The definitions of these functions are written outside the class declaration:

```
//*************************************************************
// setWidth assigns its argument to the private member width. *
//*************************************************************

void Rectangle::setWidth(double w)
{
   width = w;
}


//**************************************************************
// setLength assigns its argument to the private member length. *
//**************************************************************

void Rectangle::setLength(double len)
{
   length = len;
}

//*************************************************************
// getWidth returns the value in the private member width. *
//*************************************************************

double Rectangle::getWidth() const
{
   return width;
}
//*************************************************************
// getLength returns the value in the private member length. *
//*************************************************************

double Rectangle::getLength() const
{
   return length;
}
```

```
//****************************************************
// getArea returns the product of width times length. *
//****************************************************

double Rectangle::getArea() const
{
   return width * length;
}
```

In each function definition, the following precedes the name of each function:

```
Rectangle::
```

The two colons are called the *scope resolution operator*. When `Rectangle::` appears before the name of a function in a function header, it identifies the function as a member of the `Rectangle` class.

Here is the general format of the function header of any member function defined outside the declaration of a class:

```
ReturnType ClassName::functionName(ParameterList)
```

In the general format, *`ReturnType`* is the function's return type. *`ClassName`* is the name of the class that the function is a member of. *`functionName`* is the name of the member function. *`ParameterList`* is an optional list of parameter variable declarations.

**WARNING!** Remember, the class name and scope resolution operator extends the name of the function. They must appear after the return type and immediately before the function name in the function header. The following would be incorrect:

```
Rectangle::double getArea() //Incorrect!
```

In addition, if you leave the class name and scope resolution operator out of a member function's header, the function will not become a member of the class.

```
double getArea() // Not a member of the Rectangle class!
```

## Accessors and Mutators

As mentioned earlier, it is a common practice to make all of a class's member variables private and to provide public member functions for accessing and changing them. This ensures that the object owning the member variables is in control of all changes being made to them. A member function that gets a value from a class's member variable but does not change it is known as an *accessor*. A member function that stores a value in member variable or changes the value of member variable in some other way is known as a *mutator*. In the `Rectangle` class, the member functions `getLength` and `getWidth` are accessors, and the member functions `setLength` and `setWidth` are mutators.

Some programmers refer to mutators as *setter functions* because they set the value of an attribute, and accessors as *getter functions* because they get the value of an attribute.

### Using `const` with Accessors

Notice that the key word `const` appears in the headers of the `getWidth`, `getLength`, and `getArea` member functions, as shown here:

```
double Rectangle::getWidth() const
double Rectangle::getLength() const
double Rectangle::getArea() const
```

Recall that these functions were also declared in the class with the `const` key word. When you mark a member function as `const`, the `const` key word must appear in both the declaration and the function header.

In essence, when you mark a member function as `const`, you are telling the compiler that the calling object is a constant. The compiler will generate an error if you inadvertently write code in the function that changes the calling object's data. Because this decreases the chances of having bugs in your code, it is a good practice to mark all accessor functions as `const`.

### The Importance of Data Hiding

As a beginning student, you might be wondering why you would want to hide the data that is inside the classes you create. As you learn to program, you will be the user of your own classes, so it might seem that you are putting forth a great effort to hide data from yourself. If you write software in industry, however, the classes that you create will be used as components in large software systems; programmers other than yourself will use your classes. By hiding a class's data and allowing it to be accessed through only the class's member functions, you can better ensure that the class will operate as you intended it to.

## 13.3 Defining an Instance of a Class

**CONCEPT:** Class objects must be defined after the class is declared.

VideoNote
**Defining an Instance of a Class**

Like structure variables, class objects are not created in memory until they are defined. This is because a class declaration by itself does not create an object, but is merely the description of an object. We can use it to create one or more objects, which are instances of the class.

Class objects are created with simple definition statements, just like variables. Here is the general format of a simple object definition statement:

```
ClassName objectName;
```

In the general format, *`ClassName`* is the name of a class, and *`objectName`* is the name we are giving the object.

For example, the following statement defines `box` as an object of the `Rectangle` class:

```
Rectangle box;
```

Defining a class object is called the *instantiation* of a class. In this statement, `box` is an *instance* of the `Rectangle` class.

## Accessing an Object's Members

The box object that we previously defined is an instance of the Rectangle class. Suppose we want to change the value in the box object's width variable. To do so, we must use the box object to call the setWidth member function, as shown here:

```
box.setWidth(12.7);
```

Just as you use the dot operator to access a structure's members, you use the dot operator to call a class's member functions. This statement uses the box object to call the setWidth member function, passing 12.7 as an argument. As a result, the box object's width variable will be set to 12.7. Here are other examples of statements that use the box object to call member functions:

```
box.setLength(4.8);       // Set box's length to 4.8.
x = box.getWidth();       // Assign box's width to x.
cout << box.getLength(); // Display box's length.
cout << box.getArea();    // Display box's area.
```

**NOTE:** Notice that inside the Rectangle class's member functions, the dot operator is not used to access any of the class's member variables. When an object is used to call a member function, the member function has direct access to that object's member variables.

## A Class Demonstration Program

Program 13-1 is a complete program that demonstrates the Rectangle class.

**Program 13-1**

```
1  // This program demonstrates a simple class.
2  #include <iostream>
3  using namespace std;
4
5  // Rectangle class declaration.
6  class Rectangle
7  {
8      private:
9          double width;
10         double length;
11     public:
12         void setWidth(double);
13         void setLength(double);
14         double getWidth() const;
15         double getLength() const;
16         double getArea() const;
17 };
18
19 //************************************************
20 // setWidth assigns a value to the width member.  *
21 //************************************************
22
```

```
23  void Rectangle::setWidth(double w)
24  {
25      width = w;
26  }
27
28  //***************************************************
29  // setLength assigns a value to the length member. *
30  //***************************************************
31
32  void Rectangle::setLength(double len)
33  {
34      length = len;
35  }
36
37  //***************************************************
38  // getWidth returns the value in the width member. *
39  //***************************************************
40
41  double Rectangle::getWidth() const
42  {
43      return width;
44  }
45
46  //***************************************************
47  // getLength returns the value in the length member. *
48  //***************************************************
49
50  double Rectangle::getLength() const
51  {
52      return length;
53  }
54
55  //***************************************************
56  // getArea returns the product of width times length. *
57  //***************************************************
58
59  double Rectangle::getArea() const
60  {
61      return width * length;
62  }
63
64  //***************************************************
65  // Function main                                   *
66  //***************************************************
67
68  int main()
69  {
70      Rectangle box;      // Define an instance of the Rectangle class
71      double rectWidth;   // Local variable for width
72      double rectLength;  // Local variable for length
73
74      // Get the rectangle's width and length from the user.
75      cout << "This program will calculate the area of a\n";
76      cout << "rectangle. What is the width? ";
```

*(program continues)*

**Program 13-1** *(continued)*

```
77       cin >> rectWidth;
78       cout << "What is the length? ";
79       cin >> rectLength;
80
81       // Store the width and length of the rectangle
82       // in the box object.
83       box.setWidth(rectWidth);
84       box.setLength(rectLength);
85
86       // Display the rectangle's data.
87       cout << "Here is the rectangle's data:\n";
88       cout << "Width: " << box.getWidth() << endl;
89       cout << "Length: " << box.getLength() << endl;
90       cout << "Area: " << box.getArea() << endl;
91       return 0;
92   }
```

**Program Output with Example Input Shown in Bold**

```
This program will calculate the area of a
rectangle. What is the width? 10 [Enter]
What is the length? 5 [Enter]
Here is the rectangle's data:
Width: 10
Length: 5
Area: 50
```

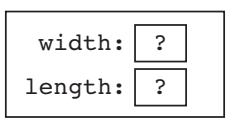
The `Rectangle` class declaration, along with the class's member functions, appears in lines 6 through 62. Inside the `main` function, in line 70, the following statement creates a `Rectangle` object named `box`.

```
Rectangle box;
```

The `box` object is illustrated in Figure 13-6. Notice that the `width` and `length` member variables do not yet hold meaningful values. An object's member variables are not automatically initialized to 0. When an object's member variable is first created, it holds whatever random value happens to exist at the variable's memory location. We commonly refer to such a random value as "garbage."

**Figure 13-6**

The box object when first created




In lines 75 through 79 the program prompts the user to enter the width and length of a rectangle. The width that is entered is stored in the `rectWidth` variable, and the length that is entered is stored in the `rectLength` variable. In line 83 the following statement uses

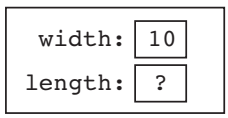
the box object to call the setWidth member function, passing the value of the rectWidth variable as an argument:

```
box.setWidth(rectWidth);
```

This sets box's width member variable to the value in rectWidth. Assuming rectWidth holds the value 10, Figure 13-7 shows the state of the box object after this statement executes.

**Figure 13-7**

The box object with width set to 10



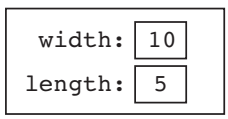

In line 84 the following statement uses the box object to call the setLength member function, passing the value of the rectLength variable as an argument.

```
box.setLength(rectLength);
```

This sets box's length member variable to the value in rectLength. Assuming rectLength holds the value 5, Figure 13-8 shows the state of the box object after this statement executes.

**Figure 13-8**

The box object with width set to 10 and length set to 5




Lines 88, 89, and 90 use the box object to call the getWidth, getLength, and getArea member functions, displaying their return values on the screen.

**NOTE:** Figures 13-6 through 13-8 show the state of the box object at various times during the execution of the program. An object's *state* is simply the data that is stored in the object's attributes at any given moment.

Program 13-1 creates only one Rectangle object. It is possible to create many instances of the same class, each with its own data. For example, Program 13-2 creates three Rectangle objects, named kitchen, bedroom, and den. Note that lines 6 through 62 have been left out of the listing because they contain the Rectangle class declaration and the definitions for the class's member functions. These lines are identical to those same lines in Program 13-1.

**Program 13-2**

```
1  // This program creates three instances of the Rectangle class.
2  #include <iostream>
3  using namespace std;
4
5  // Rectangle class declaration.
       Lines 6 through 62 have been left out.
63
64 //***************************************************
65 // Function main                                    *
66 //***************************************************
67
68 int main()
69 {
70     double number;         // To hold a number
71     double totalArea;      // The total area
72     Rectangle kitchen;     // To hold kitchen dimensions
73     Rectangle bedroom;     // To hold bedroom dimensions
74     Rectangle den;         // To hold den dimensions
75
76     // Get the kitchen dimensions.
77     cout << "What is the kitchen's length? ";
78     cin >> number;                          // Get the length
79     kitchen.setLength(number);              // Store in kitchen object
80     cout << "What is the kitchen's width? ";
81     cin >> number;                          // Get the width
82     kitchen.setWidth(number);               // Store in kitchen object
83
84     // Get the bedroom dimensions.
85     cout << "What is the bedroom's length? ";
86     cin >> number;                          // Get the length
87     bedroom.setLength(number);              // Store in bedroom object
88     cout << "What is the bedroom's width? ";
89     cin >> number;                          // Get the width
90     bedroom.setWidth(number);               // Store in bedroom object
91
92     // Get the den dimensions.
93     cout << "What is the den's length? ";
94     cin >> number;                          // Get the length
95     den.setLength(number);                  // Store in den object
96     cout << "What is the den's width? ";
97     cin >> number;                          // Get the width
98     den.setWidth(number);                   // Store in den object
99
100    // Calculate the total area of the three rooms.
101    totalArea = kitchen.getArea() + bedroom.getArea() +
102                den.getArea();
103
104    // Display the total area of the three rooms.
105    cout << "The total area of the three rooms is "
106         << totalArea << endl;
107
```

```
108      return 0;
109 }
```

**Program Output with Example Input Shown in Bold**

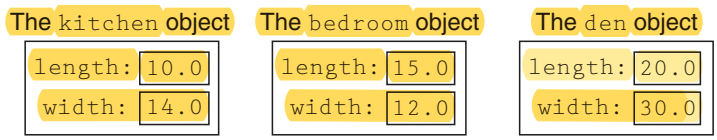
```
What is the kitchen's length? 10 [Enter]
What is the kitchen's width? 14 [Enter]
What is the bedroom's length? 15 [Enter]
What is the bedroom's width? 12 [Enter]
What is the den's length? 20 [Enter]
What is the den's width? 30 [Enter]
The total area of the three rooms is 920
```

In lines 72, 73, and 74, the following code defines three `Rectangle` variables. This creates three objects, each an instance of the `Rectangle` class:

```
Rectangle kitchen;   // To hold kitchen dimensions
Rectangle bedroom;   // To hold bedroom dimensions
Rectangle den;       // To hold den dimensions
```

In the example output, the user enters 10 and 14 as the length and width of the kitchen, 15 and 12 as the length and width of the bedroom, and 20 and 30 as the length and width of the den. Figure 13-9 shows the states of the objects after these values are stored in them.

**Figure 13-9**




Notice from Figure 13-9 that each instance of the `Rectangle` class has its own `length` and `width` variables. Every instance of a class has its own set of member variables that can hold their own values. The class's member functions can perform operations on specific instances of the class. For example, look at the following statement in line 79 of Program 13-2:

```
kitchen.setLength(number);
```

This statement calls the `setLength` member function, which stores a value in the `kitchen` object's `length` variable. Now look at the following statement in line 87:

```
bedroom.setLength(number);
```

This statement also calls the `setLength` member function, but this time it stores a value in the `bedroom` object's `length` variable. Likewise, the following statement in line 95 calls the `setLength` member function to store a value in the `den` object's `length` variable:

```
den.setLength(number);
```

The `setLength` member function stores a value in a specific instance of the `Rectangle` class. All of the other `Rectangle` class member functions work in a similar way. They access one or more member variables of a specific `Rectangle` object.

## Avoiding Stale Data

In the `Rectangle` class, the `getLength` and `getWidth` member functions return the values stored in member variables, but the `getArea` member function returns the result of a calculation. You might be wondering why the area of the rectangle is not stored in a member variable, like the length and the width. The area is not stored in a member variable because it could potentially become stale. When the value of an item is dependent on other data and that item is not updated when the other data are changed, it is said that the item has become *stale*. If the area of the rectangle were stored in a member variable, the value of the member variable would become incorrect as soon as either the `length` or `width` member variables changed.

When designing a class, you should take care not to store in a member variable calculated data that could potentially become stale. Instead, provide a member function that returns the result of the calculation.

## Pointers to Objects

You can also define pointers to class objects. For example, the following statement defines a pointer variable named `rectPtr`:

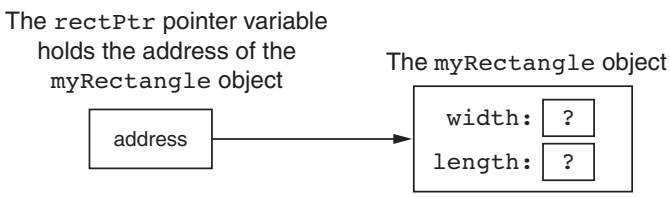
```
Rectangle *rectPtr = nullptr;
```

The `rectPtr` variable is not an object, but it can hold the address of a `Rectangle` object. The following code shows an example.

```
Rectangle myRectangle;          // A Rectangle object
Rectangle *rectPtr = nullptr;   // A Rectangle pointer
rectPtr = &myRectangle;         // rectPtr now points to myRectangle
```

The first statement creates a `Rectangle` object named `myRectangle`. The second statement creates a `Rectangle` pointer named `rectPtr`. The third statement stores the address of the `myRectangle` object in the `rectPtr` pointer. This is illustrated in Figure 13-10.

**Figure 13-10**



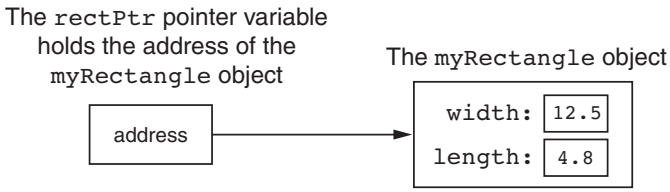

The `rectPtr` pointer can then be used to call member functions by using the `->` operator. The following statements show examples.

```
rectPtr->setWidth(12.5);
rectPtr->setLength(4.8);
```

The first statement calls the `setWidth` member function, passing 12.5 as an argument. Because `rectPtr` points to the `myRectangle` object, this will cause 12.5 to be stored in the `myRectangle` object's `width` variable. The second statement calls the `setLength` member function, passing 4.8 as an argument. This will cause 4.8 to be stored in the `myRectangle`

object's `length` variable. Figure 13-11 shows the state of the `myRectangle` object after these statements have executed.

**Figure 13-11**

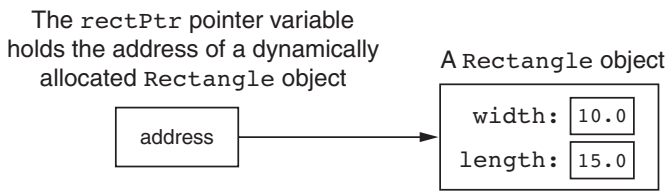



Class object pointers can be used to dynamically allocate objects. The following code shows an example.

```
1  // Define a Rectangle pointer.
2  Rectangle *rectPtr = nullptr;
3
4  // Dynamically allocate a Rectangle object.
5  rectPtr = new Rectangle;
6
7  // Store values in the object's width and length.
8  rectPtr->setWidth(10.0);
9  rectPtr->setLength(15.0);
10
11 // Delete the object from memory.
12 delete rectPtr;
13 rectPtr = nullptr;
```

Line 2 defines `rectPtr` as a `Rectangle` pointer. Line 5 uses the `new` operator to dynamically allocate a `Rectangle` object and assign its address to `rectPtr`. Lines 8 and 9 store values in the dynamically allocated object's `width` and `length` variables. Figure 13-12 shows the state of the dynamically allocated object after these statements have executed.

**Figure 13-12**




Line 12 deletes the object from memory, and line 13 stores the address 0 in `rectPtr`. Recall from Chapter 9 that this prevents code from inadvertently using the pointer to access the area of memory that has been freed. It also prevents errors from occurring if `delete` is accidentally called on the pointer again.

Program 13-3 is a modification of Program 13-2. In this program, `kitchen`, `bedroom`, and `den` are `Rectangle` pointers. They are used to dynamically allocate `Rectangle` objects. The output is the same as Program 13-2.

**Program 13-3**

```
1  // This program creates three instances of the Rectangle class.
2  #include <iostream>
3  using namespace std;
4
5  // Rectangle class declaration.
```

*Lines 6 through 62 have been left out.*

```
63
64 //*************************************************
65 // Function main                                  *
66 //*************************************************
67
68 int main()
69 {
70     double number;                 // To hold a number
71     double totalArea;              // The total area
72     Rectangle *kitchen = nullptr;  // To point to kitchen dimensions
73     Rectangle *bedroom = nullptr;  // To point to bedroom dimensions
74     Rectangle *den = nullptr;      // To point to den dimensions
75
76     // Dynamically allocate the objects.
77     kitchen = new Rectangle;
78     bedroom = new Rectangle;
79     den = new Rectangle;
80
81     // Get the kitchen dimensions.
82     cout << "What is the kitchen's length? ";
83     cin >> number;                           // Get the length
84     kitchen->setLength(number);              // Store in kitchen object
85     cout << "What is the kitchen's width? ";
86     cin >> number;                           // Get the width
87     kitchen->setWidth(number);               // Store in kitchen object
88
89     // Get the bedroom dimensions.
90     cout << "What is the bedroom's length? ";
91     cin >> number;                           // Get the length
92     bedroom->setLength(number);              // Store in bedroom object
93     cout << "What is the bedroom's width? ";
94     cin >> number;                           // Get the width
95     bedroom->setWidth(number);               // Store in bedroom object
96
97     // Get the den dimensions.
98     cout << "What is the den's length? ";
99     cin >> number;                           // Get the length
100    den->setLength(number);                  // Store in den object
101    cout << "What is the den's width? ";
102    cin >> number;                           // Get the width
103    den->setWidth(number);                   // Store in den object
104
105    // Calculate the total area of the three rooms.
```

```
106     totalArea = kitchen->getArea() + bedroom->getArea() +
107                 den->getArea();
108
109     // Display the total area of the three rooms.
110     cout << "The total area of the three rooms is "
111          << totalArea << endl;
112
113     // Delete the objects from memory.
114     delete kitchen;
115     delete bedroom;
116     delete den;
117     kitchen = nullptr;    // Make kitchen a null pointer.
118     bedroom = nullptr;    // Make bedroom a null pointer.
119     den = nullptr;        // Make den a null pointer.
120
121     return 0;
122 }
```

## Using Smart Pointers to Allocate Objects

Chapter 9 discussed the smart pointer data type `unique_ptr`, which was introduced in C++ 11. Recall from Chapter 9 that you can use a `unique_ptr` to dynamically allocate memory, and not worry about deleting the memory when you are finished using it. A `unique_ptr` automatically deletes a chunk of dynamically allocated memory when the memory is no longer being used. This helps to prevent memory leaks from occurring.

To use the `unique_ptr` data type, you must `#include` the `memory` header file with the following directive:

```
#include <memory>
```

Here is an example of the syntax for defining a `unique_ptr` that points to a dynamically allocated `Rectangle` object:

```
unique_ptr<Rectangle> rectanglePtr(new Rectangle);
```

This statement defines a `unique_ptr` named `rectanglePtr` that points to a dynamically allocated `Rectangle` object. Here are some details about the statement:

- The notation `<Rectangle>` that appears immediately after `unique_ptr` indicates that the pointer can point to a `Rectangle`.
- The name of the pointer is `rectanglePtr`.
- The expression `new Rectangle` that appears inside the parentheses allocates a chunk of memory to hold a `Rectangle`. The address of the chunk of memory will be assigned to the `rectanglePtr` pointer.

Once you have defined a `unique_ptr`, you can use it in the same way as a regular pointer. This is demonstrated in Program 13-4. This is a revised version of Program 13-3, modified to use `unique_ptrs` instead of regular pointers. The output is the same as Programs 13-2 and 13-3.

**Program 13-4**

```
// This program uses smart pointers to allocate three
// instances of the Rectangle class.
#include <iostream>
#include <memory>
using namespace std;

// Rectangle class declaration.
```

*Lines 8 through 64 have been left out.*

```

//*************************************************
// Function main                                  *
//*************************************************

int main()
{
    double number;           // To hold a number
    double totalArea;        // The total area

    // Dynamically allocate the objects.
    unique_ptr<Rectangle> kitchen(new Rectangle);
    unique_ptr<Rectangle> bedroom(new Rectangle);
    unique_ptr<Rectangle> den(new Rectangle);

    // Get the kitchen dimensions.
    cout << "What is the kitchen's length? ";
    cin >> number;                          // Get the length
    kitchen->setLength(number);             // Store in kitchen object
    cout << "What is the kitchen's width? ";
    cin >> number;                          // Get the width
    kitchen->setWidth(number);              // Store in kitchen object

    // Get the bedroom dimensions.
    cout << "What is the bedroom's length? ";
    cin >> number;                          // Get the length
    bedroom->setLength(number);             // Store in bedroom object
    cout << "What is the bedroom's width? ";
    cin >> number;                          // Get the width
    bedroom->setWidth(number);              // Store in bedroom object

    // Get the den dimensions.
    cout << "What is the den's length? ";
    cin >> number;                          // Get the length
    den->setLength(number);                 // Store in den object
    cout << "What is the den's width? ";
    cin >> number;                          // Get the width
    den->setWidth(number);                  // Store in den object

    // Calculate the total area of the three rooms.
    totalArea = kitchen->getArea() + bedroom->getArea() +
                den->getArea();

```

```
108     // Display the total area of the three rooms.
109     cout << "The total area of the three rooms is "
110          << totalArea << endl;
111
112     return 0;
113 }
```

In line 4, we have a `#include` directive for the `memory` header file. Lines 76 through 78 define three `unique_ptrs`, named `kitchen`, `bedroom`, and `den`. Each of these points to a dynamically allocated `Rectangle`. Notice there are no `delete` statements at the end of the `main` function to free the dynamically allocated memory. It is unnecessary to delete the dynamically allocated `Rectangle` objects because the smart pointer will automatically delete them as the function comes to an end.

## Checkpoint

13.1 True or False: You must declare all private members of a class before the public members.

13.2 Assume that `RetailItem` is the name of a class, and the class has a `void` member function named `setPrice`, which accepts a `double` argument. Which of the following shows the correct use of the scope resolution operator in the member function definition?

A) `RetailItem::void setPrice(double p)`
B) `void RetailItem::setPrice(double p)`

13.3 An object's private member variables are accessed from outside the object by

A) public member functions
B) any function
C) the dot operator
D) the scope resolution operator

13.4 Assume that `RetailItem` is the name of a class, and the class has a `void` member function named `setPrice`, which accepts a `double` argument. If `soap` is an instance of the `RetailItem` class, which of the following statements properly uses the `soap` object to call the `setPrice` member function?

A) `RetailItem::setPrice(1.49);`
B) `soap::setPrice(1.49);`
C) `soap.setPrice(1.49);`
D) `soap:setPrice(1.49);`

13.5 Complete the following code skeleton to declare a class named `Date`. The class should contain variables and functions to store and retrieve a date in the form 4/2/2014.

```
class Date
{
    private:
    public:
}
```

## 13.4 Why Have Private Members?

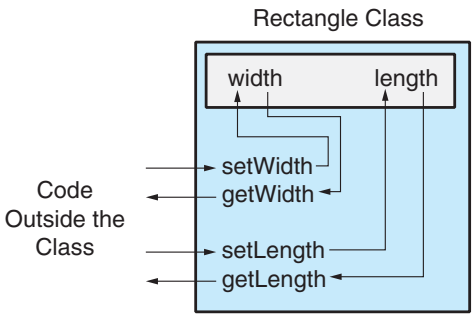
**CONCEPT:** In object-oriented programming, an object should protect its important data by making it private and providing a public interface to access that data.

You might be questioning the rationale behind making the member variables in the `Rectangle` class private. You might also be questioning why member functions were defined for such simple tasks as setting variables and getting their contents. After all, if the member variables were declared as `public`, the member functions wouldn't be needed.

As mentioned earlier in this chapter, classes usually have variables and functions that are meant only to be used internally. They are not intended to be accessed by statements outside the class. This protects critical data from being accidentally modified or used in a way that might adversely affect the state of the object. When a member variable is declared as `private`, the only way for an application to store values in the variable is through a public member function. Likewise, the only way for an application to retrieve the contents of a private member variable is through a public member function. In essence, the public members become an interface to the object. They are the only members that may be accessed by any application that uses the object.

In the `Rectangle` class, the `width` and `length` member variables hold critical data. Therefore they are declared as `private`, and an interface is constructed with public member functions. If a program creates a `Rectangle` object, the program must use the `setWidth` and `getWidth` member functions to access the object's `width` member. To access the object's `length` member, the program must use the `setLength` and `getLength` member functions. This idea is illustrated in Figure 13-13.

**Figure 13-13**




The public member functions can be written to filter out invalid data. For example, look at the following version of the `setWidth` member function.

```
void Rectangle::setWidth(double w)
{
   if (w >= 0)
      width = w;
```

```
    else
    {
        cout << "Invalid width\n";
        exit(EXIT_FAILURE);
    }
}
```

Notice that this version of the function doesn't just assign the parameter value to the `width` variable. It first tests the parameter to make sure it is 0 or greater. If a negative number was passed to the function, an error message is displayed, and then the standard library function `exit` is called to abort the program. The `setLength` function could be written in a similar way:

```
void Rectangle::setLength(double len)
{
    if (len >= 0)
        length = len;
    else
    {
        cout << "Invalid length\n";
        exit(EXIT_FAILURE);
    }
}
```

The point being made here is that mutator functions can do much more than simply store values in attributes. They can also validate those values to ensure that only acceptable data is stored in the object's attributes. Keep in mind, however, that calling the `exit` function, as we have done in these examples, is not the best way to deal with invalid data. In reality, you would not design a class to abort the entire program just because invalid data were passed to a mutator function. In Chapter 15 we will discuss exceptions, which provide a much better way for classes to handle errors. Until we discuss exceptions, however, we will keep our code simple by using only rudimentary data validation techniques.

## 13.5 Focus on Software Engineering: Separating Class Specification from Implementation

**CONCEPT:** **Usually class declarations are stored in their own header files. Member function definitions are stored in their own `.cpp` files.**

In the programs we've looked at so far, the class declaration, member function definitions, and application program are all stored in one file. A more conventional way of designing C++ programs is to store class declarations and member function definitions in their own separate files. Typically, program components are stored in the following fashion:

- Class declarations are stored in their own header files. A header file that contains a class declaration is called a *class specification* file. The name of the class specification file is usually the same as the name of the class, with a `.h` extension. For example, the `Rectangle` class would be declared in the file `Rectangle.h`.

- The member function definitions for a class are stored in a separate `.cpp` file called the *class implementation* file. The file usually has the same name as the class, with the `.cpp` extension. For example, the `Rectangle` class's member functions would be defined in the file `Rectangle.cpp`.
- Any program that uses the class should `#include` the class's header file. The class's `.cpp` file (that which contains the member function definitions) should be compiled and linked with the `main` program. This process can be automated with a `project` or `make` utility. Integrated development environments such as Visual Studio also provide the means to create the multi-file projects.

Let's see how we could rewrite Program 13-1 using this design approach. First, the `Rectangle` class declaration would be stored in the following `Rectangle.h` file. (This file is stored in the Student Source Code Folder `Chapter 13\Rectangle Version 1`.)

### Contents of `Rectangle.h` (Version 1)

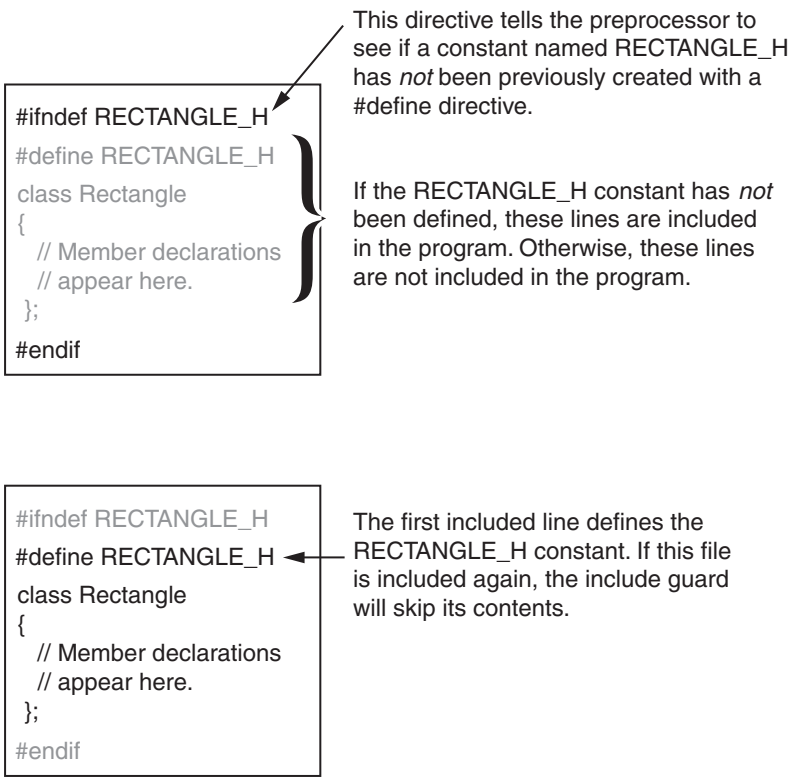
```
1  // Specification file for the Rectangle class.
2  #ifndef RECTANGLE_H
3  #define RECTANGLE_H
4
5  // Rectangle class declaration.
6
7  class Rectangle
8  {
9       private:
10           double width;
11           double length;
12      public:
13           void setWidth(double);
14           void setLength(double);
15           double getWidth() const;
16           double getLength() const;
17           double getArea() const;
18 };
19
20 #endif
```

This is the specification file for the `Rectangle` class. It contains only the declaration of the `Rectangle` class. It does not contain any member function definitions. When we write other programs that use the `Rectangle` class, we can have an `#include` directive that includes this file. That way, we won't have to write the class declaration in every program that uses the `Rectangle` class.

This file also introduces two new preprocessor directives: `#ifndef` and `#endif`. The `#ifndef` directive that appears in line 2 is called an *include guard*. It prevents the header file from accidentally being included more than once. When your main program file has an `#include` directive for a header file, there is always the possibility that the header file will have an `#include` directive for a second header file. If your main program file also has an `#include` directive for the second header file, then the preprocessor will include the second header file twice. Unless an include guard has been written into the second header file, an error will occur because the compiler will process the declarations in the second header file twice. Let's see how an include guard works.

The word `ifndef` stands for "if not defined." It is used to determine whether a specific constant has not been defined with a `#define` directive. When the `Rectangle.h` file is being compiled, the `#ifndef` directive checks for the existence of a constant named `RECTANGLE_H`. If the constant has not been defined, it is immediately defined in line 3, and the rest of the file is included. If the constant has been defined, it means that the file has already been included. In that case, everything between the `#ifndef` and `#endif` directives is skipped. This is illustrated in Figure 13-14.

**Figure 13-14**




Next we need an implementation file that contains the class's member function definitions. The implementation file for the `Rectangle` class is `Rectangle.cpp`. (This file is stored in the Student Source Code Folder `Chapter 13\Rectangle Version 1`.)

### Contents of `Rectangle.cpp` (Version 1)

```
1  // Implementation file for the Rectangle class.
2  #include "Rectangle.h"    // Needed for the Rectangle class
3  #include <iostream>       // Needed for cout
4  #include <cstdlib>        // Needed for the exit function
5  using namespace std;
6
7  //********************************************************
8  // setWidth sets the value of the member variable width.  *
9  //********************************************************
10
```

```
11  void Rectangle::setWidth(double w)
12  {
13      if (w >= 0)
14          width = w;
15      else
16      {
17          cout << "Invalid width\n";
18          exit(EXIT_FAILURE);
19      }
20  }
21
22  //*********************************************************
23  // setLength sets the value of the member variable length. *
24  //*********************************************************
25
26  void Rectangle::setLength(double len)
27  {
28      if (len >= 0)
29          length = len;
30      else
31      {
32          cout << "Invalid length\n";
33          exit(EXIT_FAILURE);
34      }
35  }
36
37  //*********************************************************
38  // getWidth returns the value in the member variable width. *
39  //*********************************************************
40
41  double Rectangle::getWidth() const
42  {
43      return width;
44  }
45
46  //**********************************************************
47  // getLength returns the value in the member variable length. *
48  //**********************************************************
49
50  double Rectangle::getLength() const
51  {
52      return length;
53  }
54
55  //*********************************************************
56  // getArea returns the product of width times length.       *
57  //*********************************************************
58
59  double Rectangle::getArea() const
60  {
61      return width * length;
62  }
```

Look at line 2, which has the following `#include` directive:

```
#include "Rectangle.h"
```

This directive includes the `Rectangle.h` file, which contains the `Rectangle` class declaration. Notice that the name of the header file is enclosed in double-quote characters (`" "`) instead of angled brackets (< >). When you are including a C++ system header file, such as `iostream`, you enclose the name of the file in angled brackets. This indicates that the file is located in the compiler's *include file directory*. The include file directory is the directory or folder where all of the standard C++ header files are located. When you are including a header file that you have written, such as a class specification file, you enclose the name of the file in double-quote marks. This indicates that the file is located in the current project directory.

Any file that uses the `Rectangle` class must have an `#include` directive for the `Rectangle.h` file. We need to include `Rectangle.h` in the class specification file because the functions in this file belong to the `Rectangle` class. Before the compiler can process a function with `Rectangle::` in its name, it must have already processed the `Rectangle` class declaration.

Now that we have the `Rectangle` class stored in its own specification and implementation files, we can see how to use them in a program. Program 13-5 shows a modified version of Program 13-1. This version of the program does not contain the `Rectangle` class declaration, or the definitions of any of the class's member functions. Instead, it is designed to be compiled and linked with the class specification and implementation files. (This file is stored in the Student Source Code Folder `Chapter 13\Rectangle Version 1`.)

**Program 13-5**

```
1  // This program uses the Rectangle class, which is declared in
2  // the Rectangle.h file. The member Rectangle class's member
3  // functions are defined in the Rectangle.cpp file. This program
4  // should be compiled with those files in a project.
5  #include <iostream>
6  #include "Rectangle.h"  // Needed for Rectangle class
7  using namespace std;
8
9  int main()
10 {
11     Rectangle box;      // Define an instance of the Rectangle class
12     double rectWidth;   // Local variable for width
13     double rectLength;  // Local variable for length
14
15     // Get the rectangle's width and length from the user.
16     cout << "This program will calculate the area of a\n";
17     cout << "rectangle. What is the width? ";
18     cin >> rectWidth;
19     cout << "What is the length? ";
20     cin >> rectLength;
21
```

*(program continues)*

**Program 13-5** *(continued)*

```
22      // Store the width and length of the rectangle
23      // in the box object.
24      box.setWidth(rectWidth);
25      box.setLength(rectLength);
26
27      // Display the rectangle's data.
28      cout << "Here is the rectangle's data:\n";
29      cout << "Width: " << box.getWidth() << endl;
30      cout << "Length: " << box.getLength() << endl;
31      cout << "Area: " << box.getArea() << endl;
32      return 0;
33  }
```

Notice that Program 13-5 has an `#include` directive for the `Rectangle.h` file in line 6. This causes the declaration for the `Rectangle` class to be included in the file. To create an executable program from this file, the following steps must be taken:

- The implementation file, `Rectangle.cpp`, must be compiled. `Rectangle.cpp` is not a complete program, so you cannot create an executable file from it alone. Instead, you compile `Rectangle.cpp` to an object file which contains the compiled code for the `Rectangle` class. This file would typically be named `Rectangle.obj`.
- The main program file, `Pr13-4.cpp`, must be compiled. This file is not a complete program either because it does not contain any of the implementation code for the `Rectangle` class. So, you compile this file to an object file such as `Pr13—4.obj`.
- The object files, `Pr13-4.obj` and `Rectangle.obj`, are linked together to create an executable file, which would be named something like `Pr13-4.exe`.

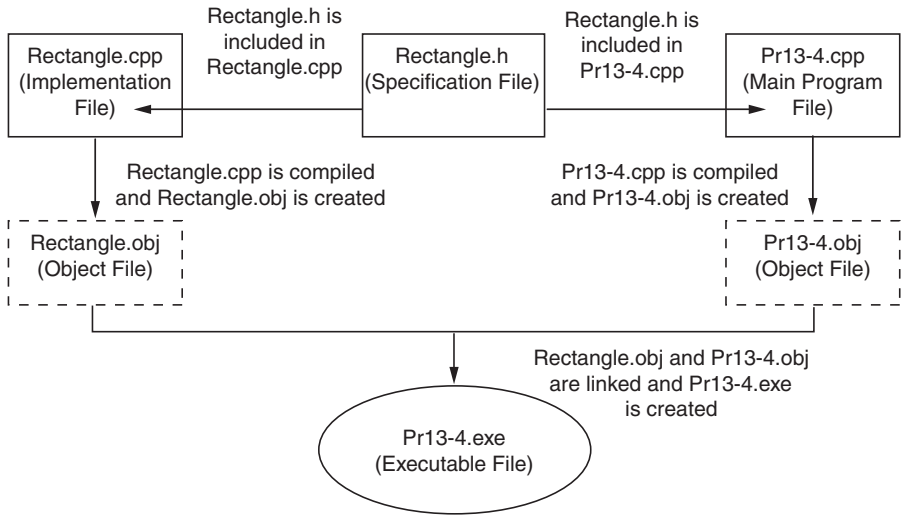
This process is illustrated in Figure 13-15.

The exact details on how these steps take place are different for each C++ development system. Fortunately, most systems perform all of these steps automatically for you. For example, in Microsoft Visual C++ you create a project, and then you simply add all of the files to the project. When you compile the project, the steps are taken care of for you and an executable file is generated.

**NOTE:** Appendix M gives step-by-step instructions for creating multi-file projects in Microsoft Visual Studio Express. You can download Appendix M from the book's companion Web site at www.pearsonhighered.com/gaddis.

Separating a class into a specification file and an implementation file provides a great deal of flexibility. First, if you wish to give your class to another programmer, you don't have to share all of your source code with that programmer. You can give him or her the specification file and the compiled object file for the class's implementation. The other programmer simply inserts the necessary `#include` directive into his or her program, compiles it, and links it with your class's object file. This prevents the other programmer, who might not know all the details of your code, from making changes that will introduce bugs.

**Figure 13-15**




Separating a class into specification and implementation files also makes things easier when the class's member functions must be modified. It is only necessary to modify the implementation file and recompile it to a new object file. Programs that use the class don't have to be completely recompiled, just linked with the new object file.

## 13.6 Inline Member Functions

**CONCEPT:** When the body of a member function is written inside a class declaration, it is declared inline.

When the body of a member function is small, it is usually more convenient to place the function's definition, instead of its prototype, in the class declaration. For example, in the `Rectangle` class the member functions `getWidth`, `getLength`, and `getArea` each have only one statement. The `Rectangle` class could be revised as shown in the following listing. (This file is stored in the Student Source Code Folder `Chapter 13\Rectangle Version 2`.)

### Contents of `Rectangle.h` (Version 2)

```
1  // Specification file for the Rectangle class
2  // This version uses some inline member functions.
3  #ifndef RECTANGLE_H
4  #define RECTANGLE_H
5
6  class Rectangle
7  {
8      private:
9         double width;
10        double length;
```

```
11      public:
12         void setWidth(double);
13         void setLength(double);
14
15         double getWidth() const
16              { return width; }
17
18         double getLength() const
19              { return length; }
20
21         double getArea() const
22              { return width * length; }
23  };
24  #endif
```

When a member function is defined in the declaration of a class, it is called an *inline function*. Notice that because the function definitions are part of the class, there is no need to use the scope resolution operator and class name in the function header.

Notice that the `getWidth`, `getLength`, and `getArea` functions are declared inline, but the `setWidth` and `setLength` functions are not. They are still defined outside the class declaration. The following listing shows the implementation file for the revised `Rectangle` class. (This file is also stored in the Student Source Code Folder `Chapter 13\ Rectangle Version 2`.)

### Contents of `Rectangle.cpp` (Version 2)

```
1   // Implementation file for the Rectangle class.
2   // In this version of the class, the getWidth, getLength,
3   // and getArea functions are written inline in Rectangle.h.
4   #include "Rectangle.h"    // Needed for the Rectangle class
5   #include <iostream>       // Needed for cout
6   #include <cstdlib>        // Needed for the exit function
7   using namespace std;
8
9   //*********************************************************
10  // setWidth sets the value of the member variable width.   *
11  //*********************************************************
12
13  void Rectangle::setWidth(double w)
14  {
15      if (w >= 0)
16         width = w;
17      else
18      {
19         cout << "Invalid width\n";
20         exit(EXIT_FAILURE);
21      }
22  }
23
24  //*********************************************************
25  // setLength sets the value of the member variable length. *
26  //*********************************************************
27
```

```
28  void Rectangle::setLength(double len)
29  {
30      if (len >= 0)
31         length = len;
32      else
33      {
34         cout << "Invalid length\n";
35         exit(EXIT_FAILURE);
36      }
37  }
```

## Inline Functions and Performance

A lot goes on "behind the scenes" each time a function is called. A number of special items, such as the function's return address in the program and the values of arguments, are stored in a section of memory called the *stack*. In addition, local variables are created and a location is reserved for the function's return value. All this overhead, which sets the stage for a function call, takes precious CPU time. Although the time needed is minuscule, it can add up if a function is called many times, as in a loop.

Inline functions are compiled differently than other functions. In the executable code, inline functions aren't "called" in the conventional sense. In a process known as *inline expansion*, the compiler replaces the call to an inline function with the code of the function itself. This means that the overhead needed for a conventional function call isn't necessary for an inline function and can result in improved performance.* Because the inline function's code can appear multiple times in the executable program, however, the size of the program can increase.†

### Checkpoint

13.6 Why would you declare a class's member variables `private`?

13.7 When a class's member variables are declared `private`, how does code outside the class store values in, or retrieve values from, the member variables?

13.8 What is a class specification file? What is a class implementation file?

13.9 What is the purpose of an include guard?

13.10 Assume the following class components exist in a program:

`BasePay` class declaration
`BasePay` member function definitions
`overtime` class declaration
`overtime` member function definitions
In what files would you store each of these components?

13.11 What is an inline member function?

---

* Because inline functions cause code to increase in size, they can decrease performance on systems that use paging.

† Writing a function inline is a request to the compiler. The compiler will ignore the request if inline expansion is not possible or practical.

## 13.7 Constructors

**CONCEPT:** A constructor is a member function that is automatically called when a class object is created.

A constructor is a member function that has the same name as the class. It is automatically called when the object is created in memory, or instantiated. It is helpful to think of constructors as initialization routines. They are very useful for initializing member variables or performing other setup operations.

To illustrate how constructors work, look at this `Demo` class declaration:

```
class Demo
{
public:
   Demo(); // Constructor
};
Demo::Demo()
{
   cout << "Welcome to the constructor!\n";
}
```

The class `Demo` only has one member: a function also named `Demo`. This function is the constructor. When an instance of this class is defined, the function `Demo` is automatically called. This is illustrated in Program 13-6.

**Program 13-6**

```
1  // This program demonstrates a constructor.
2  #include <iostream>
3  using namespace std;
4
5  // Demo class declaration.
6
7  class Demo
8  {
9  public:
10     Demo();     // Constructor
11 };
12
13 Demo::Demo()
14 {
15     cout << "Welcome to the constructor!\n";
16 }
17
18 //***************************************
19 // Function main.                       *
20 //***************************************
21
```

```
22  int main()
23  {
24       Demo demoObject; // Define a Demo object;
25
26       cout << "This program demonstrates an object\n";
27       cout << "with a constructor.\n";
28       return 0;
29  }
```

**Program Output**

```
Welcome to the constructor!
This program demonstrates an object
with a constructor.
```

Notice that the constructor's function header looks different than that of a regular member function. There is no return type—not even `void`. This is because constructors are not executed by explicit function calls and cannot return a value. The function header of a constructor's external definition takes the following form:

```
ClassName::ClassName(ParameterList)
```

In the general format, *`ClassName`* is the name of the class, and *`ParameterList`* is an optional list of parameter variable declarations.

In Program 13-6, `demoObject`'s constructor executes automatically when the object is defined. Because the object is defined before the `cout` statements in function `main`, the constructor displays its message first. Suppose we had defined the `Demo` object between two `cout` statements, as shown here.

```
cout << "This is displayed before the object is created.\n";
Demo demoObject;   // Define a Demo object.
cout << "\nThis is displayed after the object is created.\n";
```

This code would produce the following output:

```
This is displayed before the object is created.
Welcome to the constructor!
This is displayed after the object is created.
```

This simple `Demo` example illustrates when a constructor executes. More importantly, you should understand why a class should have a constructor. A constructor's purpose is to initialize an object's attributes. Because the constructor executes as soon as the object is created, it can initialize the object's data members to valid values before those members are used by other code. It is a good practice to always write a constructor for every class.

For example, the `Rectangle` class that we looked at earlier could benefit from having a constructor. A program could define a `Rectangle` object and then use that object to call the `getArea` function before any values were stored in `width` and `length`. Because the `width` and `length` member variables are not initialized, the function would return garbage. The following code shows a better version of the `Rectangle` class, equipped with a constructor. The constructor initializes both `width` and `length` to 0.0. (These files are stored in the Student Source Code Folder `Chapter 13\Rectangle Version 3`.)

### Contents of `Rectangle.h` (Version 3)

```
1  // Specification file for the Rectangle class
2  // This version has a constructor.
3  #ifndef RECTANGLE_H
4  #define RECTANGLE_H
5
6  class Rectangle
7  {
8     private:
9        double width;
10       double length;
11    public:
12        Rectangle();                // Constructor
13        void setWidth(double);
14        void setLength(double);
15
16        double getWidth() const
17           { return width; }
18
19        double getLength() const
20           { return length; }
21
22        double getArea() const
23           { return width * length; }
24 };
25 #endif
```

### Contents of `Rectangle.cpp` (Version 3)

```
1  // Implementation file for the Rectangle class.
2  // This version has a constructor.
3  #include "Rectangle.h"    // Needed for the Rectangle class
4  #include <iostream>       // Needed for cout
5  #include <cstdlib>        // Needed for the exit function
6  using namespace std;
7
8  //*********************************************************
9  // The constructor initializes width and length to 0.0.    *
10 //*********************************************************
11
12 Rectangle::Rectangle()
13 {
14     width = 0.0;
15     length = 0.0;
16 }
17
18 //*********************************************************
19 // setWidth sets the value of the member variable width.   *
20 //*********************************************************
21
```

```
22  void Rectangle::setWidth(double w)
23  {
24      if (w >= 0)
25          width = w;
26      else
27      {
28          cout << "Invalid width\n";
29          exit(EXIT_FAILURE);
30      }
31  }
32
33  //*********************************************************
34  // setLength sets the value of the member variable length. *
35  //*********************************************************
36
37  void Rectangle::setLength(double len)
38  {
39      if (len >= 0)
40          length = len;
41      else
42      {
43          cout << "Invalid length\n";
44          exit(EXIT_FAILURE);
45      }
46  }
```

Program 13-7 demonstrates this new version of the class. It creates a `Rectangle` object and then displays the values returned by the `getWidth`, `getLength`, and `getArea` member functions. (This file is also stored in the Student Source Code Folder `Chapter 13\Rectangle Version 3`.)

**Program 13-7**

```
1  // This program uses the Rectangle class's constructor.
2  #include <iostream>
3  #include "Rectangle.h" // Needed for Rectangle class
4  using namespace std;
5
6  int main()
7  {
8      Rectangle box;     // Define an instance of the Rectangle class
9
10     // Display the rectangle's data.
11     cout << "Here is the rectangle's data:\n";
12     cout << "Width: " << box.getWidth() << endl;
13     cout << "Length: " << box.getLength() << endl;
14     cout << "Area: " << box.getArea() << endl;
15     return 0;
16 }
```

*(program output continues)*

**Program 13-7** *(continued)*

**Program Output**

```
Here is the rectangle's data:
Width: 0
Length: 0
Area: 0
```

### The Default Constructor

All of the examples we have looked at in this section demonstrate default constructors. A *default constructor* is a constructor that takes no arguments. Like regular functions, constructors may accept arguments, have default arguments, be declared inline, and be overloaded. We will see examples of these as we progress through the chapter.

If you write a class with no constructor whatsoever, when the class is compiled C++ will automatically write a default constructor that does nothing. For example, the first version of the `Rectangle` class had no constructor; so, when the class was compiled C++ generated the following constructor:

```
Rectangle::Rectangle()
{ }
```

### Default Constructors and Dynamically Allocated Objects

Earlier we discussed how class objects may be dynamically allocated in memory. For example, assume the following pointer is defined in a program:

```
Rectangle *rectPtr = nullptr;
```

This statement defines `rectPtr` as a `Rectangle` pointer. It can hold the address of any `Rectangle` object. But because this statement does not actually create a `Rectangle` object, the constructor does not execute. Suppose we use the pointer in a statement that dynamically allocates a `Rectangle` object, as shown in the following code.

```
rectPtr = new Rectangle;
```

This statement creates a `Rectangle` object. When the `Rectangle` object is created by the `new` operator, its default constructor is automatically executed.

## 13.8 Passing Arguments to Constructors

**CONCEPT:** **A constructor can have parameters and can accept arguments when an object is created.**

Constructors may accept arguments in the same way as other functions. When a class has a constructor that accepts arguments, you can pass initialization values to the constructor when you create an object. For example, the following code shows yet another version of the `Rectangle` class. This version has a constructor that accepts arguments for the rectangle's width and length. (These files are stored in the Student Source Code Folder `Chapter 13\Rectangle Version 4`.)

### Contents of `Rectangle.h` (Version 4)

```
1  // Specification file for the Rectangle class
2  // This version has a constructor.
3  #ifndef RECTANGLE_H
4  #define RECTANGLE_H
5
6  class Rectangle
7  {
8      private:
9          double width;
10         double length;
11     public:
12         Rectangle(double, double);    // Constructor
13         void setWidth(double);
14         void setLength(double);
15
16         double getWidth() const
17             { return width; }
18
19         double getLength() const
20             { return length; }
21
22         double getArea() const
23             { return width * length; }
24 };
25 #endif
```

### Contents of `Rectangle.cpp` (Version 4)

```
1  // Implementation file for the Rectangle class.
2  // This version has a constructor that accepts arguments.
3  #include "Rectangle.h"    // Needed for the Rectangle class
4  #include <iostream>       // Needed for cout
5  #include <cstdlib>        // Needed for the exit function
6  using namespace std;
7
8  //*********************************************************
9  // The constructor accepts arguments for width and length. *
10 //*********************************************************
11
12 Rectangle::Rectangle(double w, double len)
13 {
14     width = w;
15     length = len;
16 }
17
18 //*********************************************************
19 // setWidth sets the value of the member variable width.   *
20 //*********************************************************
21
22 void Rectangle::setWidth(double w)
```

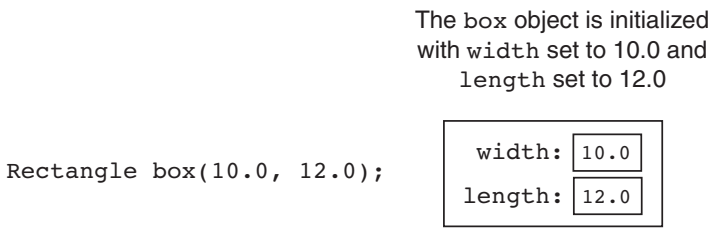
```
23  {
24       if (w >= 0)
25            width = w;
26       else
27       {
28            cout << "Invalid width\n";
29            exit(EXIT_FAILURE);
30       }
31  }
32
33  //*********************************************************
34  // setLength sets the value of the member variable length. *
35  //*********************************************************
36
37  void Rectangle::setLength(double len)
38  {
39       if (len >= 0)
40            length = len;
41       else
42       {
43            cout << "Invalid length\n";
44            exit(EXIT_FAILURE);
45       }
46  }
```

The constructor, which appears in lines 12 through 16 of `Rectangle.cpp`, accepts two arguments, which are passed into the `w` and `len` parameters. The parameters are assigned to the `width` and `length` member variables. Because the constructor is automatically called when a `Rectangle` object is created, the arguments are passed to the constructor as part of the object definition. Here is an example:

```
Rectangle box(10.0, 12.0);
```

This statement defines `box` as an instance of the `Rectangle` class. The constructor is called with the value 10.0 passed into the `w` parameter and 12.0 passed into the `len` parameter. As a result, the object's `width` member variable will be assigned 10.0 and the `length` member variable will be assigned 12.0. This is illustrated in Figure 13-16.

**Figure 13-16**




Program 13-8 demonstrates the class. (This file is also stored in the Student Source Code Folder `Chapter 13\Rectangle Version 4`.)

**Program 13-8**

```
1  // This program calls the Rectangle class constructor.
2  #include <iostream>
3  #include <iomanip>
4  #include "Rectangle.h"
5  using namespace std;
6
7  int main()
8  {
9      double houseWidth,    // To hold the room width
10            houseLength;   // To hold the room length
11
12     // Get the width of the house.
13     cout << "In feet, how wide is your house? ";
14     cin >> houseWidth;
15
16     // Get the length of the house.
17     cout << "In feet, how long is your house? ";
18     cin >> houseLength;
19
20     // Create a Rectangle object.
21     Rectangle house(houseWidth, houseLength);
22
23     // Display the house's width, length, and area.
24     cout << setprecision(2) << fixed;
25     cout << "The house is " << house.getWidth()
26          << " feet wide.\n";
27     cout << "The house is " << house.getLength()
28          << " feet long.\n";
29     cout << "The house is " << house.getArea()
30          << " square feet in area.\n";
31     return 0;
32 }
```

**Program Output with Example Input Shown in Bold**

```
In feet, how wide is your house? 30 [Enter]
In feet, how long is your house? 60 [Enter]
The house is 30.00 feet wide.
The house is 60.00 feet long.
The house is 1800.00 square feet in area.
```

The statement in line 21 creates a `Rectangle` object, passing the values in `houseWidth` and `houseLength` as arguments.

The following code shows another example: the `Sale` class. This class might be used in a retail environment where sales transactions take place. An object of the `Sale` class represents the sale of an item. (This file is stored in the Student Source Code Folder `Chapter 13\Sale Version 1`.)

### Contents of `Sale.h` (Version 1)

```
1  // Specification file for the Sale class.
2  #ifndef SALE_H
3  #define SALE_H
4
5  class Sale
6  {
7  private:
8     double itemCost;    // Cost of the item
9     double taxRate;     // Sales tax rate
10 public:
11    Sale(double cost, double rate)
12       { itemCost = cost;
13         taxRate = rate; }
14
15    double getItemCost() const
16       { return itemCost; }
17
18    double getTaxRate() const
19       { return taxRate; }
20
21    double getTax() const
22       { return (itemCost * taxRate); }
23
24    double getTotal() const
25       { return (itemCost + getTax()); }
26 };
27 #endif
```

The `itemCost` member variable, declared in line 8, holds the selling price of the item. The `taxRate` member variable, declared in line 9, holds the sales tax rate. The constructor appears in lines 11 through 13. Notice that the constructor is written inline. It accepts two arguments, the item cost and the sales tax rate. These arguments are used to initialize the `itemCost` and `taxRate` member variables. The `getItemCost` member function, in lines 15 through 16, returns the value in `itemCost`, and the `getTaxRate` member function, in lines 18 through 19, returns the value in `taxRate`. The `getTax` member function, in lines 21 through 22, calculates and returns the amount of sales tax for the purchase. The `getTotal` member function, in lines 24 through 25, calculates and returns the total of the sale. The total is the item cost plus the sales tax. Program 13-9 demonstrates the class. (This file is stored in the Student Source Code Folder `Chapter 13\Sale Version 1`.)

### Program 13-9

```
1  // This program demonstrates passing an argument to a constructor.
2  #include <iostream>
3  #include <iomanip>
4  #include "Sale.h"
5  using namespace std;
6
```

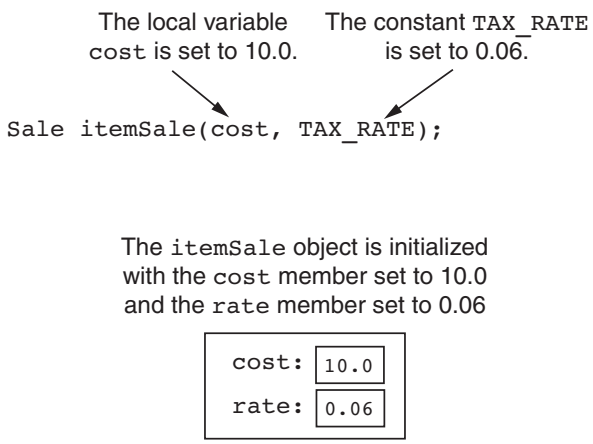
```
 7  int main()
 8  {
 9     const double TAX_RATE = 0.06;  // 6 percent sales tax rate
10     double cost;                   // To hold the item cost
11
12     // Get the cost of the item.
13     cout << "Enter the cost of the item: ";
14     cin >> cost;
15
16     // Create a Sale object for this transaction.
17     Sale itemSale(cost, TAX_RATE);
18
19     // Set numeric output formatting.
20     cout << fixed << showpoint << setprecision(2);
21
22     // Display the sales tax and total.
23     cout << "The amount of sales tax is $"
24          << itemSale.getTax() << endl;
25     cout << "The total of the sale is $";
26     cout << itemSale.getTotal() << endl;
27     return 0;
28  }
```

**Program Output with Example Input Shown in Bold**

```
Enter the cost of the item: 10.00 [Enter]
The amount of sales tax is $0.60
The total of the sale is $10.60
```

In the example run of the program the user enters 10.00 as the cost of the item. This value is stored in the local variable `cost`. In line 17 the `itemSale` object is created. The values of the `cost` variable and the `TAX_RATE` constant are passed as arguments to the constructor. As a result, the object's `cost` member variable is initialized with the value 10.0 and the `rate` member variable is initialized with the value 0.06. This is illustrated in Figure 13-17.

**Figure 13-17**



## Using Default Arguments with Constructors

Like other functions, constructors may have default arguments. Recall from Chapter 6 that default arguments are passed to parameters automatically if no argument is provided in the function call. The default value is listed in the parameter list of the function's declaration or the function header. The following code shows a modified version of the `Sale` class. This version's constructor uses a default argument for the tax rate. (This file is stored in the Student Source Code Folder `Chapter 13\Sale Version 2`.)

### Contents of `Sale.h` (Version 2)

```
1  // This version of the Sale class uses a default argument
2  // in the constructor.
3  #ifndef SALE_H
4  #define SALE_H
5
6  class Sale
7  {
8  private:
9     double itemCost;    // Cost of the item
10    double taxRate;     // Sales tax rate
11 public:
12    Sale(double cost, double rate = 0.05)
13       { itemCost = cost;
14         taxRate = rate; }
15
16    double getItemCost() const
17       { return itemCost; }
18
19    double getTaxRate() const
20       { return taxRate; }
21
22    double getTax() const
23       { return (itemCost * taxRate); }
24
25    double getTotal() const
26       { return (itemCost + getTax()); }
27 };
28 #endif
```

If an object of this `Sale` class is defined with only one argument (for the `cost` parameter) passed to the constructor, the default argument 0.05 will be provided for the `rate` parameter. This is demonstrated in Program 13-10. (This file is stored in the Student Source Code Folder `Chapter 13\Sale Version 2`.)

### Program 13-10

```
1  // This program uses a constructor's default argument.
2  #include <iostream>
3  #include <iomanip>
4  #include "Sale.h"
5  using namespace std;
6
```

```
 7  int main()
 8  {
 9       double cost;  // To hold the item cost
10
11       // Get the cost of the item.
12       cout << "Enter the cost of the item: ";
13       cin >> cost;
14
15       // Create a Sale object for this transaction.
16       // Specify the item cost, but use the default
17       // tax rate of 5 percent.
18       Sale itemSale(cost);
19
20       // Set numeric output formatting.
21       cout << fixed << showpoint << setprecision(2);
22
23       // Display the sales tax and total.
24       cout << "The amount of sales tax is $"
25            << itemSale.getTax() << endl;
26       cout << "The total of the sale is $";
27       cout << itemSale.getTotal() << endl;
28       return 0;
29  }
```

**Program Output with Example Input Shown in Bold**

```
Enter the cost of the item: 10.00 [Enter]
The amount of sales tax is $0.50
The total of the sale is $10.50
```

## More About the Default Constructor

It was mentioned earlier that when a constructor doesn't accept arguments, it is known as the default constructor. If a constructor has default arguments for all its parameters, it can be called with no explicit arguments. It then becomes the default constructor. For example, suppose the constructor for the `Sale` class had been written as the following:

```
Sale(double cost = 0.0, double rate = 0.05)
   { itemCost = cost;
     taxRate = rate; }
```

This constructor has default arguments for each of its parameters. As a result, the constructor can be called with no arguments, as shown here:

```
Sale itemSale;
```

This statement defines a `Sale` object. No arguments were passed to the constructor, so the default arguments for both parameters are used. Because this constructor can be called with no arguments, it is the default constructor.

### Classes with No Default Constructor

When all of a class's constructors require arguments, then the class does not have a default constructor. In such a case you must pass the required arguments to the constructor when creating an object. Otherwise, a compiler error will result.

## 13.9 Destructors

**CONCEPT:** A destructor is a member function that is automatically called when an object is destroyed.

Destructors are member functions with the same name as the class, preceded by a tilde character (~). For example, the destructor for the `Rectangle` class would be named `~Rectangle`.

Destructors are automatically called when an object is destroyed. In the same way that constructors set things up when an object is created, destructors perform shutdown procedures when the object goes out of existence. For example, a common use of destructors is to free memory that was dynamically allocated by the class object.

Program 13-11 shows a simple class with a constructor and a destructor. It illustrates when, during the program's execution, each is called.

**Program 13-11**

```
1  // This program demonstrates a destructor.
2  #include <iostream>
3  using namespace std;
4
5  class Demo
6  {
7  public:
8       Demo();      // Constructor
9       ~Demo();     // Destructor
10 };
11
12 Demo::Demo()
13 {
14      cout << "Welcome to the constructor!\n";
15 }
16
17 Demo::~Demo()
18 {
19      cout << "The destructor is now running.\n";
20 }
21
22 //*****************************************
23 // Function main.                         *
24 //*****************************************
25
26 int main()
27 {
28      Demo demoObject;  // Define a demo object;
29
30      cout << "This program demonstrates an object\n";
31      cout << "with a constructor and destructor.\n";
32      return 0;
33 }
```

**Program Output**

```
Welcome to the constructor!
This program demonstrates an object
with a constructor and destructor.
The destructor is now running.
```

The following code shows a more practical example of a class with a destructor. The `ContactInfo` class holds the following data about a contact:

- The contact's name
- The contact's phone number

The constructor accepts arguments for both items. The name and phone number are passed as a pointer to a C-string. Rather than storing the name and phone number in a `char` array with a fixed size, the constructor gets the length of the C-string and dynamically allocates just enough memory to hold it. The destructor frees the allocated memory when the object is destroyed. (This file is stored in the Student Source Code Folder `Chapter 13\ContactInfo Version 1`.)

### Contents of `ContactInfo.h` (Version 1)

```
1  // Specification file for the Contact class.
2  #ifndef CONTACTINFO_H
3  #define CONTACTINFO_H
4  #include <cstring>   // Needed for strlen and strcpy
5
6  // ContactInfo class declaration.
7  class ContactInfo
8  {
9  private:
10     char *name;   // The name
11     char *phone;  // The phone number
12 public:
13     // Constructor
14     ContactInfo(char *n, char *p)
15     { // Allocate just enough memory for the name and phone number.
16         name = new char[strlen(n) + 1];
17         phone = new char[strlen(p) + 1];
18
19         // Copy the name and phone number to the allocated memory.
20         strcpy(name, n);
21         strcpy(phone, p); }
22
23     // Destructor
24     ~ContactInfo()
25     { delete [] name;
26       delete [] phone; }
27
28     const char *getName() const
29     { return name; }
30
```

```
31      const char *getPhoneNumber() const
32      { return phone; }
33  };
34  #endif
```

Notice that the return type of the `getName` and `getPhoneNumber` functions in lines 28 through 32 is `const char *`. This means that each function returns a pointer to a constant `char`. This is a security measure. It prevents any code that calls the functions from changing the string that the pointer points to.

Program 13-12 demonstrates the class. (This file is also stored in the Student Source Code Folder `Chapter 13\ContactInfo Version 1`.)

**Program 13-12**

```
1  // This program demonstrates a class with a destructor.
2  #include <iostream>
3  #include "ContactInfo.h"
4  using namespace std;
5
6  int main()
7  {
8       // Define a ContactInfo object with the following data:
9       // Name: Kristen Lee Phone Number: 555-2021
10      ContactInfo entry("Kristen Lee", "555-2021");
11
12      // Display the object's data.
13      cout << "Name: " << entry.getName() << endl;
14      cout << "Phone Number: " << entry.getPhoneNumber() << endl;
15      return 0;
16 }
```

**Program Output**

```
Name: Kristen Lee
Phone Number: 555-2021
```

In addition to the fact that destructors are automatically called when an object is destroyed, the following points should be mentioned:

- Like constructors, destructors have no return type.
- Destructors cannot accept arguments, so they never have a parameter list.

## Destructors and Dynamically Allocated Class Objects

If a class object has been dynamically allocated by the `new` operator, its memory should be released when the object is no longer needed. For example, in the following code `objectPtr` is a pointer to a dynamically allocated `ContactInfo` class object.

```
// Define a ContactInfo pointer.
ContactInfo *objectPtr = nullptr;

// Dynamically create a ContactInfo object.
objectPtr = new ContactInfo("Kristen Lee", "555-2021");
```

The following statement shows the `delete` operator being used to destroy the dynamically created object.

```
delete objectPtr;
```

When the object pointed to by `objectPtr` is destroyed, its destructor is automatically called.

**NOTE:** If you have used a smart pointer such as `unique_ptr` (introduced in C++ 11) to allocate an object, the object will automatically be deleted, and its destructor will be called when the smart pointer goes out of scope. It is not necessary to use `delete` with a `unique_ptr`.

## Checkpoint

13.12 Briefly describe the purpose of a constructor.

13.13 Briefly describe the purpose of a destructor.

13.14 A member function that is never declared with a return data type, but that may have arguments is

A) The constructor
B) The destructor
C) Both the constructor and the destructor
D) Neither the constructor nor the destructor

13.15 A member function that is never declared with a return data type and can never have arguments is

A) The constructor
B) The destructor
C) Both the constructor and the destructor
D) Neither the constructor nor the destructor

13.16 Destructor function names always start with

A) A number
B) Tilde character (~)
C) A data type name
D) None of the above

13.17 A constructor that requires no arguments is called

A) A default constructor
B) An overloaded constructor
C) A null constructor
D) None of the above

13.18 TRUE or FALSE: Constructors are never declared with a return data type.

13.19 TRUE or FALSE: Destructors are never declared with a return type.

13.20 TRUE or FALSE: Destructors may take any number of arguments.

## 13.10 Overloading Constructors

**CONCEPT: A class can have more than one constructor.**

Recall from Chapter 6 that when two or more functions share the same name, the function is said to be overloaded. Multiple functions with the same name may exist in a C++ program, as long as their parameter lists are different.

A class's member functions may be overloaded, including the constructor. One constructor might take an integer argument, for example, while another constructor takes a `double`. There could even be a third constructor taking two integers. As long as each constructor takes a different list of parameters, the compiler can tell them apart. For example, the `string` class has several overloaded constructors. The following statement creates a `string` object with no arguments passed to the constructor:

```
string str;
```

This executes the `string` class's default constructor, which stores an empty string in the object. Another way to create a `string` object is to pass a string literal as an argument to the constructor, as shown here:

```
string str("Hello");
```

This executes an overloaded constructor, which stores the string "Hello" in the object.

Let's look at an example of how you can create overloaded constructors. The `InventoryItem` class holds the following data about an item that is stored in inventory:

- Item's description (a `string` object)
- Item's cost (a `double`)
- Number of units in inventory (an `int`)

The following code shows the class. To simplify the code, all the member functions are written inline. (This file is stored in the Student Source Code Folder `Chapter 13\InventoryItem`.)

### Contents of `InventoryItem.h`

```
1  // This class has overloaded constructors.
2  #ifndef INVENTORYITEM_H
3  #define INVENTORYITEM_H
4  #include <string>
5  using namespace std;
6
7  class InventoryItem
8  {
9  private:
10    string description; // The item description
11    double cost;        // The item cost
12    int units;          // Number of units on hand
13 public:
14    // Constructor #1
15    InventoryItem()
16       { // Initialize description, cost, and units.
17         description = "";
```

```
18           cost = 0.0;
19           units = 0; }
20
21     // Constructor #2
22     InventoryItem(string desc)
23        { // Assign the value to description.
24          description = desc;
25
26          // Initialize cost and units.
27          cost = 0.0;
28          units = 0; }
29
30     // Constructor #3
31     InventoryItem(string desc, double c, int u)
32        { // Assign values to description, cost, and units.
33          description = desc;
34          cost = c;
35          units = u; }
36
37     // Mutator functions
38     void setDescription(string d)
39        { description = d; }
40
41     void setCost(double c)
42        { cost = c; }
43
44     void setUnits(int u)
45        { units = u; }
46
47     // Accessor functions
48     string getDescription() const
49        { return description; }
50
51     double getCost() const
52        { return cost; }
53
54     int getUnits() const
55        { return units; }
56  };
57  #endif
```

The first constructor appears in lines 15 through 19. It takes no arguments, so it is the default constructor. It initializes the `description` variable to an empty string. The `cost` and `units` variables are initialized to 0.

The second constructor appears in lines 22 through 28. This constructor accepts only one argument, the item description. The `cost` and `units` variables are initialized to 0.

The third constructor appears in lines 31 through 35. This constructor accepts arguments for the description, cost, and units.

The mutator functions set values for `description`, `cost`, and `units`. Program 13-13 demonstrates the class. (This file is also stored in the Student Source Code Folder `Chapter 13\InventoryItem`.)

**Program 13-13**

```
// This program demonstrates a class with overloaded constructors.
#include <iostream>
#include <iomanip>
#include "InventoryItem.h"

int main()
{
     // Create an InventoryItem object and call
     // the default constructor.
     InventoryItem item1;
     item1.setDescription("Hammer"); // Set the description
     item1.setCost(6.95);            // Set the cost
     item1.setUnits(12);             // Set the units

     // Create an InventoryItem object and call
     // constructor #2.
     InventoryItem item2("Pliers");

     // Create an InventoryItem object and call
     // constructor #3.
     InventoryItem item3("Wrench", 8.75, 20);

     cout << "The following items are in inventory:\n";
     cout << setprecision(2) << fixed << showpoint;

     // Display the data for item 1.
     cout << "Description: " << item1.getDescription() << endl;
     cout << "Cost: $" << item1.getCost() << endl;
     cout << "Units on Hand: " << item1.getUnits() << endl << endl;

     // Display the data for item 2.
     cout << "Description: " << item2.getDescription() << endl;
     cout << "Cost: $" << item2.getCost() << endl;
     cout << "Units on Hand: " << item2.getUnits() << endl << endl;

     // Display the data for item 3.
     cout << "Description: " << item3.getDescription() << endl;
     cout << "Cost: $" << item3.getCost() << endl;
     cout << "Units on Hand: " << item3.getUnits() << endl;
     return 0;
}
```

**Program Output**

```
The following items are in inventory:
Description: Hammer
Cost: $6.95
Units on Hand: 12

Description: Pliers
Cost: $0.00
Units on Hand: 0
```

```
Description: Wrench
Cost: $8.75
Units on Hand: 20
```

### Only One Default Constructor and One Destructor

When an object is defined without an argument list for its constructor, the compiler automatically calls the default constructor. For this reason, a class may have only one default constructor. If there were more than one constructor that could be called without an argument, the compiler would not know which one to call by default.

Remember, a constructor whose parameters all have a default argument is considered a default constructor. It would be an error to create a constructor that accepts no parameters along with another constructor that has default arguments for all its parameters. In such a case the compiler would not be able to resolve which constructor to execute.

Classes may also only have one destructor. Because destructors take no arguments, the compiler has no way to distinguish different destructors.

### Other Overloaded Member Functions

Member functions other than constructors can also be overloaded. This can be useful because sometimes you need several different ways to perform the same operation. For example, in the `InventoryItem` class we could have overloaded the `setCost` function as shown here:

```
void setCost(double c)
   { cost = c; }
void setCost(string c)
    { cost = atof(c.c_str()); }
```

The first version of the function accepts a `double` argument and assigns it to `cost`. The second version of the function accepts a `string` object. This could be used where you have the cost of the item stored in a `string` object. The function calls the `atof` function to convert the string to a `double` and assigns its value to `cost`.

## 13.11 Private Member Functions

**CONCEPT:** A private member function may only be called from a function that is a member of the same class.

Sometimes a class will contain one or more member functions that are necessary for internal processing, but should not be called by code outside the class. For example, a class might have a member function that performs a calculation only when a value is stored in a particular member variable and should not be performed at any other time. That function should not be directly accessible by code outside the class because it might get called at the wrong time. In this case, the member function should be declared `private`. When a member function is declared `private`, it may only be called internally.

For example, consider the following version of the `ContactInfo` class. (This file is stored in the Student Source Code Folder `Chapter 13\ContactInfo Version 2`.)

### Contents of `ContactInfo.h` (Version 2)

```
1  // Contact class specification file (version 2)
2  #ifndef CONTACTINFO_H
3  #define CONTACTINFO_H
4  #include <cstring>   // Needed for strlen and strcpy
5
6  // ContactInfo class declaration.
7  class ContactInfo
8  {
9  private:
10    char *name;   // The contact's name
11    char *phone;  // The contact's phone number
12
13    // Private member function: initName
14    // This function initializes the name attribute.
15    void initName(char *n)
16    { name = new char[strlen(n) + 1];
17      strcpy(name, n); }
18
19    // Private member function: initPhone
20    // This function initializes the phone attribute.
21    void initPhone(char *p)
22    { phone = new char[strlen(p) + 1];
23      strcpy(phone, p); }
24 public:
25    // Constructor
26    ContactInfo(char *n, char *p)
27    { // Initialize the name attribute.
28      initName(n);
29
30      // Initialize the phone attribute.
31      initPhone(n); }
32
33    // Destructor
34    ~ContactInfo()
35    { delete [] name;
36      delete [] phone; }
37
38    const char *getName() const
39    { return name; }
40
41    const char *getPhoneNumber() const
42    { return phone; }
43 };
44 #endif
```

In this version of the class, the logic in the constructor is modularized. It calls two private member functions, `initName` and `initPhone`. The `initName` function allocates memory for the `name` attribute and initializes it with the value pointed to by the `n` parameter. The

initPhone function allocates memory for the phone attribute and initializes it with the value pointed to by the p parameter. These functions are private because they should be called only from the constructor. If they were ever called by code outside the class, they would change the values of the name and phone pointers without deallocating the memory that they currently point to.

## 13.12 Arrays of Objects

**CONCEPT:** You may define and work with arrays of class objects.

As with any other data type in C++, you can define arrays of class objects. An array of InventoryItem objects could be created to represent a business's inventory records. Here is an example of such a definition:

```
const int ARRAY_SIZE = 40;
InventoryItem inventory[ARRAY_SIZE];
```

This statement defines an array of 40 InventoryItem objects. The name of the array is inventory, and the default constructor is called for each object in the array.

If you wish to define an array of objects and call a constructor that requires arguments, you must specify the arguments for each object individually in an initializer list. Here is an example:

```
InventoryItem inventory[] = {"Hammer", "Wrench", "Pliers"};
```

The compiler treats each item in the initializer list as an argument for an array element's constructor. Recall that the second constructor in the InventoryItem class declaration takes the item description as an argument. So, this statement defines an array of three objects and calls that constructor for each object. The constructor for inventory[0] is called with "Hammer" as its argument, the constructor for inventory[1] is called with "Wrench" as its argument, and the constructor for inventory[2] is called with "Pliers" as its argument.

**WARNING!** If the class does not have a default constructor you must provide an initializer for each object in the array.

If a constructor requires more than one argument, the initializer must take the form of a function call. For example, look at the following definition statement.

```
InventoryItem inventory[] = { InventoryItem("Hammer", 6.95, 12),
                              InventoryItem("Wrench", 8.75, 20),
                              InventoryItem("Pliers", 3.75, 10) };
```

This statement calls the third constructor in the InventoryItem class declaration for each object in the inventory array.

It isn't necessary to call the same constructor for each object in an array. For example, look at the following statement.

```
InventoryItem inventory[] = { "Hammer",
                              InventoryItem("Wrench", 8.75, 20),
                              "Pliers" };
```

This statement calls the second constructor for `inventory[0]` and `inventory[2]`, and calls the third constructor for `inventory[1]`.

If you do not provide an initializer for all of the objects in an array, the default constructor will be called for each object that does not have an initializer. For example, the following statement defines an array of three objects, but only provides initializers for the first two. The default constructor is called for the third object.

```
const int SIZE = 3;
InventoryItem inventory [SIZE] = { "Hammer",
                                   InventoryItem("Wrench", 8.75, 20) };
```

In summary, if you use an initializer list for class object arrays, there are three things to remember:

- If there is no default constructor you must furnish an initializer for each object in the array.
- If there are fewer initializers in the list than objects in the array, the default constructor will be called for all the remaining objects.
- If a constructor requires more than one argument, the initializer takes the form of a constructor function call.

## Accessing Members of Objects in an Array

Objects in an array are accessed with subscripts, just like any other data type in an array. For example, to call the `setUnits` member function of `inventory[2]`, the following statement could be used:

```
inventory[2].setUnits(30);
```

This statement sets the `units` variable of `inventory[2]` to the value 30. Program 13-14 shows an array of `InventoryItem` objects being used in a complete program. (This file is stored in the Student Source Code Folder `Chapter 13\InventoryItem`.)

**Program 13-14**

```
1  // This program demonstrates an array of class objects.
2  #include <iostream>
3  #include <iomanip>
4  #include "InventoryItem.h"
5  using namespace std;
6
7  int main()
8  {
9     const int NUM_ITEMS = 5;
10    InventoryItem inventory[NUM_ITEMS] = {
11                  InventoryItem("Hammer", 6.95, 12),
12                  InventoryItem("Wrench", 8.75, 20),
13                  InventoryItem("Pliers", 3.75, 10),
14                  InventoryItem("Ratchet", 7.95, 14),
15                  InventoryItem("Screwdriver", 2.50, 22) };
16
```

```
17      cout << setw(14) <<"Inventory Item"
18           << setw(8) << "Cost" << setw(8)
19           << setw(16) << "Units on Hand\n";
20      cout << "---------------------------------------\n";
21
22      for (int i = 0; i < NUM_ITEMS; i++)
23      {
24         cout << setw(14) << inventory[i].getDescription();
25         cout << setw(8) << inventory[i].getCost();
26         cout << setw(7) << inventory[i].getUnits() << endl;
27      }
28
29      return 0;
30   }
```

**Program Output**

```
Inventory Item    Cost   Units on Hand
---------------------------------------
         Hammer    6.95         12
         Wrench    8.75         20
         Pliers    3.75         10
        Ratchet    7.95         14
    Screwdriver     2.5         22
```

## Checkpoint

13.21 What will the following program display on the screen?

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

class Tank
{
private:
    int gallons;
public:
    Tank()
        { gallons = 50; }
    Tank(int gal)
        { gallons = gal; }
    int getGallons()
        { return gallons; }
};

int main()
{
    Tank storage[3] = { 10, 20 };
    for (int index = 0; index < 3; index++)
        cout << storage[index].getGallons() << endl;
    return 0;
}
```

13.22 What will the following program display on the screen?

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

class Package
{
private:
    int value;
public:
    Package()
        { value = 7; cout << value << endl; }
    Package(int v)
        { value = v; cout << value << endl; }
    ~Package()
        { cout << value << endl; }
};

int main()
{
    Package obj1(4);
    Package obj2();
    Package obj3(2);
    return 0;
}
```

13.23 In your answer for Checkpoint 13.22 indicate for each line of output whether the line is displayed by constructor #1, constructor #2, or the destructor.

13.24 Why would a member function be declared private?

13.25 Define an array of three `InventoryItem` objects.

13.26 Complete the following program so it defines an array of `Yard` objects. The program should use a loop to ask the user for the length and width of each `Yard`.

```
#include <iostream>
using namespace std;
class Yard
{
private:
    int length, width;
public:
    Yard()
        { length = 0; width = 0; }
    setLength(int len)
        { length = len; }
    setWidth(int w)
        { width = w; }
};

int main()
{
    // Finish this program
}
```

# 13.13 Focus on Problem Solving and Program Design: An OOP Case Study

You are a programmer for the Home Software Company. You have been assigned to develop a class that models the basic workings of a bank account. The class should perform the following tasks:

- Save the account balance.
- Save the number of transactions performed on the account.
- Allow deposits to be made to the account.
- Allow withdrawals to be taken from the account.
- Calculate interest for the period.
- Report the current account balance at any time.
- Report the current number of transactions at any time.

## Private Member Variables

Table 13-4 lists the private member variables needed by the class.

**Table 13-4**

| Variable | Description |
|---|---|
| `balance` | A `double` that holds the current account balance. |
| `interestRate` | A `double` that holds the interest rate for the period. |
| `interest` | A `double` that holds the interest earned for the current period. |
| `transactions` | An integer that holds the current number of transactions. |

## Public Member Functions

Table 13-5 lists the public member functions needed by the class.

**Table 13-5**

| Function | Description |
|---|---|
| Constructor | Takes arguments to be initially stored in the `balance` and `interestRate` members. The default value for the balance is zero and the default value for the interest rate is 0.045. |
| `setInterestRate` | Takes a `double` argument which is stored in the `interestRate` member. |
| `makeDeposit` | Takes a `double` argument, which is the amount of the deposit. This argument is added to `balance`. |
| `withdraw` | Takes a `double` argument which is the amount of the withdrawal. This value is subtracted from the balance, unless the withdrawal amount is greater than the balance. If this happens, the function reports an error. |

(*continued*)

**Table 13-5** *(continued)*

| Function | Description |
|---|---|
| `calcInterest` | Takes no arguments. This function calculates the amount of interest for the current period, stores this value in the `interest` member, and then adds it to the `balance` member. |
| `getInterestRate` | Returns the current interest rate (stored in the `interestRate` member). |
| `getBalance` | Returns the current balance (stored in the `balance` member). |
| `getInterest` | Returns the interest earned for the current period (stored in the `interest` member). |
| `getTransactions` | Returns the number of transactions for the current period (stored in the `transactions` member). |

## The Class Declaration

The following listing shows the class declaration.

### Contents of `Account.h`

```
1  // Specification file for the Account class.
2  #ifndef ACCOUNT_H
3  #define ACCOUNT_H
4
5  class Account
6  {
7  private:
8     double balance;         // Account balance
9     double interestRate;    // Interest rate for the period
10    double interest;        // Interest earned for the period
11    int transactions;       // Number of transactions
12 public:
13    Account(double iRate = 0.045, double bal = 0)
14       { balance = bal;
15         interestRate = iRate;
16         interest = 0;
17         transactions = 0; }
18
19    void setInterestRate(double iRate)
20       { interestRate = iRate; }
21
22    void makeDeposit(double amount)
23       { balance += amount; transactions++; }
24
25    void withdraw(double amount); // Defined in Account.cpp
26
27    void calcInterest()
28       { interest = balance * interestRate; balance += interest; }
29
30    double getInterestRate() const
31       { return interestRate; }
32
```

```
33      double getBalance() const
34          { return balance; }
35
36      double getInterest() const
37          { return interest; }
38
39      int getTransactions() const
40          { return transactions; }
41  };
42  #endif
```

## The `withdraw` Member Function

The only member function not written `inline` in the class declaration is `withdraw`. The purpose of that function is to subtract the amount of a withdrawal from the `balance` member. If the amount to be withdrawn is greater than the current balance, however, no withdrawal is made. The function returns true if the withdrawal is made, or false if there is not enough in the account.

### Contents of `Account.cpp`

```
1   // Implementation file for the Account class.
2   #include "Account.h"
3
4   bool Account::withdraw(double amount)
5   {
6       if (balance < amount)
7           return false; // Not enough in the account
8       else
9       {
10          balance -= amount;
11          transactions++;
12          return true;
13      }
14  }
```

## The Class's Interface

The `balance`, `interestRate`, `interest`, and `transactions` member variables are private, so they are hidden from the world outside the class. The reason is that a programmer with direct access to these variables might unknowingly commit any of the following errors:

- A deposit or withdrawal might be made without the `transactions` member being incremented.
- A withdrawal might be made for more than is in the account. This will cause the `balance` member to have a negative value.
- The interest rate might be calculated and the `balance` member adjusted, but the amount of interest might not get recorded in the `interest` member.
- The wrong interest rate might be used.

Because of the potential for these errors, the class contains public member functions that ensure the proper steps are taken when the account is manipulated.

## Implementing the Class

Program 13-15 shows an implementation of the `Account` class. It presents a menu for displaying a savings account's balance, number of transactions, and interest earned. It also allows the user to deposit an amount into the account, make a withdrawal from the account, and calculate the interest earned for the current period.

**Program 13-15**

```
// This program demonstrates the Account class.
#include <iostream>
#include <cctype>
#include <iomanip>
#include "Account.h"
using namespace std;

// Function prototypes
void displayMenu();
void makeDeposit(Account &);
void withdraw(Account &);

int main()
{
    Account savings;  // Savings account object
    char choice;      // Menu selection

    // Set numeric output formatting.
    cout << fixed << showpoint << setprecision(2);

    do
    {
        // Display the menu and get a valid selection.
        displayMenu();
        cin >> choice;
        while (toupper(choice) < 'A' || toupper(choice) > 'G')
        {
            cout << "Please make a choice in the range "
                 << "of A through G:";
            cin >> choice;
        }

        // Process the user's menu selection.
        switch(choice)
        {
        case 'a':
        case 'A': cout << "The current balance is $";
                  cout << savings.getBalance() << endl;
                  break;
        case 'b':
        case 'B': cout << "There have been ";
                  cout << savings.getTransactions()
                       << " transactions.\n";
```

```
                    break;
        case 'c':
        case 'C': cout << "Interest earned for this period: $";
                  cout << savings.getInterest() << endl;
                  break;
        case 'd':
        case 'D': makeDeposit(savings);
                  break;
        case 'e':
        case 'E': withdraw(savings);
                  break;
        case 'f':
        case 'F': savings.calcInterest();
                  cout << "Interest added.\n";
        }
    } while (toupper(choice) != 'G');

    return 0;
}

//*****************************************************
// Definition of function displayMenu. This function  *
// displays the user's menu on the screen.            *
//*****************************************************

void displayMenu()
{
    cout << "\n                MENU\n";
    cout << "----------------------------------------\n";
    cout << "A) Display the account balance\n";
    cout << "B) Display the number of transactions\n";
    cout << "C) Display interest earned for this period\n";
    cout << "D) Make a deposit\n";
    cout << "E) Make a withdrawal\n";
    cout << "F) Add interest for this period\n";
    cout << "G) Exit the program\n\n";
    cout << "Enter your choice: ";
}

//************************************************************
// Definition of function makeDeposit. This function accepts  *
// a reference to an Account object. The user is prompted for *
// the dollar amount of the deposit, and the makeDeposit      *
// member of the Account object is then called.               *
//************************************************************

void makeDeposit(Account &acnt)
{
    double dollars;

    cout << "Enter the amount of the deposit: ";
    cin >> dollars;
```

*(program continues)*

**Program 13-15** *(continued)*

```
 96      cin.ignore();
 97      acnt.makeDeposit(dollars);
 98  }
 99
100  //*************************************************************
101  // Definition of function withdraw. This function accepts     *
102  // a reference to an Account object. The user is prompted for *
103  // the dollar amount of the withdrawal, and the withdraw      *
104  // member of the Account object is then called.               *
105  //*************************************************************
106
107  void withdraw(Account &acnt)
108  {
109      double dollars;
110
111      cout << "Enter the amount of the withdrawal: ";
112      cin >> dollars;
113      cin.ignore();
114      if (!acnt.withdraw(dollars))
115          cout << "ERROR: Withdrawal amount too large.\n\n";
116  }
```

**Program Output with Example Input Shown in Bold**

```
                    MENU
-----------------------------------------
A) Display the account balance
B) Display the number of transactions
C) Display interest earned for this period
D) Make a deposit
E) Make a withdrawal
F) Add interest for this period
G) Exit the program

Enter your choice: d [Enter]
Enter the amount of the deposit: 500 [Enter]

                    MENU
-----------------------------------------
A) Display the account balance
B) Display the number of transactions
C) Display interest earned for this period
D) Make a deposit
E) Make a withdrawal
F) Add interest for this period
G) Exit the program

Enter your choice: a [Enter]
The current balance is $500.00
```

```
                    MENU
-----------------------------------------
A) Display the account balance
B) Display the number of transactions
C) Display interest earned for this period
D) Make a deposit
E) Make a withdrawal
F) Add interest for this period
G) Exit the program

Enter your choice: e [Enter]
Enter the amount of the withdrawal: 700 [Enter]
ERROR: Withdrawal amount too large.

                    MENU
-----------------------------------------
A) Display the account balance
B) Display the number of transactions
C) Display interest earned for this period
D) Make a deposit
E) Make a withdrawal
F) Add interest for this period
G) Exit the program

Enter your choice: e [Enter]
Enter the amount of the withdrawal: 200 [Enter]

                  MENU
-----------------------------------------
A) Display the account balance
B) Display the number of transactions
C) Display interest earned for this period
D) Make a deposit
E) Make a withdrawal
F) Add interest for this period
G) Exit the program

Enter your choice: f [Enter]
Interest added.

                MENU
-----------------------------------------
A) Display the account balance
B) Display the number of transactions
C) Display interest earned for this period
D) Make a deposit
E) Make a withdrawal
F) Add interest for this period
G) Exit the program

Enter your choice: a [Enter]
The current balance is $313.50
```

*(program ouput continues)*

**Program 13-15** *(continued)*

```
               MENU
----------------------------------------
A) Display the account balance
B) Display the number of transactions
C) Display interest earned for this period
D) Make a deposit
E) Make a withdrawal
F) Add interest for this period
G) Exit the program
Enter your choice: g [Enter]
```

## 13.14 Focus on Object-Oriented Programming: Simulating Dice with Objects

Dice traditionally have six sides, representing the values 1 to 6. Some games, however, use specialized dice that have a different number of sides. For example, the fantasy role-playing game *Dungeons and Dragons*® uses dice with four, six, eight, ten, twelve, and twenty sides.

Suppose you are writing a program that needs to roll simulated dice with various numbers of sides. A simple approach would be to write a `Die` class with a constructor that accepts the number of sides as an argument. The class would also have appropriate methods for rolling the die and getting the die's value. An example of such a class follows. (These files are stored in the Student Source Code Folder `Chapter 13\Dice`.)

### Contents of `Die.h`

```
1  // Specification file for the Die class
2  #ifndef DIE_H
3  #define DIE_H
4
5  class Die
6  {
7  private:
8       int sides;    // Number of sides
9       int value;    // The die's value
10
11 public:
12      Die(int = 6);     // Constructor
13      void roll();      // Rolls the die
14      int getSides();   // Returns the number of sides
15      int getValue();   // Returns the die's value
16 };
17 #endif
```

### Contents of `Die.cpp`

```
1  // Implememtation file for the Die class
2  #include <cstdlib> // For rand and srand
3  #include <ctime>   // For the time function
```

```
4  #include "Die.h"
5  using namespace std;
6
7  //*******************************************************
8  // The constructor accepts an argument for the number  *
9  // of sides for the die, and performs a roll.          *
10 //*******************************************************
11 Die::Die(int numSides)
12 {
13     // Get the system time.
14     unsigned seed = time(0);
15
16     // Seed the random number generator.
17     srand(seed);
18
19     // Set the number of sides.
20     sides = numSides;
21
22     // Perform an initial roll.
23     roll();
24 }
25
26 //*******************************************************
27 // The roll member function simulates the rolling of   *
28 // the die.                                            *
29 //*******************************************************
30 void Die::roll()
31 {
32   // Constant for the minimum die value
33   const int MIN_VALUE = 1; // Minimum die value
34
35   // Get a random value for the die.
36   value = (rand() % (sides - MIN_VALUE + 1)) + MIN_VALUE;
37 }
38
39 //*******************************************************
40 // The getSides member function returns the number of  *
41 // for this die.                                       *
42 //*******************************************************
43 int Die::getSides()
44 {
45   return sides;
46 }
47
48 //*******************************************************
49 // The getValue member function returns the die's value.*
50 //*******************************************************
51 int Die::getValue()
52 {
53   return value;
54 }
```

Here is a synopsis of the class members:

| | |
|---|---|
| `sides` | Declared in line 8 of Die.h. This is an `int` member variable that will hold the number of sides for the die. |
| `value` | Declared in line 9 of Die.h. This is an `int` member variable that will hold the die's value once it has been rolled. |
| Constructor | The constructor (lines 11 through 24 in Die.cpp) has a parameter for the number of sides. Notice in the constructor's prototype (line 12 in Die.h) that the parameter's default value is 6. When the constructor executes, line 14 gets the system time and line 17 uses that value to seed the random number generator. Line 20 assigns the constructor's parameter to the `sides` member variable, and line 23 calls the `roll` member function, which simulates the rolling of the die. |
| `roll` | The `roll` member function (lines 30 through 37 in Die.cpp) simulates the rolling of the die. The `MIN_VALUE` constant, defined in line 33, is the minimum value for the die. Line 36 generates a random number within the appropriate range for this particular die and assigns it to the `value` member variable. |
| `getSides` | The `getSides` member function (lines 43 through 46) returns the `sides` member variable. |
| `getValue` | The `getValue` member function (lines 51 through 54) returns the `value` member variable. |

The code in Program 13-16 demonstrates the class. It creates two instances of the `Die` class: one with six sides and the other with twelve sides. It then simulates five rolls of the dice.

**Program 13-16**

```
1  // This program simulates the rolling of dice.
2  #include <iostream>
3  #include "Die.h"
4  using namespace std;
5
6  int main()
7  {
8      const int DIE1_SIDES = 6;      // Number of sides for die #1
9      const int DIE2_SIDES = 12;     // Number of sides for die #2
10     const int MAX_ROLLS = 5;       // Number of times to roll
11
12     // Create two instances of the Die class.
13     Die die1(DIE1_SIDES);
14     Die die2(DIE2_SIDES);
15
16     // Display the initial state of the dice.
17     cout << "This simulates the rolling of a "
18          << die1.getSides() << " sided die and a "
19          << die2.getSides() << " sided die.\n";
20
```

```
21      cout << "Initial value of the dice:\n";
22      cout << die1.getValue() << " "
23           << die2.getValue() << endl;
24
25      // Roll the dice five times.
26      cout << "Rolling the dice " << MAX_ROLLS
27           << " times.\n";
28      for (int count = 0; count < MAX_ROLLS; count++)
29      {
30          // Roll the dice.
31          die1.roll();
32          die2.roll();
33
34          // Display the values of the dice.
35          cout << die1.getValue() << " "
36               << die2.getValue() << endl;
37      }
38      return 0;
39  }
```

**Program Output**

```
This simulates the rolling of a 6 sided die and a 12 sided die.
Initial value of the dice:
1 7
Rolling the dice 5 times.
6 2
3 5
4 2
5 11
4 7
```

Let's take a closer look at the program:

**Lines 8 to 10:** These statements declare three constants. `DIE1_SIDES` is the number of sides for the first die (6), `DIE2_SIDES` is the number of sides for the second die (12), and `MAX_ROLLS` is the number of times to roll the die (5).

**Lines 13 to 14:** These statements create two instances of the `Die` class. Notice that `DIE1_SIDES`, which is 6, is passed to the constructor in line 13, and `DIE2_SIDES`, which is 12, is passed to the constructor in line 14. As a result, `die1` will reference a `Die` object with six sides, and `die2` will reference a `Die` object with twelve sides.

**Lines 22 to 23:** This statement displays the initial value of both `Die` objects. (Recall that the `Die` class constructor performs an initial roll of the die.)

**Lines 28 to 37:** This `for` loop iterates five times. Each time the loop iterates, line 31 calls the `die1` object's `roll` method, and line 32 calls the `die2` object's `roll` method. Lines 35 and 36 display the values of both dice.

## 13.15 Focus on Object-Oriented Programming: Creating an Abstract Array Data Type

**CONCEPT:** The absence of array bounds checking in C++ is a source of potential hazard. In this section we examine a simple integer list class that provides bounds checking.

One of the benefits of object-oriented programming is the ability to create abstract data types that are improvements on built-in data types. As you know, arrays provide no bounds checking in C++. You can, however, create a class that has array-like characteristics and performs bounds checking. For example, look at the following `IntegerList` class.

### Contents of `IntegerList.h`

```
1  // Specification file for the IntegerList class.
2  #ifndef INTEGERLIST_H
3  #define INTEGERLIST_H
4
5  class IntegerList
6  {
7  private:
8     int *list;                 // Pointer to the array.
9     int numElements;           // Number of elements.
10    bool isValid(int);         // Validates subscripts.
11 public:
12    IntegerList(int);           // Constructor
13    ~IntegerList();             // Destructor
14    void setElement(int, int);  // Sets an element to a value.
15    void getElement(int, int&); // Returns an element.
16 };
17 #endif
```

### Contents of `IntegerList.cpp`

```
1  // Implementation file for the IntegerList class.
2  #include <iostream>
3  #include <cstdlib>
4  #include "IntegerList.h"
5  using namespace std;
6
7  //*********************************************************
8  // The constructor sets each element to zero.             *
9  //*********************************************************
10
11 IntegerList::IntegerList(int size)
12 {
13    list = new int[size];
14    numElements = size;
15    for (int ndx = 0; ndx < size; ndx++)
16       list[ndx] = 0;
17 }
```

```
18
19  //*********************************************************
20  // The destructor releases allocated memory.              *
21  //*********************************************************
22
23  IntegerList::~IntegerList()
24  {
25     delete [] list;
26  }
27
28  //*********************************************************
29  // isValid member function.                               *
30  // This private member function returns true if the argument *
31  // is a valid subscript, or false otherwise.              *
32  //*********************************************************
33
34  bool IntegerList::isValid(int element) const
35  {
36     bool status;
37
38     if (element < 0 || element >= numElements)
39        status = false;
40     else
41        status = true;
42     return status;
43  }
44
45  //*********************************************************
46  // setElement member function.                            *
47  // Stores a value in a specific element of the list. If an  *
48  // invalid subscript is passed, the program aborts.       *
49  //*********************************************************
50
51  void IntegerList::setElement(int element, int value)
52  {
53     if (isValid(element))
54        list[element] = value;
55     else
56     {
57        cout << "Error: Invalid subscript\n";
58        exit(EXIT_FAILURE);
59     }
60  }
61
62  //*********************************************************
63  // getElement member function.                            *
64  // Returns the value stored at the specified element.     *
65  // If an invalid subscript is passed, the program aborts.  *
66  //*********************************************************
67
68  int IntegerList::getElement(int element) const
69  {
```

```
70      if (isValid(element))
71         return list[element];
72      else
73      {
74         cout << "Error: Invalid subscript\n";
75         exit(EXIT_FAILURE);
76      }
77   }
```

The `IntegerList` class allows you to store and retrieve numbers in a dynamically allocated array of integers. Here is a synopsis of the members.

| | |
|---|---|
| `list` | A pointer to an `int`. This member points to the dynamically allocated array of integers. |
| `numElements` | An integer that holds the number of elements in the dynamically allocated array. |
| `isValid` | This function validates a subscript into the array. It accepts a subscript value as an argument and returns boolean `true` if the subscript is in the range 0 through `numElements - 1`. If the value is outside that range, boolean `false` is returned. |
| Constructor | The class constructor accepts an `int` argument that is the number of elements to allocate for the array. The array is allocated, and each element is set to zero. |
| `setElement` | The `setElement` member function sets a specific element of the `list` array to a value. The first argument is the element subscript, and the second argument is the value to be stored in that element. The function uses `isValid` to validate the subscript. If an invalid subscript is passed to the function, the program aborts. |
| `getElement` | The `getElement` member function retrieves a value from a specific element in the `list` array. The argument is the subscript of the element whose value is to be retrieved. The function uses `isValid` to validate the subscript. If the subscript is valid, the value is returned. If the subscript is invalid, the program aborts. |

Program 13-17 demonstrates the class. A loop uses the `setElement` member to fill the array with 9s and prints an asterisk on the screen each time a 9 is successfully stored. Then another loop uses the `getElement` member to retrieve the values from the array and prints them on the screen. Finally, a statement uses the `setElement` member to demonstrate the subscript validation by attempting to store a value in element 50.

**Program 13-17**

```
1  // This program demonstrates the IntegerList class.
2  #include <iostream>
3  #include "IntegerList.h"
4  using namespace std;
5
6  int main()
```

```
{
     const int SIZE = 20;
     IntegerList numbers(SIZE);
     int val, x;

     // Store 9s in the list and display an asterisk
     // each time a 9 is successfully stored.
     for (x = 0; x < SIZE; x++)
     {
          numbers.setElement(x, 9);
          cout << "* ";
     }
     cout << endl;

     // Display the 9s.
     for (x = 0; x < SIZE; x++)
     {
          val = numbers.getElement(x);
          cout << val << " ";
     }
     cout << endl;

     // Attempt to store a value outside the list's bounds.
     numbers.setElement(50, 9);

     // Will this message display?
     cout << "Element 50 successfully set.\n";
     return 0;
}
```

**Program Output**

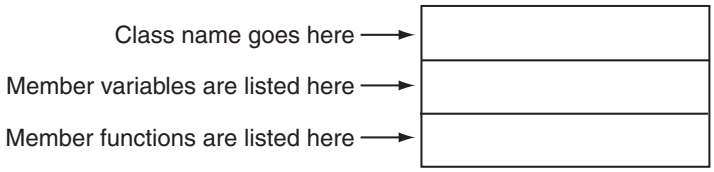
```
* * * * * * * * * * * * * * * * * * * *
9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
Error: Invalid subscript
```

## 13.16 Focus on Object-Oriented Design: The Unified Modeling Language (UML)

**CONCEPT:** **The Unified Modeling Language provides a standard method for graphically depicting an object-oriented system.**

When designing a class it is often helpful to draw a UML diagram. *UML* stands for *Unified Modeling Language*. The UML provides a set of standard diagrams for graphically depicting object-oriented systems. Figure 13-18 shows the general layout of a UML diagram for a class. Notice that the diagram is a box that is divided into three sections. The top section is where you write the name of the class. The middle section holds a list of the class's member variables. The bottom section holds a list of the class's member functions.

**Figure 13-18**




Earlier in this chapter you studied a `Rectangle` class that could be used in a program that works with rectangles. The first version of the `Rectangle` class that you studied had the following member variables:

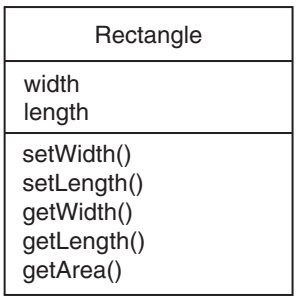
- `width`
- `length`

The class also had the following member functions:

- `setWidth`
- `setLength`
- `getWidth`
- `getLength`
- `getArea`

From this information alone we can construct a simple UML diagram for the class, as shown in Figure 13-19.

**Figure 13-19**

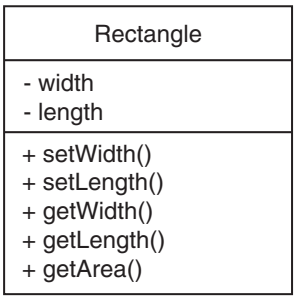



The UML diagram in Figure 13-19 tells us the name of the class, the names of the member variables, and the names of the member functions. The UML diagram in Figure 13-19 does not convey many of the class details, however, such as access specification, member variable data types, parameter data types, and function return types. The UML provides optional notation for these types of details.

## Showing Access Specification in UML Diagrams

The UML diagram in Figure 13-19 lists all of the members of the `Rectangle` class but does not indicate which members are private and which are public. In a UML diagram you may optionally place a – character before a member name to indicate that it is private, or a + character to indicate that it is public. Figure 13-20 shows the UML diagram modified to include this notation.

**Figure 13-20**




## Data Type and Parameter Notation in UML Diagrams

The Unified Modeling Language also provides notation that you may use to indicate the data types of member variables, member functions, and parameters. To indicate the data type of a member variable, place a colon followed by the name of the data type after the name of the variable. For example, the `width` variable in the `Rectangle` class is a `double`. It could be listed as follows in the UML diagram:

```
- width : double
```

**NOTE:** In UML notation the variable name is listed first, then the data type. This is the opposite of C++ syntax, which requires the data type to appear first.

The return type of a member function can be listed in the same manner: After the function's name, place a colon followed by the return type. The `Rectangle` class's `getLength` function returns a `double`, so it could be listed as follows in the UML diagram:

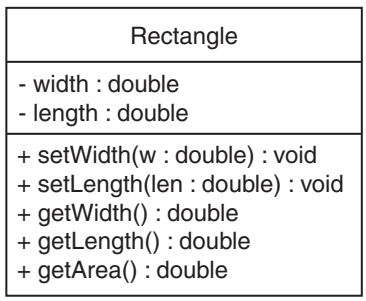
```
+ getLength() : double
```

Parameter variables and their data types may be listed inside a member function's parentheses. For example, the `Rectangle` class's `setLength` function has a `double` parameter named `len`, so it could be listed as follows in the UML diagram:

```
+ setLength(len : double) : void
```

Figure 13-21 shows a UML diagram for the `Rectangle` class with parameter and data type notation.

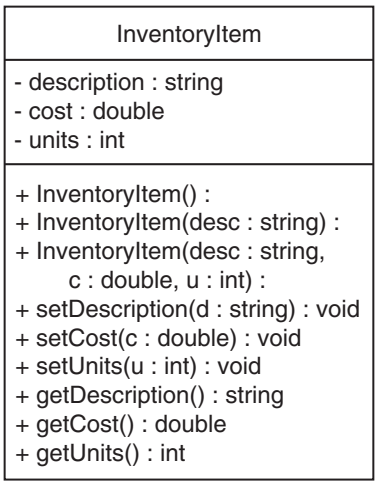
**Figure 13-21**



## Showing Constructors and Destructors in a UML Diagram

There is more than one accepted way of showing a class constructor in a UML diagram. In this book we will show a constructor just as any other function, except we will list no return type. For example, Figure 13-22 shows a UML diagram for the `InventoryItem` class that we looked at previously in this chapter.

**Figure 13-22**




# 13.17 Focus on Object-Oriented Design: Finding the Classes and Their Responsibilities

**CONCEPT:** **One of the first steps in creating an object-oriented application is determining the classes that are necessary and their responsibilities within the application.**

So far you have learned the basics of writing a class, creating an object from the class, and using the object to perform operations. This knowledge is necessary to create an object-oriented application, but it is not the first step in designing the application. The first step is to analyze the problem that you are trying to solve and determine the classes that you will need. In this section we will discuss a simple technique for finding the classes in a problem and determining their responsibilities.

## Finding the Classes

When developing an object-oriented application, one of your first tasks is to identify the classes that you will need to create. Typically, your goal is to identify the different types of real-world objects that are present in the problem and then create classes for those types of objects within your application.

Over the years, software professionals have developed numerous techniques for finding the classes in a given problem. One simple and popular technique involves the following steps.

1. Get a written description of the problem domain.
2. Identify all the nouns (including pronouns and noun phrases) in the description. Each of these is a potential class.
3. Refine the list to include only the classes that are relevant to the problem.

Let's take a closer look at each of these steps.

## Write a Description of the Problem Domain

The *problem domain* is the set of real-world objects, parties, and major events related to the problem. If you adequately understand the nature of the problem you are trying to solve, you can write a description of the problem domain yourself. If you do not thoroughly understand the nature of the problem, you should have an expert write the description for you.

For example, suppose we are programming an application that the manager of Joe's Automotive Shop will use to print service quotes for customers. Here is a description that an expert, perhaps Joe himself, might have written:

Joe's Automotive Shop services foreign cars and specializes in servicing cars made by Mercedes, Porsche, and BMW. When a customer brings a car to the shop, the manager gets the customer's name, address, and telephone number. The manager then determines the make, model, and year of the car and gives the customer a service quote. The service quote shows the estimated parts charges, estimated labor charges, sales tax, and total estimated charges.

The problem domain description should include any of the following:

- Physical objects such as vehicles, machines, or products
- Any role played by a person, such as manager, employee, customer, teacher, student, etc.
- The results of a business event, such as a customer order, or in this case a service quote
- Recordkeeping items, such as customer histories and payroll records

## Identify All of the Nouns

The next step is to identify all of the nouns and noun phrases. (If the description contains pronouns, include them too.) Here's another look at the previous problem domain description. This time the nouns and noun phrases appear in bold.

**Joe's Automotive Shop** services **foreign cars**, and specializes in servicing **cars** made by **Mercedes**, **Porsche**, and **BMW**. When a **customer** brings a **car** to the **shop**, the **manager** gets the **customer**'s **name**, **address**, and **telephone number**. The **manager** then determines the **make**, **model**, and **year** of the **car**, and gives the **customer** a **service quote**. The **service quote** shows the **estimated parts charges**, **estimated labor charges**, **sales tax**, and **total estimated charges**.

Notice that some of the nouns are repeated. The following list shows all of the nouns without duplicating any of them.

address
BMW
car
cars
customer
estimated labor charges
estimated parts charges
foreign cars
Joe's Automotive Shop
make
manager
Mercedes
model
name
Porsche
sales tax
service quote
shop
telephone number
total estimated charges
year

## Refine the List of Nouns

The nouns that appear in the problem description are merely candidates to become classes. It might not be necessary to make classes for them all. The next step is to refine the list to include only the classes that are necessary to solve the particular problem at hand. We will look at the common reasons that a noun can be eliminated from the list of potential classes.

1. **Some of the nouns really mean the same thing.**

In this example, the following sets of nouns refer to the same thing:

- **cars** and **foreign cars**
  These both refer to the general concept of a car.
- **Joe's Automotive Shop** and **shop**
  Both of these refer to the company "Joe's Automotive Shop."

We can settle on a single class for each of these. In this example we will arbitrarily eliminate **foreign cars** from the list, and use the word **cars**. Likewise we will eliminate **Joe's Automotive Shop** from the list and use the word **shop**. The updated list of potential classes is:

address
BMW
car
cars
customer
estimated labor charges
estimated parts charges
~~foreign cars~~
~~Joe's Automotive Shop~~
make
manager
Mercedes
model
name
Porsche
sales tax
service quote
shop
telephone number
total estimated charges
year

Because **cars** and **foreign cars** mean the same thing in this problem, we have eliminated **foreign cars**. Also, because **Joe's Automotive Shop** and **shop** mean the same thing, we have eliminated **Joe's Automotive Shop**.

2. **Some nouns might represent items that we do not need to be concerned with in order to solve the problem.**

A quick review of the problem description reminds us of what our application should do: print a service quote. In this example we can eliminate two unnecessary classes from the list:

- We can cross **shop** off the list because our application only needs to be concerned with individual service quotes. It doesn't need to work with or determine any company-wide information. If the problem description asked us to keep a total of all the service quotes, then it would make sense to have a class for the shop.
- We will not need a class for the **manager** because the problem statement does not direct us to process any information about the manager. If there were multiple shop managers, and the problem description had asked us to record which manager generated each service quote, then it would make sense to have a class for the manager.

The updated list of potential classes at this point is:

address
BMW
car
cars
customer
estimated labor charges
estimated parts charges
~~foreign cars~~
~~Joe's Automotive Shop~~
make
~~manager~~
Mercedes
model
name
Porsche
sales tax
service quote
~~shop~~
telephone number
total estimated charges
year

Our problem description does not direct us to process any information about the **shop**, or any information about the **manager**, so we have eliminated those from the list.

**3. Some of the nouns might represent objects, not classes.**

We can eliminate **Mercedes**, **Porsche**, and **BMW** as classes because, in this example, they all represent specific cars and can be considered instances of a **cars** class. Also, we can eliminate the word **car** from the list. In the description it refers to a specific car brought to the shop by a customer. Therefore, it would also represent an instance of a **cars** class. At this point the updated list of potential classes is:

address
~~BMW~~
~~car~~
cars
customer
estimated labor charges
estimated parts charges
~~foreign cars~~
~~Joe's Automotive Shop~~
~~manager~~
make
~~Mercedes~~
model
name
~~Porsche~~
sales tax
service quote
~~shop~~
telephone number
total estimated charges
year

We have eliminated **Mercedes**, **Porsche**, **BMW**, and **car** because they are all instances of a **cars** class. That means that these nouns identify objects, not classes.

**NOTE:** Some object-oriented designers take note of whether a noun is plural or singular. Sometimes a plural noun will indicate a class and a singular noun will indicate an object.

4. **Some of the nouns might represent simple values that can be stored in a variable and do not require a class.**

Remember, a class contains attributes and member functions. Attributes are related items that are stored within an object of the class, and define the object's state. Member functions are actions or behaviors that may be performed by an object of the class. If a noun represents a type of item that would not have any identifiable attributes or member functions, then it can probably be eliminated from the list. To help determine whether a noun represents an item that would have attributes and member functions, ask the following questions about it:

- Would you use a group of related values to represent the item's state?
- Are there any obvious actions to be performed by the item?

If the answers to both of these questions are no, then the noun probably represents a value that can be stored in a simple variable. If we apply this test to each of the nouns that remain in our list, we can conclude that the following are probably not classes: **address, estimated labor charges, estimated parts charges, make, model, name, sales tax, telephone number, total estimated charges,** and **year**. These are all simple string or numeric values that can be stored in variables. Here is the updated list of potential classes:

~~address~~
~~BMW~~
~~car~~
cars
customer
~~estimated labor charges~~
~~estimated parts charges~~
~~foreign cars~~
~~Joe's Automotive Shop~~
~~make~~
~~manager~~
~~Mercedes~~
~~model~~
~~name~~
~~Porsche~~
~~sales tax~~
~~service quote~~
~~shop~~
~~telephone number~~
~~total estimated charges~~
~~year~~
service quote

We have eliminated **address, estimated labor charges, estimated parts charges, make, model, name, sales tax, telephone number, total estimated charges,** and **year** as classes because they represent simple values that can be stored in variables.

As you can see from the list, we have eliminated everything except **cars**, **customer**, and **service quote**. This means that in our application, we will need classes to represent cars, customers, and service quotes. Ultimately, we will write a `Car` class, a `Customer` class, and a `ServiceQuote` class.

## Identifying a Class's Responsibilities

Once the classes have been identified, the next task is to identify each class's responsibilities. A class's *responsibilities* are

- the things that the class is responsible for knowing
- the actions that the class is responsible for doing

When you have identified the things that a class is responsible for knowing, then you have identified the class's attributes. Likewise, when you have identified the actions that a class is responsible for doing, you have identified its member functions.

It is often helpful to ask the questions "In the context of this problem, what must the class know? What must the class do?" The first place to look for the answers is in the description of the problem domain. Many of the things that a class must know and do will be mentioned. Some class responsibilities, however, might not be directly mentioned in the problem domain, so brainstorming is often required. Let's apply this methodology to the classes we previously identified from our problem domain.

### The `Customer` class

In the context of our problem domain, what must the `Customer` class know? The description directly mentions the following items, which are all attributes of a customer:

- the customer's name
- the customer's address
- the customer's telephone number

These are all values that can be represented as strings and stored in the class's member variables. The `Customer` class can potentially know many other things. One mistake that can be made at this point is to identify too many things that an object is responsible for knowing. In some applications, a `Customer` class might know the customer's email address. This particular problem domain does not mention that the customer's email address is used for any purpose, so we should not include it as a responsibility.

Now let's identify the class's member functions. In the context of our problem domain, what must the `Customer` class do? The only obvious actions are to

- create an object of the `Customer` class
- set and get the customer's name
- set and get the customer's address
- set and get the customer's telephone number

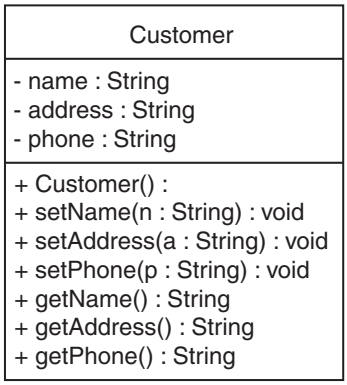
From this list we can see that the `Customer` class will have a constructor, as well as accessor and mutator functions for each of its attributes. Figure 13-23 shows a UML diagram for the `Customer` class.

### The `Car` Class

In the context of our problem domain, what must an object of the `Car` class know? The following items are all attributes of a car and are mentioned in the problem domain:

- the car's make
- the car's model
- the car's year

**Figure 13-23**

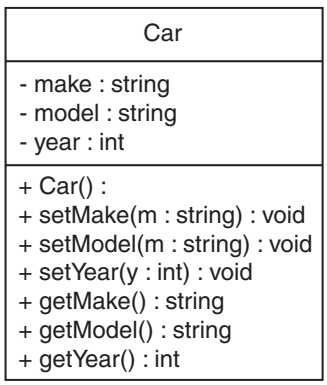



Now let's identify the class's member functions. In the context of our problem domain, what must the `Car` class do? Once again, the only obvious actions are the standard set of member functions that we will find in most classes (constructors, accessors, and mutators). Specifically, the actions are:

- create an object of the `Car` class
- set and get the car's make
- set and get the car's model
- set and get the car's year

Figure 13-24 shows a UML diagram for the `Car` class at this point.

**Figure 13-24**



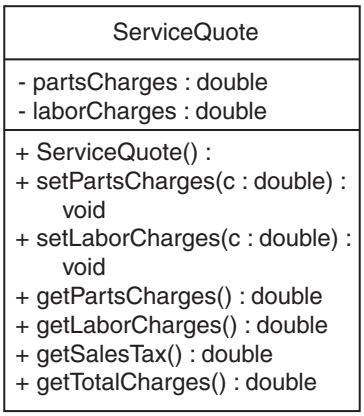

### The `ServiceQuote` Class

In the context of our problem domain, what must an object of the `ServiceQuote` class know? The problem domain mentions the following items:

- the estimated parts charges
- the estimated labor charges
- the sales tax
- the total estimated charges

Careful thought and a little brainstorming will reveal that two of these items are the results of calculations: sales tax and total estimated charges. These items are dependent on the values of the estimated parts and labor charges. In order to avoid the risk of holding stale data, we will not store these values in member variables. Rather, we will provide member functions that calculate these values and return them. The other member functions that we will need for this class are a constructor and the accessors and mutators for the estimated parts charges and estimated labor charges attributes. Figure 13-25 shows a UML diagram for the `ServiceQuote` class.

**Figure 13-25**




## This Is Only the Beginning

You should look at the process that we have discussed in this section as merely a starting point. It's important to realize that designing an object-oriented application is an iterative process. It may take you several attempts to identify all of the classes that you will need and determine all of their responsibilities. As the design process unfolds, you will gain a deeper understanding of the problem, and consequently you will see ways to improve the design.

## Checkpoint

13.27 What is a problem domain?

13.28 When designing an object-oriented application, who should write a description of the problem domain?

13.29 How do you identify the potential classes in a problem domain description?

13.30 What are a class's responsibilities?

13.31 What two questions should you ask to determine a class's responsibilities?

13.32 Will all of a class's actions always be directly mentioned in the problem domain description?

13.33 Look at the following description of a problem domain:

A doctor sees patients in her practice. When a patient comes to the practice, the doctor performs one or more procedures on the patient. Each procedure that the doctor performs has a description and a standard fee. As the patient leaves the practice, he or she receives a statement from the office manager. The statement

shows the patient's name and address, as well as the procedures that were performed, and the total charge for the procedures.

Assume that you are writing an application to generate a statement that can be printed and given to the patient.

A) Identify all of the potential classes in this problem domain.

B) Refine the list to include only the necessary class or classes for this problem.

C) Identify the responsibilities of the class or classes that you identified in step B.

## Review Questions and Exercises

### Short Answer

1. What is the difference between a class and an instance of the class?
2. What is the difference between the following `Person` structure and `Person` class?

```
struct Person
{
   string name;
   int age;
};

class Person
{
   string name;
   int age;
};
```

3. What is the default access specification of class members?
4. Look at the following function header for a member function.

```
void Circle::getRadius()
```

What is the name of the function?

What class is the function a member of?

5. A contractor uses a blueprint to build a set of identical houses. Are classes analogous to the blueprint or the houses?
6. What is a mutator function? What is an accessor function?
7. Is it a good idea to make member variables private? Why or why not?
8. Can you think of a good reason to avoid writing statements in a class member function that use `cout` or `cin`?
9. Under what circumstances should a member function be private?
10. What is a constructor? What is a destructor?
11. What is a default constructor? Is it possible to have more than one default constructor?
12. Is it possible to have more than one constructor? Is it possible to have more than one destructor?
13. If a class object is dynamically allocated in memory, does its constructor execute? If so, when?

14. When defining an array of class objects, how do you pass arguments to the constructor for each object in the array?
15. What are a class's responsibilities?
16. How do you identify the classes in a problem domain description?

### Fill-in-the-Blank

17. The two common programming methods in practice today are _____ and _____.
18. _____ programming is centered around functions or procedures.
19. _____ programming is centered around objects.
20. _____ is an object's ability to contain and manipulate its own data.
21. In C++ the _____ is the construct primarily used to create objects.
22. A class is very similar to a(n) _____.
23. A(n) _____ is a key word inside a class declaration that establishes a member's accessibility.
24. The default access specification of class members is _____.
25. The default access specification of a struct in C++ is _____.
26. Defining a class object is often called the _____ of a class.
27. Members of a class object may be accessed through a pointer to the object by using the _____ operator.
28. If you were writing the declaration of a class named `Canine`, what would you name the file it was stored in? _____
29. If you were writing the external definitions of the `Canine` class's member functions, you would save them in a file named _____.
30. When a member function's body is written inside a class declaration, the function is _____.
31. A(n) _____ is automatically called when an object is created.
32. A(n) _____ is a member function with the same name as the class.
33. _____ are useful for performing initialization or setup routines in a class object.
34. Constructors cannot have a(n) _____ type.
35. A(n) _____ constructor is one that requires no arguments.
36. A(n) _____ is a member function that is automatically called when an object is destroyed.
37. A destructor has the same name as the class, but is preceded by a(n) _____ character.
38. Like constructors, destructors cannot have a(n) _____ type.
39. A constructor whose arguments all have default values is a(n) _____ constructor.
40. A class may have more than one constructor, as long as each has a different _____.
41. A class may only have one default _____ and one _____.
42. A(n) _____ may be used to pass arguments to the constructors of elements in an object array.

## Algorithm Workbench

43. Write a class declaration named `Circle` with a private member variable named `radius`. Write set and get functions to access the `radius` variable, and a function named `getArea` that returns the area of the circle. The area is calculated as

    ```
    3.14159 * radius * radius
    ```

44. Add a default constructor to the `Circle` class in question 43. The constructor should initialize the `radius` member to 0.
45. Add an overloaded constructor to the `Circle` class in question 44. The constructor should accept an argument and assign its value to the `radius` member variable.
46. Write a statement that defines an array of five objects of the `Circle` class in question 45. Let the default constructor execute for each element of the array.
47. Write a statement that defines an array of five objects of the `Circle` class in question 45. Pass the following arguments to the elements' constructor: 12, 7, 9, 14, and 8.
48. Write a `for` loop that displays the radius and area of the circles represented by the array you defined in question 47.
49. If the items on the following list appeared in a problem domain description, which would be potential classes?

| | | |
|---|---|---|
| Animal | Medication | Nurse |
| Inoculate | Operate | Advertise |
| Doctor | Invoice | Measure |
| Patient | Client | Customer |

50. Look at the following description of a problem domain:

    The bank offers the following types of accounts to its customers: savings accounts, checking accounts, and money market accounts. Customers are allowed to deposit money into an account (thereby increasing its balance), withdraw money from an account (thereby decreasing its balance), and earn interest on the account. Each account has an interest rate.

    Assume that you are writing an application that will calculate the amount of interest earned for a bank account.

    A) Identify the potential classes in this problem domain.
    B) Refine the list to include only the necessary class or classes for this problem.
    C) Identify the responsibilities of the class or classes.

## True or False

51. T F Private members must be declared before public members.
52. T F Class members are private by default.
53. T F Members of a `struct` are private by default.
54. T F Classes and structures in C++ are very similar.
55. T F All private members of a class must be declared together.
56. T F All public members of a class must be declared together.
57. T F It is legal to define a pointer to a class object.
58. T F You can use the `new` operator to dynamically allocate an instance of a class.

59. T F A private member function may be called from a statement outside the class, as long as the statement is in the same program as the class declaration.
60. T F Constructors do not have to have the same name as the class.
61. T F Constructors may not have a return type.
62. T F Constructors cannot take arguments.
63. T F Destructors cannot take arguments.
64. T F Destructors may return a value.
65. T F Constructors may have default arguments.
66. T F Member functions may be overloaded.
67. T F Constructors may not be overloaded.
68. T F A class may not have a constructor with no parameter list, and a constructor whose arguments all have default values.
69. T F A class may only have one destructor.
70. T F When an array of objects is defined, the constructor is only called for the first element.
71. T F To find the classes needed for an object-oriented application, you identify all of the verbs in a description of the problem domain.
72. T F A class's responsibilities are the things the class is responsible for knowing, and actions the class must perform.

### Find the Errors

Each of the following class declarations or programs contain errors. Find as many as possible.

73.
```
class Circle:
{
private
   double centerX;
   double centerY;
   double radius;
public
   setCenter(double, double);
   setRadius(double);
}
```

74.
```
#include <iostream>
using namespace std;

Class Moon;
{
Private;
   double earthWeight;
   double moonWeight;
Public;
   moonWeight(double ew);
      { earthWeight = ew; moonWeight = earthWeight / 6; }
   double getMoonWeight();
      { return moonWeight; }
}
```

```
int main()
{
   double earth;
   cout >> "What is your weight? ";
   cin << earth;
   Moon lunar(earth);
   cout << "On the moon you would weigh "
        <<lunar.getMoonWeight() << endl;
   return 0;
}
```

75.
```
#include <iostream>
using namespace std;

class DumbBell;
{
   int weight;
public:
   void setWeight(int);
};
void setWeight(int w)
{
   weight = w;
}
int main()
{
   DumbBell bar;

   DumbBell(200);
   cout << "The weight is " << bar.weight << endl;
   return 0;
}
```

76.
```
class Change
{
public:
   int pennies;
   int nickels;
   int dimes;
   int quarters;
   Change()
      { pennies = nickels = dimes = quarters = 0; }
   Change(int p = 100, int n = 50, d = 50, q = 25);
};

void Change::Change(int p, int n, d, q)
{
   pennies = p;
   nickels = n;
   dimes = d;
   quarters = q;
}
```

## Programming Challenges

1. **Date**

   Design a class called `Date`. The class should store a date in three integers: `month`, `day`, and `year`. There should be member functions to print the date in the following forms:

   12/25/2014
   December 25, 2014
   25 December 2014

   Demonstrate the class by writing a complete program implementing it.

   *Input Validation: Do not accept values for the day greater than 31 or less than 1. Do not accept values for the month greater than 12 or less than 1.*

2. **`Employee` Class**

VideoNote
**Solving the `Employee` Class Problem**

   Write a class named `Employee` that has the following member variables:

   - **`name`.** A string that holds the employee's name.
   - **`idNumber`.** An `int` variable that holds the employee's ID number.
   - **`department`.** A string that holds the name of the department where the employee works.
   - **`position`.** A string that holds the employee's job title.

   The class should have the following constructors:

   - A constructor that accepts the following values as arguments and assigns them to the appropriate member variables: employee's name, employee's ID number, department, and position.
   - A constructor that accepts the following values as arguments and assigns them to the appropriate member variables: employee's name and ID number. The `department` and `position` fields should be assigned an empty string (`""`).
   - A default constructor that assigns empty strings (`""`) to the `name`, `department`, and `position` member variables, and 0 to the `idNumber` member variable.

   Write appropriate mutator functions that store values in these member variables and accessor functions that return the values in these member variables. Once you have written the class, write a separate program that creates three `Employee` objects to hold the following data.

| Name | ID Number | Department | Position |
|---|---|---|---|
| Susan Meyers | 47899 | Accounting | Vice President |
| Mark Jones | 39119 | IT | Programmer |
| Joy Rogers | 81774 | Manufacturing | Engineer |

   The program should store this data in the three objects and then display the data for each employee on the screen.

3. **`Car` Class**

   Write a class named `Car` that has the following member variables:

   - **`yearModel`.** An `int` that holds the car's year model.
   - **`make`.** A string that holds the make of the car.
   - **`speed`.** An `int` that holds the car's current speed.

In addition, the class should have the following constructor and other member functions.

- **Constructor.** The constructor should accept the car's year model and make as arguments. These values should be assigned to the object's `yearModel` and `make` member variables. The constructor should also assign 0 to the `speed` member variables.
- **Accessor.** Appropriate accessor functions to get the values stored in an object's `yearModel`, `make`, and `speed` member variables.
- **`accelerate`.** The `accelerate` function should add 5 to the `speed` member variable each time it is called.
- **`brake`.** The `brake` function should subtract 5 from the `speed` member variable each time it is called.

Demonstrate the class in a program that creates a `Car` object, and then calls the `accelerate` function five times. After each call to the `accelerate` function, get the current speed of the car and display it. Then, call the `brake` function five times. After each call to the `brake` function, get the current speed of the car and display it.

**4. Personal Information Class**

Design a class that holds the following personal data: name, address, age, and phone number. Write appropriate accessor and mutator functions. Demonstrate the class by writing a program that creates three instances of it. One instance should hold your information, and the other two should hold your friends' or family members' information.

**5. `RetailItem` Class**

Write a class named `RetailItem` that holds data about an item in a retail store. The class should have the following member variables:

- **`description`.** A string that holds a brief description of the item.
- **`unitsOnHand`.** An `int` that holds the number of units currently in inventory.
- **`price`.** A `double` that holds the item's retail price.

Write a constructor that accepts arguments for each member variable, appropriate mutator functions that store values in these member variables, and accessor functions that return the values in these member variables. Once you have written the class, write a separate program that creates three `RetailItem` objects and stores the following data in them.

| | Description | Units On Hand | Price |
|---|---|---|---|
| Item #1 | Jacket | 12 | 59.95 |
| Item #2 | Designer Jeans | 40 | 34.95 |
| Item #3 | Shirt | 20 | 24.95 |

6. **Inventory Class**

Design an `Inventory` class that can hold information and calculate data for items in a retail store's inventory. The class should have the following *private* member variables:

| Variable Name | Description |
|---|---|
| `itemNumber` | An `int` that holds the item's item number. |
| `quantity` | An `int` for holding the quantity of the items on hand. |
| `cost` | A `double` for holding the wholesale per-unit cost of the item |
| `totalCost` | A `double` for holding the total inventory cost of the item (calculated as `quantity` times `cost`). |

The class should have the following *public* member functions:

| Member Function | Description |
|---|---|
| Default Constructor | Sets all the member variables to 0. |
| Constructor #2 | Accepts an item's number, cost, and quantity as arguments. The function should copy these values to the appropriate member variables and then call the `setTotalCost` function. |
| `setItemNumber` | Accepts an integer argument that is copied to the `itemNumber` member variable. |
| `setQuantity` | Accepts an integer argument that is copied to the `quantity` member variable. |
| `setCost` | Accepts a `double` argument that is copied to the `cost` member variable. |
| `setTotalCost` | Calculates the total inventory cost for the item (`quantity` times `cost`) and stores the result in `totalCost`. |
| `getItemNumber` | Returns the value in `itemNumber`. |
| `getQuantity` | Returns the value in `quantity`. |
| `getCost` | Returns the value in `cost`. |
| `getTotalCost` | Returns the value in `totalCost`. |

Demonstrate the class in a driver program.

*Input Validation: Do not accept negative values for item number, quantity, or cost.*

7. **`TestScores` Class**

Design a `TestScores` class that has member variables to hold three test scores. The class should have a constructor, accessor, and mutator functions for the test score fields and a member function that returns the average of the test scores. Demonstrate the

class by writing a separate program that creates an instance of the class. The program should ask the user to enter three test scores, which are stored in the `TestScores` object. Then the program should display the average of the scores, as reported by the `TestScores` object.

8. **Circle Class**

   Write a `Circle` class that has the following member variables:

   - `radius`: a `double`
   - `pi`: a `double` initialized with the value 3.14159

   The class should have the following member functions:

   - **Default Constructor.** A default constructor that sets `radius` to 0.0.
   - **Constructor.** Accepts the radius of the circle as an argument.
   - **`setRadius`.** A mutator function for the radius variable.
   - **`getRadius`.** An accessor function for the radius variable.
   - **`getArea`.** Returns the area of the circle, which is calculated as
     `area = pi * radius * radius`
   - **`getDiameter`.** Returns the diameter of the circle, which is calculated as
     `diameter = radius * 2`
   - **`getCircumference`.** Returns the circumference of the circle, which is calculated as
     `circumference = 2 * pi * radius`

   Write a program that demonstrates the `Circle` class by asking the user for the circle's radius, creating a `Circle` object, and then reporting the circle's area, diameter, and circumference.

9. **Population**

   In a population, the birth rate and death rate are calculated as follows:

   Birth Rate = Number of Births ÷ Population

   Death Rate = Number of Deaths ÷ Population

   For example, in a population of 100,000 that has 8,000 births and 6,000 deaths per year, the birth rate and death rate are:

   Birth Rate = 8,000 ÷ 100,000 = 0.08

   Death Rate = 6,000 ÷ 100,000 = 0.06

   Design a `Population` class that stores a population, number of births, and number of deaths for a period of time. Member functions should return the birth rate and death rate. Implement the class in a program.

   *Input Validation: Do not accept population figures less than 1, or birth or death numbers less than 0.*

10. **Number Array Class**

    Design a class that has an array of floating-point numbers. The constructor should accept an integer argument and dynamically allocate the array to hold that many numbers. The destructor should free the memory held by the array. In addition, there should be member functions to perform the following operations:

    - Store a number in any element of the array
    - Retrieve a number from any element of the array

- Return the highest value stored in the array
- Return the lowest value stored in the array
- Return the average of all the numbers stored in the array

Demonstrate the class in a program.

**11. Payroll**

Design a `PayRoll` class that has data members for an employee's hourly pay rate, number of hours worked, and total pay for the week. Write a program with an array of seven `PayRoll` objects. The program should ask the user for the number of hours each employee has worked and will then display the amount of gross pay each has earned.

*Input Validation: Do not accept values greater than 60 for the number of hours worked.*

**12. Coin Toss Simulator**

Write a class named `Coin`. The `Coin` class should have the following member variable:

- A `string` named `sideUp`. The `sideUp` member variable will hold either "heads" or "tails" indicating the side of the coin that is facing up.

The `Coin` class should have the following member functions:

- A default constructor that randomly determines the side of the coin that is facing up ("heads" or "tails") and initializes the `sideUp` member variable accordingly.
- A `void` member function named `toss` that simulates the tossing of the coin. When the `toss` member function is called, it randomly determines the side of the coin that is facing up ("heads" or "tails") and sets the `sideUp` member variable accordingly.
- A member function named `getSideUp` that returns the value of the `sideUp` member variable.

Write a program that demonstrates the `Coin` class. The program should create an instance of the class and display the side that is initially facing up. Then, use a loop to toss the coin 20 times. Each time the coin is tossed, display the side that is facing up. The program should keep count of the number of times heads is facing up and the number of times tails is facing up, and display those values after the loop finishes.

**13. Tossing Coins for a Dollar**

For this assignment, you will create a game program using the `Coin` class from Programming Challenge 12. The program should have three instances of the `Coin` class: one representing a quarter, one representing a dime, and one representing a nickel.

When the game begins, your starting balance is $0. During each round of the game, the program will toss the simulated coins. When a coin is tossed, the value of the coin is added to your balance if it lands heads-up. For example, if the quarter lands heads-up, 25 cents is added to your balance. Nothing is added to your balance for coins that land tails-up. The game is over when your balance reaches $1 or more. If your balance is exactly $1, you win the game. You lose if your balance exceeds $1.

**14. Fishing Game Simulation**

For this assignment, you will write a program that simulates a fishing game. In this game, a six-sided die is rolled to determine what the user has caught. Each possible item is worth a certain number of fishing points. The points will not be displayed until

the user has finished fishing, and then a message is displayed congratulating the user depending on the number of fishing points gained.

Here are some suggestions for the game's design:

- Each round of the game is performed as an iteration of a loop that repeats as long as the player wants to fish for more items.
- At the beginning of each round, the program will ask the user whether he or she wants to continue fishing.
- The program simulates the rolling of a six-sided die (use the `Die` class that was demonstrated in this chapter).
- Each item that can be caught is represented by a number generated from the die. For example, 1 for "a huge fish," 2 for "an old shoe," 3 for "a little fish," and so on.
- Each item the user catches is worth a different amount of points.
- The loop keeps a running total of the user's fishing points.
- After the loop has finished, the total number of fishing points is displayed, along with a message that varies depending on the number of points earned.

**15. Mortgage Payment**

Design a class that will determine the monthly payment on a home mortgage. The monthly payment with interest compounded monthly can be calculated as follows:

$$\text{Payment} = \frac{\text{Loan} \times \frac{\text{Rate}}{12} \times \text{Term}}{\text{Term} - 1}$$

where

$$\text{Term} = \left(1 + \frac{\text{Rate}}{12}\right)^{12 \times \text{Years}}$$

Payment = the monthly payment
Loan = the dollar amount of the loan
Rate = the annual interest rate
Years = the number of years of the loan

The class should have member functions for setting the loan amount, interest rate, and number of years of the loan. It should also have member functions for returning the monthly payment amount and the total amount paid to the bank at the end of the loan period. Implement the class in a complete program.

*Input Validation: Do not accept negative numbers for any of the loan values.*

**16. Freezing and Boiling Points**

The following table lists the freezing and boiling points of several substances.

| Substance | Freezing Point | Boiling Point |
|---|---|---|
| Ethyl Alcohol | –173 | 172 |
| Oxygen | –362 | –306 |
| Water | 32 | 212 |

Design a class that stores a temperature in a `temperature` member variable and has the appropriate accessor and mutator functions. In addition to appropriate constructors, the class should have the following member functions:

- **`isEthylFreezing`.** This function should return the `bool` value `true` if the temperature stored in the `temperature` field is at or below the freezing point of ethyl alcohol. Otherwise, the function should return `false`.
- **`isEthylBoiling`.** This function should return the `bool` value `true` if the temperature stored in the `temperature` field is at or above the boiling point of ethyl alcohol. Otherwise, the function should return `false`.
- **`isOxygenFreezing`.** This function should return the `bool` value `true` if the temperature stored in the `temperature` field is at or below the freezing point of oxygen. Otherwise, the function should return `false`.
- **`isOxygenBoiling`.** This function should return the `bool` value `true` if the temperature stored in the `temperature` field is at or above the boiling point of oxygen. Otherwise, the function should return `false`.
- **`isWaterFreezing`.** This function should return the `bool` value `true` if the temperature stored in the `temperature` field is at or below the freezing point of water. Otherwise, the function should return `false`.
- **`isWaterBoiling`.** This function should return the `bool` value `true` if the temperature stored in the `temperature` field is at or above the boiling point of water. Otherwise, the function should return `false`.

Write a program that demonstrates the class. The program should ask the user to enter a temperature and then display a list of the substances that will freeze at that temperature and those that will boil at that temperature. For example, if the temperature is –20 the class should report that water will freeze and oxygen will boil at that temperature.

**17. Cash Register**

Design a `CashRegister` class that can be used with the `InventoryItem` class discussed in this chapter. The `CashRegister` class should perform the following:

1. Ask the user for the item and quantity being purchased.
2. Get the item's cost from the `InventoryItem` object.
3. Add a 30% profit to the cost to get the item's unit price.
4. Multiply the unit price times the quantity being purchased to get the purchase subtotal.
5. Compute a 6% sales tax on the subtotal to get the purchase total.
6. Display the purchase subtotal, tax, and total on the screen.
7. Subtract the quantity being purchased from the `onHand` variable of the `InventoryItem` class object.

Implement both classes in a complete program. Feel free to modify the `InventoryItem` class in any way necessary.

*Input Validation: Do not accept a negative value for the quantity of items being purchased.*

**18. A Game of 21**

For this assignment, you will write a program that lets the user play against the computer in a variation of the popular blackjack card game. In this variation of the game,

two six-sided dice are used instead of cards. The dice are rolled, and the player tries to beat the computer's hidden total without going over 21.

Here are some suggestions for the game's design:

- Each round of the game is performed as an iteration of a loop that repeats as long as the player agrees to roll the dice, and the player's total does not exceed 21.
- At the beginning of each round, the program will ask the users whether they want to roll the dice to accumulate points.
- During each round, the program simulates the rolling of two six-sided dice. It rolls the dice first for the computer, and then it asks the user if he or she wants to roll. (Use the `Die` class that was demonstrated in this chapter to simulate the dice).
- The loop keeps a running total of both the computer and the user's points.
- The computer's total should remain hidden until the loop has finished.
- After the loop has finished, the computer's total is revealed, and the player with the most points without going over 21 wins.

**19. Trivia Game**

In this programming challenge you will create a simple trivia game for two players. The program will work like this:

- Starting with player 1, each player gets a turn at answering five trivia questions. (There are a total of 10 questions.) When a question is displayed, four possible answers are also displayed. Only one of the answers is correct, and if the player selects the correct answer he or she earns a point.
- After answers have been selected for all of the questions, the program displays the number of points earned by each player and declares the player with the highest number of points the winner.

In this program you will design a `Question` class to hold the data for a trivia question. The `Question` class should have member variables for the following data:

- A trivia question
- Possible answer #1
- Possible answer #2
- Possible answer #3
- Possible answer #4
- The number of the correct answer (1, 2, 3, or 4)

The `Question` class should have appropriate constructor(s), accessor, and mutator functions.

The program should create an array of 10 `Question` objects, one for each trivia question. Make up your own trivia questions on the subject or subjects of your choice for the objects.

## Group Project

**20. Patient Fees**

1. This program should be designed and written by a team of students. Here are some suggestions:
   - One or more students may work on a single class.
   - The requirements of the program should be analyzed so each student is given about the same workload.

- The parameters and return types of each function and class member function should be decided in advance.
- The program will be best implemented as a multi-file program.

2. You are to write a program that computes a patient's bill for a hospital stay. The different components of the program are

   The `PatientAccount` class
   The `Surgery` class
   The `Pharmacy` class
   The `main` program

   - The `PatientAccount` class will keep a total of the patient's charges. It will also keep track of the number of days spent in the hospital. The group must decide on the hospital's daily rate.
   - The `Surgery` class will have stored within it the charges for at least five types of surgery. It can update the charges variable of the `PatientAccount` class.
   - The `Pharmacy` class will have stored within it the price of at least five types of medication. It can update the charges variable of the `PatientAccount` class.
   - The student who designs the main program will design a menu that allows the user to enter a type of surgery and a type of medication, and check the patient out of the hospital. When the patient checks out, the total charges should be displayed.

CHAPTER 14

# More About Classes

**TOPICS**



## 14.1 Instance and Static Members

**CONCEPT:** **Each instance of a class has its own copies of the class's instance variables. If a member variable is declared `static`, however, all instances of that class have access to that variable. If a member function is declared `static`, it may be called without any instances of the class being defined.**

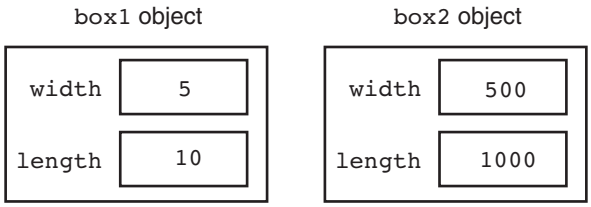
### Instance Variables

Each class object (an instance of a class) has its own copy of the class's member variables. An object's member variables are separate and distinct from the member variables of other objects of the same class. For example, recall that the `Rectangle` class discussed in Chapter 13 has two member variables: `width` and `length`. Suppose that we define two objects of the `Rectangle` class and set their `width` and `length` member variables as shown in the following code.

```
Rectangle box1, box2;

// Set the width and length for box1.
box1.setWidth(5);
box1.setLength(10);

// Set the width and length for box2.
box2.setWidth(500);
box2.setLength(1000);
```

This code creates `box1` and `box2`, which are two distinct objects. Each has its own `width` and `length` member variables, as illustrated in Figure 14-1.

**Figure 14-1**




When the `getWidth` member function is called, it returns the value stored in the calling object's `width` member variable. For example, the following statement displays `5 500`.

```
cout << box1.getWidth() << " " << box2.getWidth() << endl;
```

In object-oriented programming, member variables such as the `Rectangle` class's `width` and `length` members are known as *instance variables*. They are called instance variables because each instance of the class has its own copies of the variables.

## Static Members

It is possible to create a member variable or member function that does not belong to any instance of a class. Such members are known as a *static member variables* and *static member functions*. When a value is stored in a static member variable, it is not stored in an instance of the class. In fact, an instance of the class doesn't even have to exist in order for values to be stored in the class's static member variables. Likewise, static member functions do not operate on instance variables. Instead, they can operate only on static member variables. You can think of static member variables and static member functions as belonging to the class instead of to an instance of the class. In this section, we will take a closer look at static members. First we will examine static member variables.

## Static Member Variables

When a member variable is declared with the key word `static`, there will be only one copy of the member variable in memory, regardless of the number of instances of the class that might exist. A single copy of a class's static member variable is shared by all instances of the class. For example, the following `Tree` class uses a static member variable to keep count of the number of instances of the class that are created.

### Contents of `Tree.h`

```
1  // Tree class
2  class Tree
3  {
4  private:
5       static int objectCount;     // Static member variable.
6  public:
7       // Constructor
8       Tree()
9          { objectCount++; }
10
```

```
11      // Accessor function for objectCount
12      int getObjectCount() const
13         { return objectCount; }
14  };
15
16  // Definition of the static member variable, written
17  // outside the class.
18  int Tree::objectCount = 0;
```

First, notice in line 5 the declaration of the static member variable named `objectCount`: A static member variable is created by placing the key word `static` before the variable's data type. Also notice that in line 18 we have written a definition statement for the `objectCount` variable and that the statement is outside the class declaration. This external definition statement causes the variable to be created in memory and is required. In line 18 we have explicitly initialized the `objectCount` variable with the value 0. We could have left out the initialization because C++ automatically stores 0 in all uninitialized static member variables. It is a good practice to initialize the variable anyway, so it is clear to anyone reading the code that the variable starts out with the value 0.

Next, look at the constructor in lines 8 and 9. In line 9 the ++ operator is used to increment `objectCount`. Each time an instance of the `Tree` class is created, the constructor will be called, and the `objectCount` member variable will be incremented. As a result, the `objectCount` member variable will contain the number of instances of the `Tree` class that have been created. The `getObjectCount` function, in lines 12 and 13, returns the value in `objectCount`. Program 14-1 demonstrates this class.

**Program 14-1**

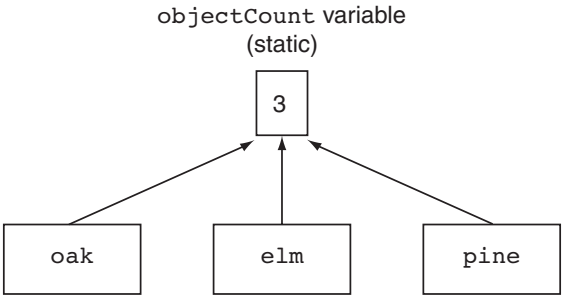
```
1  // This program demonstrates a static member variable.
2  #include <iostream>
3  #include "Tree.h"
4  using namespace std;
5
6  int main()
7  {
8       // Define three Tree objects.
9       Tree oak;
10      Tree elm;
11      Tree pine;
12
13      // Display the number of Tree objects we have.
14      cout << "We have " << pine.getObjectCount()
15           << " trees in our program!\n";
16      return 0;
17 }
```

**Program Output**

```
We have 3 trees in our program!
```

The program creates three instances of the `Tree` class, stored in the variables `oak`, `elm`, and `pine`. Although there are three instances of the class, there is only one copy of the static `objectCount` variable. This is illustrated in Figure 14-2.

**Figure 14-2**




Instances of the `Tree` class

In line 14 the program calls the `getObjectCount` member function to retrieve the number of instances that have been created. Although the program uses the `pine` object to call the member function, the same value would be returned if any of the objects had been used. For example, all three of the following `cout` statements would display the same thing.

```
cout << "We have " << oak.getObjectCount() << " trees\n";
cout << "We have " << elm.getObjectCount() << " trees\n";
cout << "We have " << pine.getObjectCount() << " trees\n";
```

A more practical use of a static member variable is demonstrated in Program 14-2. The `Budget` class is used to gather the budget requests for all the divisions of a company. The class uses a static member, `corpBudget`, to hold the amount of the overall corporate budget. When the member function `addBudget` is called, its argument is added to the current contents of `corpBudget`. By the time the program is finished, `corpBudget` will contain the total of all the values placed there by all the `Budget` class objects. (These files are stored in the Student Source Code Folder `Chapter 14\Budget Version 1`.)

### Contents of `Budget.h` (Version 1)

```
 1  #ifndef BUDGET_H
 2  #define BUDGET_H
 3
 4  // Budget class declaration
 5  class Budget
 6  {
 7  private:
 8       static double corpBudget;   // Static member
 9       double divisionBudget;      // Instance member
10  public:
11       Budget()
12          { divisionBudget = 0; }
13
14       void addBudget(double b)
15          { divisionBudget += b;
16            corpBudget += b; }
17
18       double getDivisionBudget() const
19          { return divisionBudget; }
20
```

```
21      double getCorpBudget() const
22        { return corpBudget; }
23  };
24
25  // Definition of static member variable corpBudget
26  double Budget::corpBudget = 0;
27
28  #endif
```

**Program 14-2**

```
1  // This program demonstrates a static class member variable.
2  #include <iostream>
3  #include <iomanip>
4  #include "Budget.h"
5  using namespace std;
6
7  int main()
8  {
9       int count;                          // Loop counter
10      const int NUM_DIVISIONS = 4;        // Number of divisions
11      Budget divisions[NUM_DIVISIONS];  // Array of Budget objects
12
13      // Get the budget requests for each division.
14      for (count = 0; count < NUM_DIVISIONS; count++)
15      {
16           double budgetAmount;
17           cout << "Enter the budget request for division ";
18           cout << (count + 1) << ": ";
19           cin >> budgetAmount;
20           divisions[count].addBudget(budgetAmount);
21      }
22
23      // Display the budget requests and the corporate budget.
24      cout << fixed << showpoint << setprecision(2);
25      cout << "\nHere are the division budget requests:\n";
26      for (count = 0; count < NUM_DIVISIONS; count++)
27      {
28           cout << "\tDivision " << (count + 1) << "\t$ ";
29           cout << divisions[count].getDivisionBudget() << endl;
30      }
31      cout << "\tTotal Budget Requests:\t$ ";
32      cout << divisions[0].getCorpBudget() << endl;
33
34      return 0;
35 }
```

**Program Output with Example Input Shown in Bold**

```
Enter the budget request for division 1: 100000 [Enter]
Enter the budget request for division 2: 200000 [Enter]
Enter the budget request for division 3: 300000 [Enter]
Enter the budget request for division 4: 400000 [Enter]  (program output continues)
```

**Program 14-2** *(continued)*

```
Here are the division budget requests:
     Division 1     $ 100000.00
     Division 2     $ 200000.00
     Division 3     $ 300000.00
     Division 4     $ 400000.00
     Total Budget Requests:   $ 1000000.00
```

## Static Member Functions

You declare a static member function by placing the `static` keyword in the function's prototype. Here is the general form:

```
static ReturnType FunctionName (ParameterTypeList);
```

A function that is a static member of a class cannot access any nonstatic member data in its class. With this limitation in mind, you might wonder what purpose static member functions serve. The following two points are important for understanding their usefulness:

- Even though static member variables are declared in a class, they are actually defined outside the class declaration. The lifetime of a class's static member variable is the lifetime of the program. This means that a class's static member variables come into existence before any instances of the class are created.
- A class's static member functions can be called before any instances of the class are created. This means that a class's static member functions can access the class's static member variables *before* any instances of the class are defined in memory. This gives you the ability to create very specialized setup routines for class objects.

Program 14-3, a modification of Program 14-2, demonstrates this feature. It asks the user to enter the main office's budget request before any division requests are entered. The `Budget` class has been modified to include a static member function named `mainOffice`. This function adds its argument to the static `corpBudget` variable and is called before any instances of the `Budget` class are defined. (These files are stored in the Student Source Code Folder `Chapter 14\Budget Version 2`.)

### Contents of `Budget.h` (Version 2)

```
 1  #ifndef BUDGET_H
 2  #define BUDGET_H
 3
 4  // Budget class declaration
 5  class Budget
 6  {
 7  private:
 8       static double corpBudget;   // Static member variable
 9       double divisionBudget;      // Instance member variable
10  public:
11       Budget()
12            { divisionBudget = 0; }
13
```

```
14      void addBudget(double b)
15           { divisionBudget += b;
16             corpBudget += b; }
17
18      double getDivisionBudget() const
19           { return divisionBudget; }
20
21      double getCorpBudget() const
22           { return corpBudget; }
23
24      static void mainOffice(double);  // Static member function
25  };
26
27  #endif
```

### Contents of `Budget.cpp`

```
1   #include "Budget.h"
2
3   // Definition of corpBudget static member variable
4   double Budget::corpBudget = 0;
5
6   //*********************************************************
7   // Definition of static member function mainOffice.       *
8   // This function adds the main office's budget request to *
9   // the corpBudget variable.                               *
10  //*********************************************************
11
12  void Budget::mainOffice(double moffice)
13  {
14      corpBudget += moffice;
15  }
```

### Program 14-3

```
1   // This program demonstrates a static member function.
2   #include <iostream>
3   #include <iomanip>
4   #include "Budget.h"
5   using namespace std;
6
7   int main()
8   {
9       int count;                    // Loop counter
10      double mainOfficeRequest;     // Main office budget request
11      const int NUM_DIVISIONS = 4;  // Number of divisions
12
13      // Get the main office's budget request.
14      // Note that no instances of the Budget class have been defined.
15      cout << "Enter the main office's budget request: ";
16      cin >> mainOfficeRequest;
17      Budget::mainOffice(mainOfficeRequest);
18
```

*(program continues)*

**Program 14-3** *(continued)*

```
19       Budget divisions[NUM_DIVISIONS]; // An array of Budget objects.
20
21       // Get the budget requests for each division.
22       for (count = 0; count < NUM_DIVISIONS; count++)
23       {
24            double budgetAmount;
25            cout << "Enter the budget request for division ";
26            cout << (count + 1) << ": ";
27            cin >> budgetAmount;
28            divisions[count].addBudget(budgetAmount);
29       }
30
31       // Display the budget requests and the corporate budget.
32       cout << fixed << showpoint << setprecision(2);
33       cout << "\nHere are the division budget requests:\n";
34       for (count = 0; count < NUM_DIVISIONS; count++)
35       {
36            cout << "\tDivision " << (count + 1) << "\t$ ";
37            cout << divisions[count].getDivisionBudget() << endl;
38       }
39       cout << "\tTotal Budget Requests:\t$ ";
40       cout << divisions[0].getCorpBudget() << endl;
41
42       return 0;
43  }
```

**Program Output with Example Input Shown in Bold**

```
Enter the main office's budget request: 100000 [Enter]
Enter the budget request for division 1: 100000 [Enter]
Enter the budget request for division 2: 200000 [Enter]
Enter the budget request for division 3: 300000 [Enter]
Enter the budget request for division 4: 400000 [Enter]

Here are the division budget requests:
     Division 1     $ 100000.00
     Division 2     $ 200000.00
     Division 3     $ 300000.00
     Division 4     $ 400000.00
    Total Requests (including main office): $ 1100000.00
```

Notice in line 17 the statement that calls the static function `mainOffice`:

```
Budget::mainOffice(amount);
```

Calls to static member functions do not use the regular notation of connecting the function name to an object name with the dot operator. Instead, static member functions are called by connecting the function name to the class name with the scope resolution operator.

**NOTE:** If an instance of a class with a static member function exists, the static member function can be called with the class object name and the dot operator, just like any other member function.

## 14.2 Friends of Classes

**CONCEPT:** **A friend is a function or class that is not a member of a class, but has access to the private members of the class.**

Private members are hidden from all parts of the program outside the class, and accessing them requires a call to a public member function. Sometimes you will want to create an exception to that rule. A *friend* function is a function that is not part of a class, but that has access to the class's private members. In other words, a friend function is treated as if it were a member of the class. A friend function can be a regular stand-alone function, or it can be a member of another class. (In fact, an entire class can be declared a friend of another class.)

In order for a function or class to become a friend of another class, it must be declared as such by the class granting it access. Classes keep a "list" of their friends, and only the external functions or classes whose names appear in the list are granted access. A function is declared a friend by placing the key word `friend` in front of a prototype of the function. Here is the general format:

```
friend ReturnType FunctionName (ParameterTypeList)
```

In the following declaration of the `Budget` class, the `addBudget` function of another class, `AuxiliaryOffice` has been declared a friend. (This file is stored in the Student Source Code Folder `Chapter 14\Budget Version 3`.)

### Contents of `Budget.h` (Version 3)

```
1  #ifndef BUDGET_H
2  #define BUDGET_H
3  #include "Auxil.h"
4
5  // Budget class declaration
6  class Budget
7  {
8  private:
9       static double corpBudget;   // Static member variable
10      double divisionBudget;      // Instance member variable
11 public:
12      Budget()
13           { divisionBudget = 0; }
14
15      void addBudget(double b)
16           { divisionBudget += b;
17             corpBudget += b; }
18
19      double getDivisionBudget() const
20           { return divisionBudget; }
21
22      double getCorpBudget() const
23           { return corpBudget; }
24
```

```
25      // Static member function
26      static void mainOffice(double);
27
28      // Friend function
29      friend void AuxiliaryOffice::addBudget(double, Budget &);
30  };
31
32  #endif
```

Let's assume another class, `AuxiliaryOffice`, represents a division's auxiliary office, perhaps in another country. The auxiliary office makes a separate budget request, which must be added to the overall corporate budget. The friend declaration of the `AuxiliaryOffice::addBudget` function tells the compiler that the function is to be granted access to `Budget`'s private members. Notice the function takes two arguments: a `double` and a reference object of the `Budget` class. The `Budget` class object that is to be modified by the function is passed to it, by reference, as an argument. The following code shows the declaration of the `AuxillaryOffice` class. (This file is stored in the Student Source Code Folder `Chapter 14\Budget Version 3`.)

### Contents of `Auxil.h`

```
1   #ifndef AUXIL_H
2   #define AUXIL_H
3
4   class Budget; // Forward declaration of Budget class
5
6   // Aux class declaration
7
8   class AuxiliaryOffice
9   {
10  private:
11      double auxBudget;
12  public:
13      AuxiliaryOffice()
14          { auxBudget = 0; }
15
16      double getDivisionBudget() const
17          { return auxBudget; }
18
19      void addBudget(double, Budget &);
20  };
21
22  #endif
```

### Contents of `Auxil.cpp`

```
1   #include "Auxil.h"
2   #include "Budget.h"
3
4   //*********************************************************
5   // Definition of member function mainOffice.              *
6   // This function is declared a friend by the Budget class.  *
7   // It adds the value of argument b to the static corpBudget *
8   // member variable of the Budget class.                   *
9   //*********************************************************
```

```
10
11  void AuxiliaryOffice::addBudget(double b, Budget &div)
12  {
13       auxBudget += b;
14       div.corpBudget += b;
15  }
```

Notice the `Auxil.h` file contains the following statement in line 4:

```
class Budget; // Forward declaration of Budget class
```

This is a *forward declaration* of the `Budget` class. It simply tells the compiler that a class named `Budget` will be declared later in the program. This is necessary because the compiler will process the `Auxil.h` file before it processes the `Budget` class declaration. When it is processing the `Auxil.h` file it will see the following function declaration in line 19:

```
void addBudget(double, Budget &);
```

The `addBudget` function's second parameter is a `Budget` reference variable. At this point, the compiler has not processed the `Budget` class declaration, so, without the forward declaration, it wouldn't know what a `Budget` reference variable is.

The following code shows the definition of the `addBudget` function. (This file is also stored in the Student Source Code Folder `Chapter 14\Budget Version 3`.)

### Contents of `Auxil.cpp`

```
1   #include "Auxil.h"
2   #include "Budget.h"
3
4   //*********************************************************
5   // Definition of member function mainOffice.              *
6   // This function is declared a friend by the Budget class.  *
7   // It adds the value of argument b to the static corpBudget *
8   // member variable of the Budget class.                   *
9   //*********************************************************
10
11  void AuxiliaryOffice::addBudget(double b, Budget &div)
12  {
13       auxBudget += b;
14       div.corpBudget += b;
15  }
```

The parameter `div`, a reference to a `Budget` class object, is used in line 14. This statement adds the parameter `b` to `div.corpBudget`. Program 14-4 demonstrates the classes.

**Program 14-4**

```
1   // This program demonstrates a static member function.
2   #include <iostream>
3   #include <iomanip>
4   #include "Budget.h"
5   using namespace std;
6
```

*(program continues)*

**Program 14-4** *(continued)*

```
 7  int main()
 8  {
 9       int count;                      // Loop counter
10       double mainOfficeRequest;       // Main office budget request
11       const int NUM_DIVISIONS = 4;    // Number of divisions
12
13       // Get the main office's budget request.
14       cout << "Enter the main office's budget request: ";
15       cin >> mainOfficeRequest;
16       Budget::mainOffice(mainOfficeRequest);
17
18       Budget divisions[NUM_DIVISIONS];  // Array of Budget objects
19       AuxiliaryOffice auxOffices[4];    // Array of AuxiliaryOffice
20
21       // Get the budget requests for each division
22       // and their auxiliary offices.
23       for (count = 0; count < NUM_DIVISIONS; count++)
24       {
25            double budgetAmount; // To hold input
26
27            // Get the request for the division office.
28            cout << "Enter the budget request for division ";
29            cout << (count + 1) << ": ";
30            cin >> budgetAmount;
31            divisions[count].addBudget(budgetAmount);
32
33            // Get the request for the auxiliary office.
34            cout << "Enter the budget request for that division's\n";
35            cout << "auxiliary office: ";
36            cin >> budgetAmount;
37            auxOffices[count].addBudget(budgetAmount, divisions[count]);
38       }
39
40       // Display the budget requests and the corporate budget.
41       cout << fixed << showpoint << setprecision(2);
42       cout << "\nHere are the division budget requests:\n";
43       for (count = 0; count < NUM_DIVISIONS; count++)
44       {
45            cout << "\tDivision " << (count + 1) << "\t\t$";
46            cout << divisions[count].getDivisionBudget() << endl;
47            cout << "\tAuxiliary office:\t$";
48            cout << auxOffices[count].getDivisionBudget() << endl << endl;
49       }
50       cout << "Total Budget Requests:\t$ ";
51       cout << divisions[0].getCorpBudget() << endl;
52       return 0;
53  }
```

**Program Output with Example Input Shown in Bold**

```
Enter the main office's budget request: 100000 [Enter]
Enter the budget request for division 1: 100000 [Enter]
Enter the budget request for that division's
auxiliary office: 50000 [Enter]
Enter the budget request for division 2: 200000 [Enter]
Enter the budget request for that division's
auxiliary office: 40000 [Enter]
Enter the budget request for division 3: 300000 [Enter]
Enter the budget request for that division's
auxiliary office: 70000 [Enter]
Enter the budget request for division 4: 400000 [Enter]
Enter the budget request for that division's
auxiliary office: 65000 [Enter]

Here are the division budget requests:
     Division 1                  $100000.00
     Auxiliary office:           $50000.00

     Division 2                  $200000.00
     Auxiliary office:           $40000.00

     Division 3                  $300000.00
     Auxiliary office:           $70000.00

     Division 4                  $400000.00
     Auxiliary office:           $65000.00

Total Budget Requests:  $ 1325000.00
```

As mentioned before, it is possible to make an entire class a friend of another class. The `Budget` class could make the `AuxiliaryOffice` class its friend with the following declaration:

```
friend class AuxiliaryOffice;
```

This may not be a good idea, however. Every member function of `AuxiliaryOffice` (including ones that may be added later) would have access to the private members of `Budget`. The best practice is to declare as friends only those functions that must have access to the private members of the class.

## Checkpoint

14.1 What is the difference between an instance member variable and a static member variable?

14.2 Static member variables are declared inside the class declaration. Where are static member variables defined?

14.3 Does a static member variable come into existence in memory before, at the same time as, or after any instances of its class?

14.4 What limitation does a static member function have?

14.5 What action is possible with a static member function that isn't possible with an instance member function?

14.6 If class `X` declares function `f` as a friend, does function `f` become a member of class `X`?

14.7 Class `Y` is a friend of class `X`, which means the member functions of class `Y` have access to the private members of class `X`. Does the friend key word appear in class `Y`'s declaration or in class `X`'s declaration?

## 14.3 Memberwise Assignment

**CONCEPT:** **The = operator may be used to assign one object's data to another object, or to initialize one object with another object's data. By default, each member of one object is copied to its counterpart in the other object.**

Like other variables (except arrays), objects may be assigned to one another using the = operator. As an example, consider Program 14-5, which uses the `Rectangle` class (version 4) that we discussed in Chapter 13. Recall that the `Rectangle` class has two member variables: `width` and `length`. The constructor accepts two arguments, one for `width` and one for `length`.

**Program 14-5**

```
1  // This program demonstrates memberwise assignment.
2  #include <iostream>
3  #include "Rectangle.h"
4  using namespace std;
5
6  int main()
7  {
8       // Define two Rectangle objects.
9       Rectangle box1(10.0, 10.0);   // width = 10.0, length = 10.0
10      Rectangle box2 (20.0, 20.0);  // width = 20.0, length = 20.0
11
12      // Display each object's width and length.
13      cout << "box1's width and length: " << box1.getWidth()
14           << " " << box1.getLength() << endl;
15      cout << "box2's width and length: " << box2.getWidth()
16           << " " << box2.getLength() << endl << endl;
17
18      // Assign the members of box1 to box2.
19      box2 = box1;
20
21      // Display each object's width and length again.
22      cout << "box1's width and length: " << box1.getWidth()
23           << " " << box1.getLength() << endl;
24      cout << "box2's width and length: " << box2.getWidth()
25           << " " << box2.getLength() << endl;
26
27      return 0;
28 }
```

**Program Output**

```
box1's width and length: 10 10
box2's width and length: 20 20

box1's width and length: 10 10
box2's width and length: 10 10
```

The following statement, which appears in line 19, copies the `width` and `length` member variables of `box1` directly into the `width` and `length` member variables of `box2`:

```
box2 = box1;
```

Memberwise assignment also occurs when one object is initialized with another object's values. Remember the difference between assignment and initialization: assignment occurs between two objects that already exist, and initialization happens to an object being created. Consider the following code:

```
// Define box1.
Rectangle box1(100.0, 50.0);

// Define box2, initialize with box1's values
Rectangle box2 = box1;
```

The last statement defines a `Rectangle` object, `box2`, and initializes it to the values stored in `box1`. Because memberwise assignment takes place, the `box2` object will contain the exact same values as the `box1` object.

## 14.4 Copy Constructors

**CONCEPT:** A copy constructor is a special constructor that is called whenever a new object is created and initialized with another object's data.

Most of the time, the default memberwise assignment behavior in C++ is perfectly acceptable. There are instances, however, where memberwise assignment cannot be used. For example, consider the following class. (This file is stored in the Student Source Code Folder `Chapter 14\StudentTestScores Version 1`.)

### Contents of `StudentTestScores.h` (Version 1)

```
1  #ifndef STUDENTTESTSCORES_H
2  #define STUDENTTESTSCORES_H
3  #include <string>
4  using namespace std;
5
6  const double DEFAULT_SCORE = 0.0;
7
8  class StudentTestScores
9  {
10 private:
11     string studentName; // The student's name
12     double *testScores; // Points to array of test scores
13     int numTestScores;  // Number of test scores
14
15     // Private member function to create an
16     // array of test scores.
17     void createTestScoresArray(int size)
18     { numTestScores = size;
19       testScores = new double[size];
20       for (int i = 0; i < size; i++)
21           testScores[i] = DEFAULT_SCORE;}
22
```

```
23  public:
24       // Constructor
25       StudentTestScores(string name, int numScores)
26       { studentName = name;
27         createTestScoresArray(numScores); }
28
29       // Destructor
30       ~StudentTestScores()
31       { delete [] testScores; }
32
33       // The setTestScore function sets a specific
34       // test score's value.
35       void setTestScore(double score, int index)
36       { testScores[index] = score; }
37
38       // Set the student's name.
39       void setStudentName(string name)
40       { studentName = name; }
41
42       // Get the student's name.
43       string getStudentName() const
44       { return studentName; }
45
46       // Get the number of test scores.
47       int getNumTestScores() const
48       { return numTestScores; }
49
50       // Get a specific test score.
51       double getTestScore(int index) const
52       { return testScores[index]; }
53  };
54  #endif
```

This class stores a student's name and a set of test scores. Let's take a closer look at the code:

- Lines 11 through 13 declare the class's attributes. The `studentName` attribute is a `string` object that holds a student's name. The `testScores` attribute is an `int` pointer. Its purpose is to point to a dynamically allocated `int` array that holds the student's test score. The `numTestScore` attribute is an `int` that holds the number of test scores.
- The `createTestScoresArray` private member function, in lines 17 through 21, creates an array to hold the student's test scores. It accepts an argument for the number of test scores, assigns this value to the `numTestScores` attribute (line 18), and then dynamically allocates an `int` array for the `testScores` attribute (line 19). The `for` loop in lines 20 through 21 initializes each element of the array to the default value 0.0.
- The constructor, in lines 25 through 27, accepts the student's name and the number of test scores as arguments. In line 26 the name is assigned to the `studentName` attribute, and in line 27 the number of test scores is passed to the `createTestScoresArray` member function.
- The destructor, in lines 30 through 31, deallocates the test score array.
- The `setTestScore` member function, in lines 35 through 36, sets a specific score in the `testScores` attribute. The function accepts arguments for the score and the index where the score should be stored in the `testScores` array.

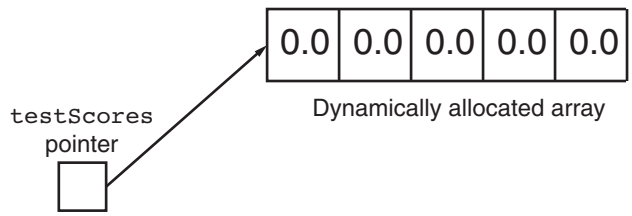
- The `setStudentName` member function, in lines 39 through 40, accepts an argument that is assigned to the `studentName` attribute.
- The `getStudentName` member function, in lines 43 through 44, returns the value of the `studentName` attribute.
- The `getNumTestScores` member function, in lines 47 through 48, returns the number of test scores stored in the object.
- The `getTestScore` member function, in lines 51 through 52, returns a specific score (specified by the index parameter) from the `testScores` attribute.

A potential problem with this class lies in the fact that one of its members, `testScores`, is a pointer. The `createTestScoresArray` member function (called by the constructor) performs a critical operation with the pointer: it dynamically allocates a section of memory for the `testScores` array and assigns default values to each of its element. For instance, the following statement creates a `StudentTestScores` object named `student1`, whose `testScores` member references dynamically allocated memory holding an array of 5 `double`'s:

```
StudentTestScores("Maria Jones Tucker", 5);
```

This is depicted in Figure 14-3.

**Figure 14-3**

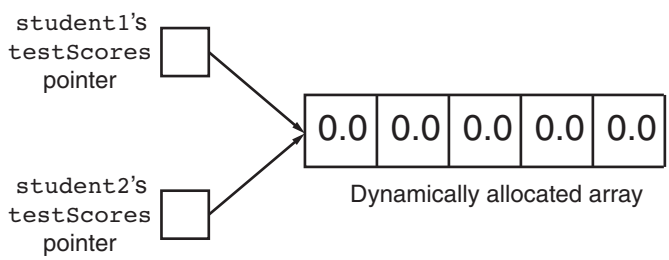



Consider what happens when another `StudentTestScores` object is created and initialized with the `student1` object, as in the following statement:

```
StudentTestScores student2 = student1;
```

In the statement above, `student2`'s constructor isn't called. Instead, memberwise assignment takes place, copying each of `student1`'s member variables into `student2`. This means that a separate section of memory is not allocated for `student2`'s `testScores` member. It simply gets a copy of the address stored in `student1`'s `testScores` member. Both pointers will point to the same address, as depicted in Figure 14-4.

**Figure 14-4**



In this situation, either object can manipulate the values stored in the array, causing the changes to show up in the other object. Likewise, one object can be destroyed, causing its destructor to be called, which frees the allocated memory. The remaining object's `testScores` pointer would still reference this section of memory, although it should no longer be used.

The solution to this problem is to create a *copy constructor* for the object. A copy constructor is a special constructor that's called when an object is initialized with another object's data. It has the same form as other constructors, except it has a reference parameter of the same class type as the object itself. For example, here is a copy constructor for the `StudentTestScores` class:

```
StudentTestScores(StudentTestScores &obj)
{ studentName = obj.studentName;
  numTestScores = obj.numTestScores;
  testScores = new double[numTestScores];
  for (int i = 0; i < length; i++)
       testScores[i] = obj.testScores[i]; }
```

When the = operator is used to initialize a `StudentTestScores` object with the contents of another `StudentTestScores` object, the copy constructor is called. The `StudentTestScores` object that appears on the right side of the = operator is passed as an argument to the copy constructor. For example, look at the following statement:

```
StudentTestScores student1 ("Molly McBride", 8);
StudentTestScores student2 = student1;
```

In this code, the `student1` object is passed as an argument to the `student2` object's copy constructor.

**NOTE:** C++ requires that a copy constructor's parameter be a reference object.

As you can see from studying the copy constructor's code, `student2`'s `testScores` member will properly reference its own dynamically allocated memory. There will be no danger of `student1` inadvertently destroying or corrupting `student2`'s data.

## Using `const` Parameters in Copy Constructors

Because copy constructors are required to use reference parameters, they have access to their argument's data. Since the purpose of a copy constructor is to make a copy of the argument, there is no reason the constructor should modify the argument's data. With this in mind, it's a good idea to make copy constructors' parameters constant by specifying the `const` key word in the parameter list. Here is an example:

```
StudentTestScores(const StudentTestScores &obj)
{ studentName = obj.studentName;
  numTestScores = obj.numTestScores;
  testScores = new double[numTestScores];
  for (int i = 0; i < numTestScores; i++)
       testScores[i] = obj.testScores[i]; }
```

The `const` key word ensures that the function cannot change the contents of the parameter. This will prevent you from inadvertently writing code that corrupts data.

The complete listing for the revised `StudentTestScores` class is shown here. (This file is stored in the Student Source Code Folder `Chapter 14\StudentTestScores Version 2`.)

### Contents of `StudentTestScores.h` (Version 2)

```
#ifndef STUDENTTESTSCORES_H
#define STUDENTTESTSCORES_H
#include <string>
using namespace std;

const double DEFAULT_SCORE = 0.0;

class StudentTestScores
{
private:
    string studentName;  // The student's name
    double *testScores;  // Points to array of test scores
    int numTestScores;   // Number of test scores

     // Private member function to create an
     // array of test scores.
    void createTestScoresArray(int size)
    { numTestScores = size;
      testScores = new double[size];
      for (int i = 0; i < size; i++)
          testScores[i] = DEFAULT_SCORE; }

public:
    // Constructor
    StudentTestScores(string name, int numScores)
    { studentName = name;
      createTestScoresArray(numScores); }

    // Copy constructor
    StudentTestScores(const StudentTestScores &obj)
    { studentName = obj.studentName;
      numTestScores = obj.numTestScores;
      testScores = new double[numTestScores];
      for (int i = 0; i < numTestScores; i++)
          testScores[i] = obj.testScores[i]; }

    // Destructor
    ~StudentTestScores()
    { delete [] testScores; }

    // The setTestScore function sets a specific
    // test score's value.
    void setTestScore(double score, int index)
    { testScores[index] = score; }

```

```
46       // Set the student's name.
47       void setStudentName(string name)
48       { studentName = name; }
49
50       // Get the student's name.
51       string getStudentName() const
52       { return studentName; }
53
54       // Get the number of test scores.
55       int getNumTestScores() const
56       { return numTestScores; }
57
58       // Get a specific test score.
59       double getTestScore(int index) const
60       { return testScores[index]; }
61   };
62   #endif
```

## Copy Constructors and Function Parameters

When a class object is passed by value as an argument to a function, it is passed to a parameter that is also a class object, and the copy constructor of the function's parameter is called. Remember that when a nonreference class object is used as a function parameter, it is created when the function is called, and it is initialized with the argument's value.

This is why C++ requires the parameter of a copy constructor to be a reference object. If an object were passed to the copy constructor by value, the copy constructor would create a copy of the argument and store it in the parameter object. When the parameter object is created, its copy constructor will be called, thus causing another parameter object to be created. This process will continue indefinitely (or at least until the available memory fills up, causing the program to halt).

To prevent the copy constructor from calling itself an infinite number of times, C++ requires its parameter to be a reference object.

## The Default Copy Constructor

Although you may not realize it, you have seen the action of a copy constructor before. If a class doesn't have a copy constructor, C++ creates a *default copy constructor* for it. The default copy constructor performs the memberwise assignment discussed in the previous section.

### Checkpoint

14.8 Briefly describe what is meant by memberwise assignment.

14.9 Describe two instances when memberwise assignment occurs.

14.10 Describe a situation in which memberwise assignment should not be used.

14.11 When is a copy constructor called?

14.12 How does the compiler know that a member function is a copy constructor?

14.13 What action is performed by a class's default copy constructor?

## 14.5 Operator Overloading

**CONCEPT:** **C++ allows you to redefine how standard operators work when used with class objects.**


VideoNote
**Operator Overloading**

C++ provides many operators to manipulate data of the primitive data types. However, what if you wish to use an operator to manipulate class objects? For example, assume that a class named `Date` exists, and objects of the `Date` class hold the month, day, and year in member variables. Suppose the `Date` class has a member function named `add`. The `add` member function adds a number of days to the date and adjusts the member variables if the date goes to another month or year. For example, the following statement adds five days to the date stored in the `today` object:

```
today.add(5);
```

Although it might be obvious that the statement is adding five days to the date stored in `today`, the use of an operator might be more intuitive. For example, look at the following statement:

```
today += 5;
```

This statement uses the standard `+=` operator to add 5 to `today`. This behavior does not happen automatically, however. The `+=` operator must be *overloaded* for this action to occur. In this section, you will learn to overload many of C++'s operators to perform specialized operations on class objects.

**NOTE:** You have already experienced the behavior of an overloaded operator. The / operator performs two types of division: floating point and integer. If one of the / operator's operands is a floating point type, the result will be a floating point value. If both of the / operator's operands are integers, however, a different behavior occurs: the result is an integer and any fractional part is thrown away.

### Overloading the = Operator

Although copy constructors solve the initialization problems inherent with objects containing pointer members, they do not work with simple assignment statements. Copy constructors are just that—constructors. They are only invoked when an object is created. Statements like the following still perform memberwise assignment:

```
student2 = student1;
```

In order to change the way the assignment operator works, it must be overloaded. Operator overloading permits you to redefine an existing operator's behavior when used with a class object.

C++ allows a class to have special member functions called *operator functions*. If you wish to redefine the way a particular operator works with an object, you define a function for that operator. The Operator function is then executed any time the operator is used with an object of that class. For example, the following version of the `StudentTestScores` class overloads the `=` operator. (This file is stored in the Student Source Code Folder `Chapter 14\StudentTestScores Version 3`.)

### Contents of `StudentTestScores` (Version 3)

```
#ifndef STUDENTTESTSCORES_H
#define STUDENTTESTSCORES_H
#include <string>
using namespace std;

const double DEFAULT_SCORE = 0.0;

class StudentTestScores
{
private:
    string studentName;  // The student's name
    double *testScores;  // Points to array of test scores
    int numTestScores;   // Number of test scores

    // Private member function to create an
    // array of test scores.
    void createTestScoresArray(int size)
    { numTestScores = size;
      testScores = new double[size];
      for (int i = 0; i < size; i++)
          testScores[i] = DEFAULT_SCORE; }

public:
    // Constructor
    StudentTestScores(string name, int numScores)
    { studentName = name;
     createTestScoresArray(numScores); }

    // Copy constructor
    StudentTestScores(const StudentTestScores &obj)
    { studentName = obj.studentName;
      numTestScores = obj.numTestScores;
      testScores = new double[numTestScores];
      for (int i = 0; i < numTestScores; i++)
          testScores[i] = obj.testScores[i]; }

    // Destructor
    ~StudentTestScores()
    { delete [] testScores; }

    // The setTestScore function sets a specific
    // test score's value.
    void setTestScore(double score, int index)
    { testScores[index] = score; }

    // Set the student's name.
    void setStudentName(string name)
    { studentName = name; }

    // Get the student's name.
    string getStudentName() const
    { return studentName; }

```

```
54       // Get the number of test scores.
55       int getNumTestScores()
56       { return numTestScores; }
57
58       // Get a specific test score.
59       double getTestScore(int index) const
60       { return testScores[index]; }
61
62       // Overloaded = operator
63       void operator=(const StudentTestScores &right)
64       { delete [] testScores;
65         studentName = right.studentName;
66         numTestScores = right.numTestScores;
67         testScores = new double[numTestScores];
68         for (int i = 0; i < numTestScores; i++)
69           testScores[i] = right.testScores[i]; }
70   };
71   #endif
```

Let's examine the operator function to understand how it works. First look at the function header:

Return type — Function name — Parameter for object on the right side of operator

```
void operator = (const studnetTestScores &right)
```

The name of the function is `operator=`. This specifies that the function overloads the = operator. Because it is a member of the `StudentTestScores` class, this function will be called only when an assignment statement executes where the object on the left side of the = operator is a `StudentTestScores` object.

**NOTE:** You can, if you choose, put spaces around the operator symbol. For instance, the function header above could also read:

```
void operator = (const StudentTestScores &right)
```

The function has one parameter: a constant reference object named `right`. This parameter references the object on the right side of the operator. For example, when the following statement is executed, `right` will reference the `student1` object:

```
student2 = student1;
```

It is not required that the parameter of an operator function be a reference object. The `StudentTestScores` example declares `right` as a `const` reference for the following reasons:

- It was declared as a reference for efficiency purposes. This prevents the compiler from making a copy of the object being passed into the function.
- It was declared constant so the function will not accidentally change the contents of the argument.

**NOTE:** In the example, the parameter was named `right` simply to illustrate that it references the object on the right side of the operator. You can name the parameter anything you wish. It will always take the object on the operator's right as its argument.

In learning the mechanics of operator overloading, it is helpful to know that the following two statements do the same thing:

```
student2 = student1;           // Call operator= function
student2.operator=(student1);  // Call operator= function
```

In the last statement you can see exactly what is going on in the function call. The `student1` object is being passed to the function's parameter, `right`. Inside the function, the values in `right`'s members are used to initialize `student2`. Notice that the `operator=` function has access to the `right` parameter's private members. Because the `operator=` function is a member of the `StudentTestScores` class, it has access to the private members of any `StudentTestScores` object that is passed into it.

**NOTE:** C++ allows operator functions to be called with regular function call notation, or by using the operator symbol.

Program 14-6 demonstrates the `StudentTestScores` class with its overloaded assignment operator. (This file is stored in the Student Source Code Folder `Chapter 14\StudentTestScores Version 3`.)

**Program 14-6**

```
1  // This program demonstrates the overloaded = operator
2  #include <iostream>
3  #include "StudentTestScores.h"
4  using namespace std;
5
6  // Function prototype
7  void displayStudent(StudentTestScores);
8
9  int main()
10 {
11     // Create a StudentTestScores object and
12     // assign test scores.
13     StudentTestScores student1("Kelly Thorton", 3);
14     student1.setTestScore(100.0, 0);
15     student1.setTestScore(95.0, 1);
16     student1.setTestScore(80, 2);
17
18     // Create another StudentTestScore object
19     // with default test scores.
20     StudentTestScores student2("Jimmy Griffin", 5);
21
22     // Assign the student1 object to student2
23     student2 = student1;
24
25     // Display both objects. They should
26     // contain the same data.
27     displayStudent(student1);
28     displayStudent(student2);
29     return 0;
30 }
31
```

```
32  // The displayStudent function accepts a
33  // StudentTestScores object's data.
34  void displayStudent(StudentTestScores s)
35  {
36       cout << "Name: " << s.getStudentName() << endl;
37       cout << "Test Scores: ";
38       for (int i = 0; i < s.getNumTestScores(); i++)
39            cout << s.getTestScore(i) << " ";
40       cout << endl;
41  }
```

**Program Output**

```
Name: Kelly Thorton
Test Scores: 100 95 80
Name: Kelly Thorton
Test Scores: 100 95 80
```

## The = Operator's Return Value

There is only one problem with the overloaded = operator shown in Program 14-6: it has a `void` return type. C++'s built-in = operator allows multiple assignment statements such as:

```
a = b = c;
```

In this statement, the expression `b = c` causes `c` to be assigned to `b` and then returns the value of `c`. The return value is then stored in `a`. If a class object's overloaded = operator is to function this way, it too must have a valid return type.

For example, the `StudentTestScores` class's `operator=` function could be written as:

```
const StudentTestScores operator=(const StudentTestScores &right)
{ delete [] testScores;
  studentName = right.studentName;
  numTestScores = right.numTestScores;
  testScores = new double[numTestScores];
  for (int i = 0; i < numTestScores; i++)
       testScores[i] = right.testScores[i];
  return *this; }
```

The data type of the operator function specifies that a `const StudentTestScores` object is returned. Look at the last statement in the function:

```
return *this;
```

This statement returns the value of a dereferenced pointer: `this`. But what is `this`? Read on.

## The `this` Pointer

The `this` pointer is a special built-in pointer that is available to a class's member functions. It always points to the instance of the class making the function call. For example, if `student1` and `student2` are both `StudentTestScores` objects, the following statement causes the `getStudentName` function to operate on `student1`:

```
cout << student1.getStudentName() << endl;
```

Likewise, the following statement causes `getStudentName` to operate on `student2`:

```
cout << student2.getStudentName() << endl;
```

When `getStudentName` is operating on `student1`, the `this` pointer is pointing to `student1`. When `getStudentName` is operating on `student2`, `this` is pointing to `student2`. The `this` pointer always points to the object that is being used to call the member function.

**NOTE:** The `this` pointer is passed as a hidden argument to all nonstatic member functions.

The overloaded = operator function is demonstrated in Program 14-7. The multiple assignment statement in line 22 causes the `operator=` function to execute. (This file and the revised version of the `StudentTestScores` class is stored in the Student Source Code Folder `Chapter 14\StudentTestScores Version 4`.)

**Program 14-7**

```
1  // This program demonstrates the overloaded = operator returning a value.
2  #include <iostream>
3  #include "StudentTestScores.h"
4  using namespace std;
5
6  // Function prototype
7  void displayStudent(StudentTestScores);
8
9  int main()
10 {
11     // Create a StudentTestScores object.
12     StudentTestScores student1("Kelly Thorton", 3);
13     student1.setTestScore(100.0, 0);
14     student1.setTestScore(95.0, 1);
15     student1.setTestScore(80, 2);
16
17     // Create two more StudentTestScores objects.
18     StudentTestScores student2("Jimmy Griffin", 5);
19     StudentTestScores student3("Kristen Lee", 10);
20
21     // Assign student1 to student2 and student3.
22     student3 = student2 = student1;
23
24     // Display the objects.
25     displayStudent(student1);
26     displayStudent(student2);
27     displayStudent(student3);
28     return 0;
29 }
30
31 // displayStudent function
32 void displayStudent(StudentTestScores s)
```

```
33  {
34       cout << "Name: " << s.getStudentName() << endl;
35       cout << "Test Scores: ";
36       for (int i = 0; i < s.getNumTestScores(); i++)
37           cout << s.getTestScore(i) << " ";
38       cout << endl;
39  }
```

**Program Output**

```
Name: Kelly Thorton
Test Scores: 100 95 80
Name: Kelly Thorton
Test Scores: 100 95 80
Name: Kelly Thorton
Test Scores: 100 95 80
```

## Some General Issues of Operator Overloading

Now that you have had a taste of operator overloading, let's look at some of the general issues involved in this programming technique.

Although it is not a good programming practice, you can change an operator's entire meaning if that's what you wish to do. There is nothing to prevent you from changing the = symbol from an assignment operator to a "display" operator. For instance, the following class does just that:

```
class Weird
{
private:
   int value;
public:
   Weird(int v)
      { value = v; }
   void operator=(const weird &right)
      { cout << right.value << endl; }
};
```

Although the `operator=` function in the `Weird` class overloads the assignment operator, the function doesn't perform an assignment. Instead, it displays the contents of `right.value`. Consider the following program segment:

```
Weird a(5), b(10);
a = b;
```

Although the statement `a = b` looks like an assignment statement, it actually causes the contents of `b`'s `value` member to be displayed on the screen:

```
10
```

Another operator overloading issue is that you cannot change the number of operands taken by an operator. The = symbol must always be a binary operator. Likewise, ++ and -- must always be unary operators.

The last issue is that although you may overload most of the C++ operators, you cannot overload all of them. Table 14-1 shows all of the C++ operators that may be overloaded.

**Table 14-1**

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| + | - | * | / | % | ^ | & | \| | ~ | ! | = | < |
| > | += | -= | *= | /= | %= | ^= | &= | \|= | << | >> | >>= |
| <<= | == | != | <= | >= | && | \|\| | ++ | -- | ->* | , | -> |
| [] | () | new | delete | | | | | | | | |

**NOTE:** Some of the operators in Table 14-1 are beyond the scope of this book and are not covered.

The only operators that cannot be overloaded are

```
?:    .    .*    ::    sizeof
```

## Overloading Math Operators

Many classes would benefit not only from an overloaded assignment operator, but also from overloaded math operators. To illustrate this, consider the `FeetInches` class shown in the following two files. (These files are stored in the Student Source Code Folder `Chapter 14\FeetInches Version 1`.)

### Contents of `FeetInches.h` (Version 1)

```
1  #ifndef FEETINCHES_H
2  #define FEETINCHES_H
3
4  // The FeetInches class holds distances or measurements
5  // expressed in feet and inches.
6
7  class FeetInches
8  {
9  private:
10     int feet;          // To hold a number of feet
11     int inches;        // To hold a number of inches
12     void simplify();   // Defined in FeetInches.cpp
13 public:
14     // Constructor
15     FeetInches(int f = 0, int i = 0)
16          { feet = f;
17            inches = i;
18            simplify(); }
19
20     // Mutator functions
21     void setFeet(int f)
22        { feet = f; }
23
```

```
24      void setInches(int i)
25         { inches = i;
26           simplify(); }
27
28      // Accessor functions
29      int getFeet() const
30         { return feet; }
31
32      int getInches() const
33         { return inches; }
34
35      // Overloaded operator functions
36      FeetInches operator + (const FeetInches &); // Overloaded +
37      FeetInches operator - (const FeetInches &); // Overloaded -
38  };
39
40  #endif
```

### Contents of `FeetInches.cpp` (Version 1)

```
 1  // Implementation file for the FeetInches class
 2  #include <cstdlib>  // Needed for abs()
 3  #include "FeetInches.h"
 4
 5  //*************************************************************
 6  // Definition of member function simplify. This function      *
 7  // checks for values in the inches member greater than        *
 8  // twelve or less than zero. If such a value is found,        *
 9  // the numbers in feet and inches are adjusted to conform     *
10  // to a standard feet & inches expression. For example,       *
11  // 3 feet 14 inches would be adjusted to 4 feet 2 inches and  *
12  // 5 feet -2 inches would be adjusted to 4 feet 10 inches.    *
13  //*************************************************************
14
15  void FeetInches::simplify()
16  {
17      if (inches >= 12)
18      {
19          feet += (inches / 12);
20          inches = inches % 12;
21      }
22      else if (inches < 0)
23      {
24          feet -= ((abs(inches) / 12) + 1);
25          inches = 12 - (abs(inches) % 12);
26      }
27  }
28
29  //*****************************************
30  // Overloaded binary + operator.          *
31  //*****************************************
32
```

```
33  FeetInches FeetInches::operator + (const FeetInches &right)
34  {
35      FeetInches temp;
36
37      temp.inches = inches + right.inches;
38      temp.feet = feet + right.feet;
39      temp.simplify();
40      return temp;
41  }
42
43  //*****************************************
44  // Overloaded binary - operator.          *
45  //*****************************************
46
47  FeetInches FeetInches::operator - (const FeetInches &right)
48  {
49      FeetInches temp;
50
51      temp.inches = inches - right.inches;
52      temp.feet = feet - right.feet;
53      temp.simplify();
54      return temp;
55  }
```

The `FeetInches` class is designed to hold distances or measurements expressed in feet and inches. It consists of eight member functions:

- A constructor that allows the `feet` and `inches` members to be set. The default values for these members is zero.
- A `setFeet` function for storing a value in the `feet` member.
- A `setInches` function for storing a value in the `inches` member.
- A `getFeet` function for returning the value in the `feet` member.
- A `getInches` function for returning the value in the `inches` member.
- A `simplify` function for normalizing the values held in `feet` and `inches`. This function adjusts any set of values where the `inches` member is greater than 12 or less than 0.
- An `operator +` function that overloads the standard + math operator.
- An `operator -` function that overloads the standard - math operator.

**NOTE:** The `simplify` function uses the standard library function `abs()` to get the absolute value of the `inches` member. The `abs()` function requires that `cstdlib` be included.

The overloaded + and - operators allow one `FeetInches` object to be added to or subtracted from another. For example, assume the `length1` and `length2` objects are defined and initialized as follows:

```
FeetInches length1(3, 5), length2(6, 3);
```

The `length1` object is holding the value 3 feet 5 inches, and the `length2` object is holding the value 6 feet 3 inches. Because the + operator is overloaded, we can add these two objects in a statement such as:

```
length3 = length1 + length2;
```

This statement will add the values of the `length1` and `length2` objects and store the result in the `length3` object. After the statement executes, the `length3` object will be set to 9 feet 8 inches.

The member function that overloads the + operator appears in lines 33 through 41 of the `FeetInches.cpp` file.

This function is called anytime the + operator is used with two `FeetInches` objects. Just like the overloaded = operator we defined in the previous section, this function has one parameter: a constant reference object named `right`. This parameter references the object on the right side of the operator. For example, when the following statement is executed, `right` will reference the `length2` object:

```
length3 = length1 + length2;
```

As before, it might be helpful to think of the statement above as the following function call:

```
length3 = length1.operator+(length2);
```

The `length2` object is being passed to the function's parameter, `right`. When the function finishes, it will return a `FeetInches` object to `length3`. Now let's see what is happening inside the function. First, notice that a `FeetInches` object named `temp` is defined locally in line 35:

```
FeetInches temp;
```

This object is a temporary location for holding the results of the addition. Next, line 37 adds `inches` to `right.inches` and stores the result in `temp.inches`:

```
temp.inches = inches + right.inches;
```

The `inches` variable is a member of `length1`, the object making the function call. It is the object on the left side of the operator. `right.inches` references the `inches` member of `length2`. The next statement, in line 38, is very similar. It adds `feet` to `right.feet` and stores the result in `temp.feet`:

```
temp.feet = feet + right.feet;
```

At this point in the function, `temp` contains the sum of the `feet` and `inches` members of both objects in the expression. The next step is to adjust the values so they conform to a normal value expressed in feet and inches. This is accomplished in line 39 by calling `temp.simplify()`:

```
temp.simplify();
```

The last step, in line 40, is to return the value stored in `temp`:

```
return temp;
```

In the statement `length3 = length1 + length2`, the `return` statement in the operator function causes the values stored in `temp` to be returned to the `length3` object.

Program 14-8 demonstrates the overloaded operators. (This file is stored in the student source code folder `Chapter 14\FeetInches Version 1`.)

**Program 14-8**

```
// This program demonstrates the FeetInches class's overloaded
// + and - operators.
#include <iostream>
#include "FeetInches.h"
using namespace std;

int main()
{
     int feet, inches; // To hold input for feet and inches

     // Create three FeetInches objects. The default arguments
     // for the constructor will be used.
     FeetInches first, second, third;

     // Get a distance from the user.
     cout << "Enter a distance in feet and inches: ";
     cin >> feet >> inches;

     // Store the distance in the first object.
     first.setFeet(feet);
     first.setInches(inches);

     // Get another distance from the user.
     cout << "Enter another distance in feet and inches: ";
     cin >> feet >> inches;

     // Store the distance in second.
     second.setFeet(feet);
     second.setInches(inches);

     // Assign first + second to third.
     third = first + second;

     // Display the result.
     cout << "first + second = ";
     cout << third.getFeet() << " feet, ";
     cout << third.getInches() << " inches.\n";

     // Assign first - second to third.
     third = first - second;

     // Display the result.
     cout << "first - second = ";
     cout << third.getFeet() << " feet, ";
     cout << third.getInches() << " inches.\n";

     return 0;
}
```

**Program Output with Example Input Shown in Bold**

```
Enter a distance in feet and inches: 6 5 [Enter]
Enter another distance in feet and inches: 3 10 [Enter]
first + second = 10 feet, 3 inches.
first - second = 2 feet, 7 inches.
```

## Overloading the Prefix ++ Operator

Unary operators, such as ++ and – –, are overloaded in a fashion similar to the way binary operators are implemented. Because unary operators only affect the object making the operator function call, however, there is no need for a parameter. For example, let's say you wish to have a prefix increment operator for the `FeetInches` class. Assume the `FeetInches` object `distance` is set to the values 7 feet and 5 inches. A ++ operator function could be designed to increment the object's `inches` member. The following statement would cause `distance` to have the value 7 feet 6 inches:

```
++distance;
```

The following function overloads the prefix ++ operator to work in this fashion:

```
FeetInches FeetInches::operator++()
{
   ++inches;
   simplify();
   return *this;
}
```

This function first increments the object's `inches` member. The `simplify()` function is called and then the dereferenced `this` pointer is returned. This allows the operator to perform properly in statements like this:

```
distance2 = ++distance1;
```

Remember, the statement above is equivalent to

```
distance2 = distance1.operator++();
```

## Overloading the Postfix ++ Operator

Overloading the postfix ++ operator is only slightly different than overloading the prefix version. Here is the function that overloads the postfix operator with the `FeetInches` class:

```
FeetInches FeetInches::operator++(int)
{
   FeetInches temp(feet, inches);
   inches++;
   simplify();
   return temp;
}
```

The first difference you will notice is the use of a *dummy parameter*. The word `int` in the function's parentheses establishes a nameless integer parameter. When C++ sees this parameter in an operator function, it knows the function is designed to be used in postfix mode. The second difference is the use of a temporary local variable, the `temp` object. `temp`

is initialized with the `feet` and `inches` values of the object making the function call. `temp`, therefore, is a copy of the object being incremented, but before the increment takes place. After `inches` is incremented and the `simplify` function is called, the contents of `temp` is returned. This causes the postfix operator to behave correctly in a statement like this:

```
distance2 = distance1++;
```

You will find a version of the `FeetInches` class with the overloaded prefix and postfix `++` operators stored in the Student Source Code Folder `Chapter 14\FeetInches Version 2`. In that folder you will also find Program 14-9, which demonstrates these overloaded operators.

**Program 14-9**

```
// This program demonstrates the FeetInches class's overloaded
// prefix and postfix ++ operators.
#include <iostream>
#include "FeetInches.h"
using namespace std;

int main()
{
     int count;  // Loop counter

     // Define a FeetInches object with the default
     // value of 0 feet, 0 inches.
     FeetInches first;

     // Define a FeetInches object with 1 foot 5 inches.
     FeetInches second(1, 5);

     // Use the prefix ++ operator.
     cout << "Demonstrating prefix ++ operator.\n";
     for (count = 0; count < 12; count++)
     {
          first = ++second;
          cout << "first: " << first.getFeet() << " feet, ";
          cout << first.getInches() << " inches. ";
          cout << "second: " << second.getFeet() << " feet, ";
          cout << second.getInches() << " inches.\n";
     }

     // Use the postfix ++ operator.
     cout << "\nDemonstrating postfix ++ operator.\n";
     for (count = 0; count < 12; count++)
     {
          first = second++;
          cout << "first: " << first.getFeet() << " feet, ";
          cout << first.getInches() << " inches. ";
          cout << "second: " << second.getFeet() << " feet, ";
          cout << second.getInches() << " inches.\n";
     }

     return 0;
}
```

**Program Output**

```
Demonstrating prefix ++ operator.
first: 1 feet 6 inches. second: 1 feet 6 inches.
first: 1 feet 7 inches. second: 1 feet 7 inches.
first: 1 feet 8 inches. second: 1 feet 8 inches.
first: 1 feet 9 inches. second: 1 feet 9 inches.
first: 1 feet 10 inches. second: 1 feet 10 inches.
first: 1 feet 11 inches. second: 1 feet 11 inches.
first: 2 feet 0 inches. second: 2 feet 0 inches.
first: 2 feet 1 inches. second: 2 feet 1 inches.
first: 2 feet 2 inches. second: 2 feet 2 inches.
first: 2 feet 3 inches. second: 2 feet 3 inches.
first: 2 feet 4 inches. second: 2 feet 4 inches.
first: 2 feet 5 inches. second: 2 feet 5 inches.

Demonstrating postfix ++ operator.
first: 2 feet 5 inches. second: 2 feet 6 inches.
first: 2 feet 6 inches. second: 2 feet 7 inches.
first: 2 feet 7 inches. second: 2 feet 8 inches.
first: 2 feet 8 inches. second: 2 feet 9 inches.
first: 2 feet 9 inches. second: 2 feet 10 inches.
first: 2 feet 10 inches. second: 2 feet 11 inches.
first: 2 feet 11 inches. second: 3 feet 0 inches.
first: 3 feet 0 inches. second: 3 feet 1 inches.
first: 3 feet 1 inches. second: 3 feet 2 inches.
first: 3 feet 2 inches. second: 3 feet 3 inches.
first: 3 feet 3 inches. second: 3 feet 4 inches.
first: 3 feet 4 inches. second: 3 feet 5 inches.
```

## Checkpoint

14.14 Assume there is a class named `Pet`. Write the prototype for a member function of `Pet` that overloads the = operator.

14.15 Assume that `dog` and `cat` are instances of the `Pet` class, which has overloaded the = operator. Rewrite the following statement so it appears in function call notation instead of operator notation:

```
dog = cat;
```

14.16 What is the disadvantage of an overloaded = operator returning `void`?

14.17 Describe the purpose of the `this` pointer.

14.18 The `this` pointer is automatically passed to what type of functions?

14.19 Assume there is a class named `Animal` that overloads the = and + operators. In the following statement, assume `cat`, `tiger`, and `wildcat` are all instances of the `Animal` class:

```
wildcat = cat + tiger;
```

Of the three objects, `wildcat`, `cat`, or `tiger`, which is calling the `operator+` function? Which object is passed as an argument into the function?

14.20 What does the use of a dummy parameter in a unary operator function indicate to the compiler?

## Overloading Relational Operators

In addition to the assignment and math operators, relational operators may be overloaded. This capability allows classes to be compared in statements that use relational expressions such as:

```
if (distance1 < distance2)
{
   ... code ...
}
```

Overloaded relational operators are implemented like other binary operators. The only difference is that a relational operator function should always return a `true` or `false` value. The `FeetInches` class in the Student Source Code Folder `Chapter 14\FeetInches Version 3` contains functions to overload the >, <, and == relational operators. Here is the function for overloading the > operator:

```
bool FeetInches::operator > (const FeetInches &right)
{
   bool status;

   if (feet > right.feet)
      status = true;
   else if (feet == right.feet &&  inches > right.inches)
      status = true;
   else
      status = false;

   return status;
}
```

As you can see, the function compares the `feet` member (and if necessary, the `inches` member) with that of the parameter. If the calling object contains a value greater than that of the parameter, `true` is returned. Otherwise, `false` is returned.

Here is the code that overloads the < operator:

```
bool FeetInches::operator < (const FeetInches &right)
{
   bool status;

   if (feet < right.feet)
      status = true;
   else if (feet == right.feet &&  inches < right.inches)
      status = true;
   else
      status = false;

   return status;
}
```

Here is the code that overloads the == operator:

```
bool FeetInches::operator == (const FeetInches &right)
{
   bool status;

   if (feet == right.feet &&  inches == right.inches)
      status = true;
   else
      status = false;
```

```
        return status;
    }
```

Program 14-10 demonstrates these overloaded operators. (This file is also stored in the Student Source Code Folder `Chapter 14\FeetInches Version 3`.)

**Program 14-10**

```
1  // This program demonstrates the FeetInches class's overloaded
2  // relational operators.
3  #include <iostream>
4  #include "FeetInches.h"
5  using namespace std;
6
7  int main()
8  {
9      int feet, inches; // To hold input for feet and inches
10
11     // Create two FeetInches objects. The default arguments
12     // for the constructor will be used.
13     FeetInches first, second;
14
15     // Get a distance from the user.
16     cout << "Enter a distance in feet and inches: ";
17     cin >> feet >> inches;
18
19     // Store the distance in first.
20     first.setFeet(feet);
21     first.setInches(inches);
22
23     // Get another distance.
24     cout << "Enter another distance in feet and inches: ";
25     cin >> feet >> inches;
26
27     // Store the distance in second.
28     second.setFeet(feet);
29     second.setInches(inches);
30
31     // Compare the two objects.
32     if (first == second)
33         cout << "first is equal to second.\n";
34     if (first > second)
35         cout << "first is greater than second.\n";
36     if (first < second)
37         cout << "first is less than second.\n";
38
39     return 0;
40 }
```

**Program Output with Example Input Shown in Bold**

```
Enter a distance in feet and inches: 6 5 [Enter]
Enter another distance in feet and inches: 3 10 [Enter]
first is greater than second.
```

*(program output continues)*

**Program 14-10** *(continued)*

**Program Output with Different Example Input Shown in Bold**

```
Enter a distance in feet and inches: 5 5 [Enter]
Enter another distance in feet and inches: 5 5 [Enter]
first is equal to second.
```

**Program Output with Different Example Input Shown in Bold**

```
Enter a distance in feet and inches: 3 4 [Enter]
Enter another distance in feet and inches: 3 7 [Enter]
first is less than second.
```

## Overloading the << and >> Operators

Overloading the math and relational operators gives you the ability to write those types of expressions with class objects just as naturally as with integers, floats, and other built-in data types. If an object's primary data members are private, however, you still have to make explicit member function calls to send their values to `cout`. For example, assume `distance` is a `FeetInches` object. The following statements display its internal values:

```
cout << distance.getFeet() << " feet, ";
cout << distance.getInches() << "inches";
```

It is also necessary to explicitly call member functions to set a `FeetInches` object's data. For instance, the following statements set the `distance` object to user-specified values:

```
cout << "Enter a value in feet: ";
cin >> f;
distance.setFeet(f);
cout << "Enter a value in inches: ";
cin >> i;
distance.setInches(i);
```

By overloading the stream insertion operator (<<), you could send the `distance` object to `cout`, as shown in the following code, and have the screen output automatically formatted in the correct way.

```
cout << distance;
```

Likewise, by overloading the stream extraction operator (>>), the `distance` object could take values directly from `cin`, as shown here.

```
cin >> distance;
```

Overloading these operators is done in a slightly different way, however, than overloading other operators. These operators are actually part of the `ostream` and `istream` classes defined in the C++ runtime library. (The `cout` and `cin` objects are instances of `ostream` and `istream`.) You must write operator functions to overload the `ostream` version of << and the `istream` version of >>, so they work directly with a class such as `FeetInches`. The `FeetInches` class in the Student Source Code Folder `Chapter 14\FeetInches Version 4` contains functions to overload the << and >> operators. Here is the function that overloads the << operator:

```
ostream &operator << (ostream &strm, const FeetInches &obj)
{
   strm << obj.feet << " feet, " << obj.inches << " inches";
   return strm;
}
```

Notice the function has two parameters: an `ostream` reference object and a `const FeetInches` reference object. The `ostream` parameter will be a reference to the actual `ostream` object on the left side of the << operator. The second parameter is a reference to a `FeetInches` object. This parameter will reference the object on the right side of the << operator. This function tells C++ how to handle any expression that has the following form:

*ostreamObject* << *FeetInchesObject*

So, when C++ encounters the following statement, it will call the overloaded `operator<<` function:

```
cout << distance;
```

Notice that the function's return type is `ostream &`. This means that the function returns a reference to an `ostream` object. When the `return strm;` statement executes, it doesn't return a copy of `strm`, but a reference to it. This allows you to chain together several expressions using the overloaded << operator, such as:

```
cout << distance1 << " " << distance2 << endl;
```

Here is the function that overloads the stream extraction operator to work with the `FeetInches` class:

```
istream &operator >> (istream &strm, FeetInches &obj)
{
   // Prompt the user for the feet.
   cout << "Feet: ";
   strm >> obj.feet;

   // Prompt the user for the inches.
   cout << "Inches: ";
   strm >> obj.inches;

   // Normalize the values.
   obj.simplify();

   return strm;
}
```

The same principles hold true for this operator. It tells C++ how to handle any expression in the following form:

*istreamObject* >> *FeetInchesObject*

Once again, the function returns a reference to an `istream` object, so several of these expressions may be chained together.

You have probably realized that neither of these functions is quite ready to work, though. Both functions attempt to directly access the `FeetInches` object's private members. Because the functions aren't themselves members of the `FeetInches` class, they don't have this type of access. The next step is to make the operator functions friends of `FeetInches`. This is

shown in the following listing of the `FeetInches` class declaration. (This file is stored in the Student Source Code Folder `Chapter 14\FeetInches Version 4`.)

**NOTE:** Some compilers require you to prototype the >> and << operator functions outside the class. For this reason, we have added the following statements to the `FeetInches.h` class specification file.

```
class FeetInches;                          // Forward Declaration

// Function Prototypes for Overloaded Stream Operators
ostream &operator << (ostream &, const FeetInches &);
istream &operator >> (istream &, FeetInches &);
```

### Contents of `FeetInches.h` (Version 4)

```
 1  #ifndef FEETINCHES_H
 2  #define FEETINCHES_H
 3
 4  #include <iostream>
 5  using namespace std;
 6
 7  class FeetInches; // Forward Declaration
 8
 9  // Function Prototypes for Overloaded Stream Operators
10  ostream &operator << (ostream &, const FeetInches &);
11  istream &operator >> (istream &, FeetInches &);
12
13  // The FeetInches class holds distances or measurements
14  // expressed in feet and inches.
15
16  class FeetInches
17  {
18  private:
19       int feet;          // To hold a number of feet
20       int inches;        // To hold a number of inches
21       void simplify();   // Defined in FeetInches.cpp
22  public:
23       // Constructor
24       FeetInches(int f = 0, int i = 0)
25            { feet = f;
26              inches = i;
27              simplify(); }
28
29       // Mutator functions
30       void setFeet(int f)
31            { feet = f; }
32
33       void setInches(int i)
34            { inches = i;
35              simplify(); }
36
37       // Accessor functions
```

```
38      int getFeet() const
39         { return feet; }
40
41      int getInches() const
42          { return inches; }
43
44      // Overloaded operator functions
45      FeetInches operator + (const FeetInches &); // Overloaded +
46      FeetInches operator - (const FeetInches &); // Overloaded -
47      FeetInches operator ++ ();              // Prefix ++
48      FeetInches operator ++ (int);           // Postfix ++
49      bool operator > (const FeetInches &);   // Overloaded >
50      bool operator < (const FeetInches &);   // Overloaded <
51      bool operator == (const FeetInches &);  // Overloaded ==
52
53      // Friends
54      friend ostream &operator << (ostream &, const FeetInches &);
55      friend istream &operator >> (istream &, FeetInches &);
56  };
57
58  #endif
```

Lines 54 and 55 in the class declaration tell C++ to make the overloaded << and >> operator functions friends of the `FeetInches` class:

```
friend ostream &operator<<(ostream &, const FeetInches &);
friend istream &operator>>(istream &, FeetInches &);
```

These statements give the operator functions direct access to the `FeetInches` class's private members. Program 14-11 demonstrates how the overloaded operators work. (This file is also stored in the Student Source Code Folder `Chapter 14\FeetInches Version 4`.)

**Program 14-11**

```
1  // This program demonstrates the << and >> operators,
2  // overloaded to work with the FeetInches class.
3  #include <iostream>
4  #include "FeetInches.h"
5  using namespace std;
6
7  int main()
8  {
9      FeetInches first, second; // Define two objects.
10
11     // Get a distance for the first object.
12     cout << "Enter a distance in feet and inches.\n";
13     cin >> first;
14
15     // Get a distance for the second object.
16     cout << "Enter another distance in feet and inches.\n";
17     cin >> second;
18
19     // Display the values in the objects.
20     cout << "The values you entered are:\n";
```

*(program continues)*

**Program 14-11** *(continued)*

```
21       cout << first << " and " << second << endl;
22       return 0;
23   }
```

**Program Output with Example Input Shown in Bold**

```
Enter a distance in feet and inches.
Feet: 6 [Enter]
Inches: 5 [Enter]
Enter another distance in feet and inches.
Feet: 3 [Enter]
Inches: 10 [Enter]
The values you entered are:
6 feet, 5 inches and 3 feet, 10 inches
```

## Overloading the [ ] Operator

In addition to the traditional operators, C++ allows you to change the way the `[]` symbols work. This gives you the ability to write classes that have array-like behaviors. For example, the `string` class overloads the `[]` operator so you can access the individual characters stored in `string` class objects. Assume the following definition exists in a program:

```
string name = "William";
```

The first character in the string, 'W,' is stored at `name[0]`, so the following statement will display W on the screen.

```
cout << name[0];
```

You can use the overloaded `[]` operator to create an array class, like the following one. The class behaves like a regular array, but performs the bounds-checking that C++ lacks.

### Contents of `IntArray.h`

```
 1  // Specification file for the IntArray class
 2  #ifndef INTARRAY_H
 3  #define INTARRAY_H
 4
 5  class IntArray
 6  {
 7  private:
 8       int *aptr;                      // Pointer to the array
 9       int arraySize;                  // Holds the array size
10       void subscriptError();          // Handles invalid subscripts
11  public:
12       IntArray(int);                  // Constructor
13       IntArray(const IntArray &);     // Copy constructor
14       ~IntArray();                    // Destructor
15
16       int size() const                // Returns the array size
17            { return arraySize;}
18
19       int &operator[](const int &);   // Overloaded [] operator
20  };
21  #endif
```

**Contents of `IntArray.cpp`**

```
1  // Implementation file for the IntArray class
2  #include <iostream>
3  #include <cstdlib>   // For the exit function
4  #include "IntArray.h"
5  using namespace std;
6
7  //*********************************************************
8  // Constructor for IntArray class. Sets the size of the *
9  // array and allocates memory for it.                    *
10 //*********************************************************
11
12 IntArray::IntArray(int s)
13 {
14     arraySize = s;
15     aptr = new int [s];
16     for (int count = 0; count < arraySize; count++)
17         *(aptr + count) = 0;
18 }
19
20 //*********************************************************
21 // Copy Constructor for IntArray class.                  *
22 //*********************************************************
23
24 IntArray::IntArray(const IntArray &obj)
25 {
26     arraySize = obj.arraySize;
27     aptr = new int [arraySize];
28     for(int count = 0; count < arraySize; count++)
29         *(aptr + count) = *(obj.aptr + count);
30 }
31
32 //*********************************************************
33 // Destructor for IntArray class.                        *
34 //*********************************************************
35
36 IntArray::~IntArray()
37 {
38     if (arraySize > 0)
39         delete [] aptr;
40 }
41
42 //*************************************************************
43 // subscriptError function. Displays an error message and   *
44 // terminates the program when a subscript is out of range. *
45 //*************************************************************
46
47 void IntArray::subscriptError()
48 {
49     cout << "ERROR: Subscript out of range.\n";
50     exit(0);
51 }
52
```

```
53  //*******************************************************
54  // Overloaded [] operator. The argument is a subscript.*
55  // This function returns a reference to the element    *
56  // in the array indexed by the subscript.              *
57  //*******************************************************
58
59  int &IntArray::operator[](const int &sub)
60  {
61       if (sub < 0 || sub >= arraySize)
62            subscriptError();
63       return aptr[sub];
64  }
```

Before focusing on the overloaded operator, let's look at the constructors and the destructor. The code for the first constructor in lines 12 through 18 of the `IntArray.cpp` file follows:

```
IntArray::IntArray(int s)
{
   arraySize = s;
   aptr = new int [s];
   for (int count = 0; count < arraySize; count++)
      *(aptr + count) = 0;
}
```

When an instance of the class is defined, the number of elements the array is to have is passed into the constructor's parameter, `s`. This value is copied to the `arraySize` member, and then used to dynamically allocate enough memory for the array. The constructor's final step is to store zeros in all of the array's elements:

```
for (int count = 0; count < arraySize; count++)
   *(aptr + count) = 0;
```

The class also has a copy constructor in lines 24 through 30, which is used when a class object is initialized with another object's data:

```
IntArray::IntArray(const IntArray &obj)
{
   arraySize = obj.arraySize;
   aptr = new int [arraySize];
   for(int count = 0; count < arraySize; count++)
      *(aptr + count) = *(obj.aptr + count);
}
```

A reference to the initializing object is passed into the parameter `obj`. Once the memory is successfully allocated for the array, the constructor copies all the values in `obj`'s array into the calling object's array.

The destructor, in lines 36 through 40, simply frees the memory allocated by the class's constructors. First, however, it checks the value in `arraySize` to be sure the array has at least one element:

```
IntArray::~IntArray()
{
   if (arraySize > 0)
      delete [] aptr;
}
```

The [] operator is overloaded similarly to other operators. The definition of the operator[] function appears in lines 59 through 64:

```
int &IntArray::operator[](const int &sub)
{
   if (sub < 0 || sub >= arraySize)
      subscriptError();
   return aptr[sub];
}
```

The operator[] function can have only a single parameter. The one shown uses a constant reference to an integer. This parameter holds the value placed inside the brackets in an expression. For example, if table is an IntArray object, the number 12 will be passed into the sub parameter in the following statement:

```
cout << table[12];
```

Inside the function, the value in the sub parameter is tested by the following if statement:

```
if (sub < 0 || sub >= arraySize)
   subscriptError();
```

This statement determines whether sub is within the range of the array's subscripts. If sub is less than 0 or greater than or equal to arraySize, it's not a valid subscript, so the subscriptError function is called. If sub is within range, the function uses it as an offset into the array and returns a reference to the value stored at that location.

One critically important aspect of the function above is its return type. It's crucial that the function return not simply an integer, but a *reference* to an integer. The reason for this is that expressions such as the following must be possible:

```
table[5] = 27;
```

Remember, the built-in = operator requires the object on its left to be an lvalue. An lvalue must represent a modifiable memory location, such as a variable. The integer return value of a function is not an lvalue. If the operator[] function merely returns an integer, it cannot be used to create expressions placed on the left side of an assignment operator.

A reference to an integer, however, is an lvalue. If the operator[] function returns a reference, it can be used to create expressions like the following:

```
table[7] = 52;
```

In this statement, the operator[] function is called with 7 passed as its argument. Assuming 7 is within range, the function returns a reference to the integer stored at (aptr + 7). In essence, the statement above is equivalent to:

```
*(aptr + 7) = 52;
```

Because the operator[] function returns actual integers stored in the array, it is not necessary for math or relational operators to be overloaded. Even the stream operators << and >> will work just as they are with the IntArray class.

Program 14-12 demonstrates how the class works.

**Program 14-12**

```
// This program demonstrates an overloaded [] operator.
#include <iostream>
#include "IntArray.h"
using namespace std;

int main()
{
     const int SIZE = 10;  // Array size

     // Define an IntArray with 10 elements.
     IntArray table(SIZE);

     // Store values in the array.
     for (int x = 0; x < SIZE; x++)
          table[x] = (x * 2);

     // Display the values in the array.
     for (int x = 0; x < SIZE; x++)
          cout << table[x] << " ";
     cout << endl;

     // Use the standard + operator on array elements.
     for (int x = 0; x < SIZE; x++)
          table[x] = table[x] + 5;

     // Display the values in the array.
     for (int x = 0; x < SIZE; x++)
          cout << table[x] << " ";
     cout << endl;

     // Use the standard ++ operator on array elements.
     for (int x = 0; x < SIZE; x++)
          table[x]++;

     // Display the values in the array.
     for (int x = 0; x < SIZE; x++)
          cout << table[x] << " ";
     cout << endl;

     return 0;
}
```

**Program Output**

```
0 2 4 6 8 10 12 14 16 18
5 7 9 11 13 15 17 19 21 23
6 8 10 12 14 16 18 20 22 24
```

Program 14-13 demonstrates the `IntArray` class's bounds-checking capability.

**Program 14-13**

```
1  // This program demonstrates the IntArray class's bounds-checking ability.
2  #include <iostream>
3  #include "IntArray.h"
4  using namespace std;
5
6  int main()
7  {
8      const int SIZE = 10;  // Array size
9
10     // Define an IntArray with 10 elements.
11     IntArray table(SIZE);
12
13     // Store values in the array.
14     for (int x = 0; x < SIZE; x++)
15         table[x] = x;
16
17     // Display the values in the array.
18     for (int x = 0; x < SIZE; x++)
19         cout << table[x] << " ";
20     cout << endl;
21
22     // Attempt to use an invalid subscript.
23     cout << "Now attempting to use an invalid subscript.\n";
24     table[SIZE + 1] = 0;
25     return 0;
26 }
```

**Program Output**

```
0 1 2 3 4 5 6 7 8 9
Now attempting to use an invalid subscript.
ERROR: Subscript out of range.
```

## Checkpoint

14.21 Describe the values that should be returned from functions that overload relational operators.

14.22 What is the advantage of overloading the << and >> operators?

14.23 What type of object should an overloaded << operator function return?

14.24 What type of object should an overloaded >> operator function return?

14.25 If an overloaded << or >> operator accesses a private member of a class, what must be done in that class's declaration?

14.26 Assume the class `NumList` has overloaded the `[]` operator. In the expression below, `list1` is an instance of the `NumList` class:

```
list1[25]
```

Rewrite the expression above to explicitly call the function that overloads the `[]` operator.

## 14.6 Object Conversion

**CONCEPT:** **Special operator functions may be written to convert a class object to any other type.**

As you've already seen, operator functions allow classes to work more like built-in data types. Another capability that operator functions can give classes is automatic type conversion.

Data type conversion happens "behind the scenes" with the built-in data types. For instance, suppose a program uses the following variables:

```
int i;
double d;
```

The statement below automatically converts the value in `i` to a floating-point number and stores it in `d`:

```
d = i;
```

Likewise, the following statement converts the value in `d` to an integer (truncating the fractional part) and stores it in `i`:

```
i = d;
```

The same functionality can also be given to class objects. For example, assuming `distance` is a `FeetInches` object and `d` is a `double`, the following statement would conveniently convert `distance`'s value into a floating-point number and store it in `d`, if `FeetInches` is properly written:

```
d = distance;
```

To be able to use a statement such as this, an operator function must be written to perform the conversion. The Student Source Code Folder `Chapter 14\FeetInches Version 5` contains a version of the `FeetInches` class with such an operator function. Here is the code for the operator function that converts a `FeetInches` object to a `double`:

```
FeetInches::operator double()
{
   double temp = feet;

   temp += (inches / 12.0);
   return temp;
}
```

This function contains an algorithm that will calculate the decimal equivalent of a feet and inches measurement. For example, the value 4 feet 6 inches will be converted to 4.5. This value is stored in the local variable `temp`. The `temp` variable is then returned.

**NOTE:** No return type is specified in the function header. Because the function is a `FeetInches-to-double` conversion function, it will always return a `double`. Also, because the function takes no arguments, there are no parameters.

The revised `FeetInches` class also has an operator function that converts a `FeetInches` object to an `int`. The function, shown here, simply returns the `feet` member, thus truncating the `inches` value:

```
FeetInches:: operator int()
{
   return feet;
}
```

Program 14-14 demonstrates both of these conversion functions. (This file is also stored in the Student Source Code Folder `Chapter 14\FeetInches Version 5`.)

**Program 14-14**

```
1  // This program demonstrates the FeetInches class's
2  // conversion functions.
3  #include <iostream>
4  #include "FeetInches.h"
5  using namespace std;
6
7  int main()
8  {
9       double d;  // To hold double input
10      int i;     // To hold int input
11
12      // Define a FeetInches object.
13      FeetInches distance;
14
15      // Get a distance from the user.
16      cout << "Enter a distance in feet and inches:\n";
17      cin >> distance;
18
19      // Convert the distance object to a double.
20      d = distance;
21
22      // Convert the distance object to an int.
23      i = distance;
24
25      // Display the values.
26      cout << "The value " << distance;
27      cout << " is equivalent to " << d << " feet\n";
28      cout << "or " << i << " feet, rounded down.\n";
29      return 0;
30 }
```

**Program Output with Example Input Shown in Bold**

```
Enter a distance in feet and inches:
Feet: 8 [Enter]
Inches: 6 [Enter]
The value 8 feet, 6 inches is equivalent to 8.5 feet
or 8 feet, rounded down.
```

See the Case Study on Creating a String Class for another example. You can download the case study from the book's companion Web site at www.pearsonhighered.com/gaddis.

### Checkpoint

14.27 When overloading a binary operator such as + or –, what object is passed into the operator function's parameter?

14.28 Explain why overloaded prefix and postfix ++ and –– operator functions should return a value.

14.29 How does C++ tell the difference between an overloaded prefix and postfix ++ or –– operator function?

14.30 Write member functions of the `FeetInches` class that overload the prefix and postfix –– operators. Demonstrate the functions in a simple program similar to Program 14-14.

## 14.7 Aggregation

**CONCEPT: Aggregation occurs when a class contains an instance of another class.**

VideoNote
**Class Aggregation**

In real life, objects are frequently made of other objects. A house, for example, is made of door objects, window objects, wall objects, and much more. It is the combination of all these objects that makes a house object.

When designing software, it sometimes makes sense to create an object from other objects. For example, suppose you need an object to represent a course that you are taking in college. You decide to create a `Course` class, which will hold the following information:

- The course name
- The instructor's last name, first name, and office number
- The textbook's title, author, and publisher

In addition to the course name, the class will hold items related to the instructor and the textbook. You could put attributes for each of these items in the `Course` class. However, a good design principle is to separate related items into their own classes. In this example, an `Instructor` class could be created to hold the instructor-related data and a `TextBook` class could be created to hold the textbook-related data. Instances of these classes could then be used as attributes in the `Course` class.

Let's take a closer look at how this might be done. To keep things simple, the `Instructor` class will have only the following functions:

- A default constructor that assigns empty strings to the instructor's last name, first name, and office number.
- A constructor that accepts arguments for the instructor's last name, first name, and office number
- A `set` function that can be used to set all of the class's attributes
- A `print` function that displays the object's attribute values

The code for the `Instructor` class is shown here:

### Contents of `Instructor.h`

```
#ifndef INSTRUCTOR
#define INSTRUCTOR
#include <iostream>
#include <string>
using namespace std;

// Instructor class
class Instructor
{
private:
    string lastName;     // Last name
    string firstName;    // First name
    string officeNumber; // Office number
public:
    // The default constructor stores empty strings
    // in the string objects.
    Instructor()
       { set("", "", ""); }

    // Constructor
    Instructor(string lname, string fname, string office)
       { set(lname, fname, office); }

    // set function
    void set(string lname, string fname, string office)
       { lastName = lname;
         firstName = fname;
         officeNumber = office; }

     // print function
     void print() const
         { cout << "Last name: " << lastName << endl;
           cout << "First name: " << firstName << endl;
           cout << "Office number: " << officeNumber << endl; }
};
#endif
```

The code for the `TextBook` class is shown next. As before, we want to keep the class simple. The only functions it has are a default constructor, a constructor that accepts arguments, a `set` function, and a `print` function.

### Contents of `TextBook.h`

```
#ifndef TEXTBOOK
#define TEXTBOOK
#include <iostream>
#include <string>
using namespace std;

// TextBook class
class TextBook
```

```
{
private:
     string title;     // Book title
     string author;    // Author name
     string publisher; // Publisher name
public:
     // The default constructor stores empty strings
     // in the string objects.
     TextBook()
        { set("", "", ""); }

     // Constructor
     TextBook(string textTitle, string auth, string pub)
        { set(textTitle, auth, pub); }

     // set function
     void set(string textTitle, string auth, string pub)
        { title = textTitle;
          author = auth;
          publisher = pub; }

     // print function
     void print() const
        { cout << "Title: " << title << endl;
          cout << "Author: " << author << endl;
          cout << "Publisher: " << publisher << endl; }
};
#endif
```

The `Course` class is shown next. Notice that the `Course` class has an `Instructor` object and a `TextBook` object as member variables. Those objects are used as attributes of the `Course` object. Making an instance of one class an attribute of another class is called *object aggregation*. The word *aggregate* means "a whole that is made of constituent parts." In this example, the `Course` class is an aggregate class because an instance of it is made of constituent objects.

When an instance of one class is a member of another class, it is said that there is a "has a" relationship between the classes. For example, the relationships that exist among the `Course`, `Instructor`, and `TextBook` classes can be described as follows:

- The course *has an* instructor.
- The course *has a* textbook.

The "has a" relationship is sometimes called a *whole–part relationship* because one object is part of a greater whole.

### Contents of `Course.h`

```
#ifndef COURSE
#define COURSE
#include <iostream>
#include <string>
#include "Instructor.h"
#include "TextBook.h"
using namespace std;

```

```
 9  class Course
10  {
11  private:
12      string courseName;     // Course name
13      Instructor instructor; // Instructor
14      TextBook textbook;     // Textbook
15  public:
16      // Constructor
17      Course(string course, string instrLastName,
18             string instrFirstName, string instrOffice,
19             string textTitle, string author,
20             string publisher)
21          { // Assign the course name.
22            courseName = course;
23
24          // Assign the instructor.
25          instructor.set(instrLastName, instrFirstName, instrOffice);
26
27          // Assign the textbook.
28          textbook.set(textTitle, author, publisher); }
29
30      // print function
31      void print() const
32      { cout << "Course name: " << courseName << endl << endl;
33        cout << "Instructor Information:\n";
34        instructor.print();
35        cout << "\nTextbook Information:\n";
36        textbook.print();
37        cout << endl;}
38  };
39  #endif
```

Program 14-15 demonstrates the `Course` class.

**Program 14-15**

```
1  // This program demonstrates the Course class.
2  #include "Course.h"
3
4  int main()
5  {
6      // Create a Course object.
7      Course myCourse("Intro to Computer Science", // Course name
8           "Kramer", "Shawn", "RH3010",       // Instructor info
9           "Starting Out with C++", "Gaddis", // Textbook title and author
10          "Addison-Wesley");                 // Textbook publisher
11
12     // Display the course info.
13     myCourse.print();
14     return 0;
15 }
```

*(program output continues)*

**Program 14-15** *(continued)*

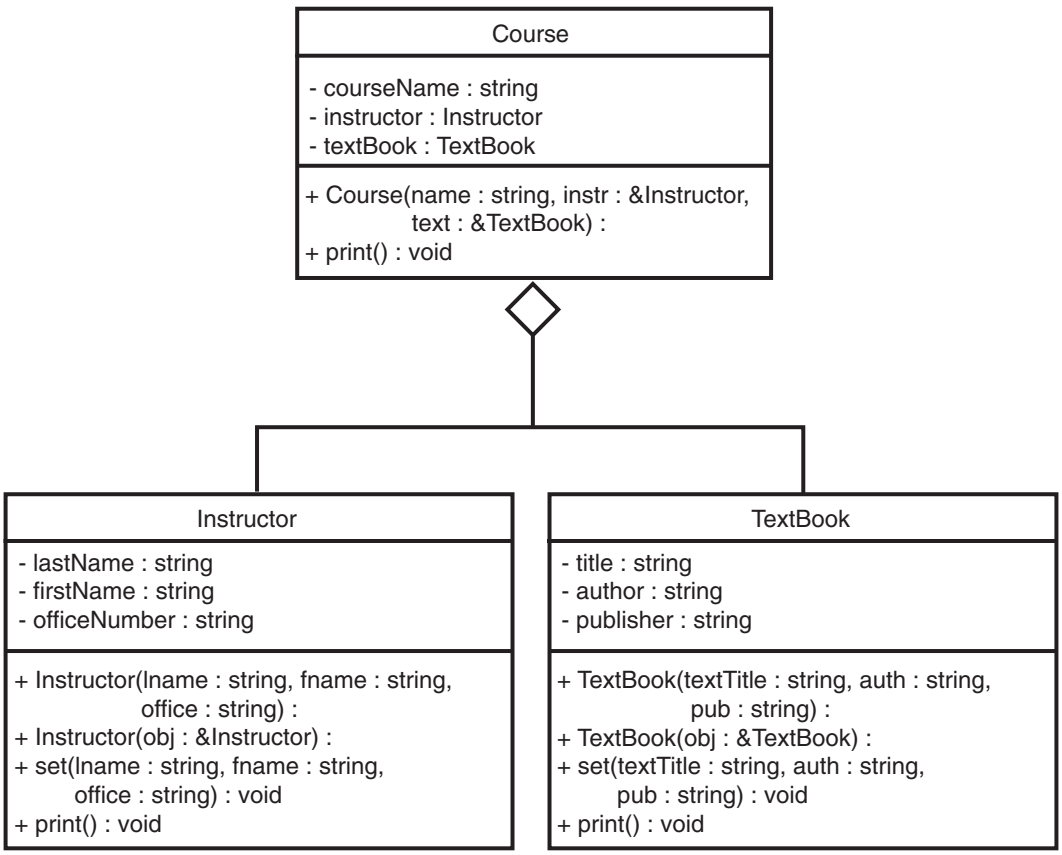
**Program Output**

```
Course name: Intro to Computer Science

Instructor Information:
Last name: Kramer
First name: Shawn
Office number: RH3010

Textbook Information:
Title: Starting Out with C++
Author: Gaddis
Publisher: Addison-Wesley
```

## Aggregation in UML Diagrams

In Chapter 13 you were introduced to the Unified Modeling Language (UML) as a tool for designing classes. You show aggregation in a UML diagram by connecting two classes with a line that has an open diamond at one end. The diamond is closest to the class that is the aggregate. Figure 14-5 shows a UML diagram depicting the relationship between the `Course`, `Instructor`, and `TextBook` classes. The open diamond is closest to the `Course` class because it is the aggregate (the whole).

**Figure 14-5**



## 14.8 Focus on Object-Oriented Design: Class Collaborations

**CONCEPT:** **It is common for classes to interact, or collaborate, with one another to perform their operations. Part of the object-oriented design process is identifying the collaborations between classes.**

In an object-oriented application it is common for objects of different classes to collaborate. This simply means that objects interact with each other. Sometimes one object will need the services of another object in order to fulfill its responsibilities. For example, let's say an object needs to read a number from the keyboard and then format the number to appear as a dollar amount. The object might use the services of the `cin` object to read the number from the keyboard and then use the services of another object that is designed to format the number.

If one object is to collaborate with another object, then it must know something about the other object's member functions and how to call them. Let's look at an example.

The following code shows a class named `Stock`. An object of this class holds data about a company's stock. This class has two attributes: `symbol` and `sharePrice`. The `symbol` attribute holds the trading symbol for the company's stock. This is a short series of characters that are used to identify the stock on the stock exchange. For example, the XYZ Company's stock might have the trading symbol XYZ. The `sharePrice` attribute holds the current price per share of the stock. The class also has the following member functions:

- A default constructor that initializes `symbol` to an empty string and `sharePrice` to 0.0.
- A constructor that accepts arguments for the symbol and share price.
- A copy constructor
- A `set` function that accepts arguments for the symbol and share price.
- A `getSymbol` function that returns the stock's trading symbol.
- A `getSharePrice` function that returns the current price of the stock.

### Contents of `Stock.h`

```
1  #ifndef STOCK
2  #define STOCK
3  #include <string>
4  using namespace std;
5
6  class Stock
7  {
8  private:
9       string symbol;      // Trading symbol of the stock
10      double sharePrice;  // Current price per share
11 public:
12      // Default constructor
13      Stock()
14           { set("", 0.0);}
15
```

```
16      // Constructor
17      Stock(const string sym, double price)
18           { set(sym, price); }
19
20      // Copy constructor
21      Stock(const Stock &obj)
22           { set(obj.symbol, obj.sharePrice); }
23
24      // Mutator function
25      void set(string sym, double price)
26           { symbol = sym;
27             sharePrice = price; }
28
29      // Accessor functions
30      string getSymbol() const
31           { return symbol; }
32
33      double getSharePrice() const
34           { return sharePrice; }
35 };
36 #endif
```

The following code shows another class named `StockPurchase` that uses an object of the `Stock` class to simulate the purchase of a stock. The `StockPurchase` class is responsible for calculating the cost of the stock purchase. To do that, the `StockPurchase` class must know how to call the `Stock` class's `getSharePrice` function to get the price per share of the stock.

### Contents of `StockPurchase.h`

```
1  #ifndef STOCK_PURCHASE
2  #define STOCK_PURCHASE
3  #include "Stock.h"
4
5  class StockPurchase
6  {
7  private:
8       Stock stock;  // The stock that was purchased
9       int shares;   // The number of shares
10 public:
11      // The default constructor sets shares to 0. The stock
12      // object is initialized by its default constructor.
13      StockPurchase()
14           { shares = 0;}
15
16      // Constructor
17      StockPurchase(const Stock &stockObject, int numShares)
18           { stock = stockObject;
19             shares = numShares; }
20
21      // Accessor function
22      double getCost() const
23           { return shares * stock.getSharePrice(); }
24 };
25 #endif
```

The second constructor for the `StockPurchase` class accepts a `Stock` object representing the stock that is being purchased and an `int` representing the number of shares to purchase. In line 18 we see the first collaboration: the `StockPurchase` constructor makes a copy of the `Stock` object by using the `Stock` class's copy constructor. The next collaboration takes place in the `getCost` function. This function calculates and returns the cost of the stock purchase. In line 23 it calls the `Stock` class's `getSharePrice` function to determine the stock's price per share. Program 14-16 demonstrates this class.

**Program 14-16**

```
1  // Stock trader program
2  #include <iostream>
3  #include <iomanip>
4  #include "Stock.h"
5  #include "StockPurchase.h"
6  using namespace std;
7
8  int main()
9  {
10     int sharesToBuy;  // Number of shares to buy
11
12     // Create a Stock object for the company stock. The
13     // trading symbol is XYZ and the stock is currently
14     // priced at $9.62 per share.
15     Stock xyzCompany("XYZ", 9.62);
16
17     // Display the symbol and current share price.
18     cout << setprecision(2) << fixed << showpoint;
19     cout << "XYZ Company's trading symbol is "
20          << xyzCompany.getSymbol() << endl;
21     cout << "The stock is currently $"
22          << xyzCompany.getSharePrice()
23          << " per share.\n";
24
25     // Get the number of shares to purchase.
26     cout << "How many shares do you want to buy? ";
27     cin >> sharesToBuy;
28
29     // Create a StockPurchase object for the transaction.
30     StockPurchase buy(xyzCompany, sharesToBuy);
31
32     // Display the cost of the transaction.
33     cout << "The cost of the transaction is $"
34          << buy.getCost() << endl;
35     return 0;
36 }
```

**Program Output with Example Input Shown in Bold**

```
XYZ Company's trading symbol is XYZ
The stock is currently $9.62 per share.
How many shares do you want to buy? 100 [Enter]
The cost of the transaction is $962.00
```

## Determining Class Collaborations with CRC Cards

During the object-oriented design process, you can determine many of the collaborations that will be necessary between classes by examining the responsibilities of the classes. In Chapter 13 we discussed the process of finding the classes and their responsibilities. Recall from that section that a class's responsibilities are

- the things that the class is responsible for knowing
- the actions that the class is responsible for doing

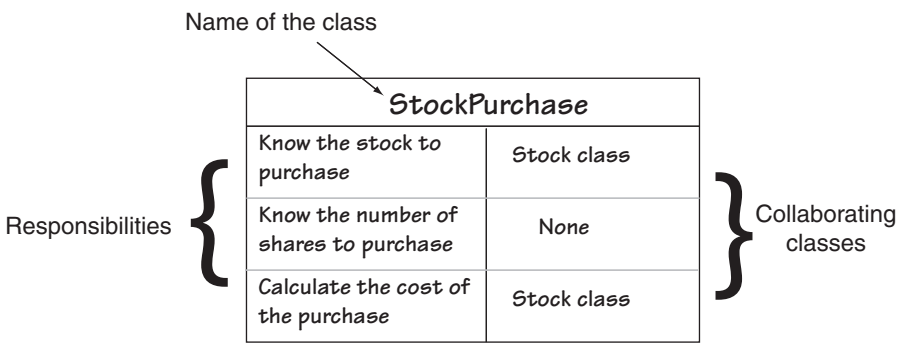
Often you will determine that the class must collaborate with another class in order to fulfill one or more of its responsibilities. One popular method of discovering a class's responsibilities and collaborations is by creating CRC cards. *CRC* stands for class, responsibilities, and collaborations.

You can use simple index cards for this procedure. Once you have gone through the process of finding the classes (which is discussed in Chapter 13), set aside one index card for each class. At the top of the index card, write the name of the class. Divide the rest of the card into two columns. In the left column, write each of the class's responsibilities. As you write each responsibility, think about whether the class needs to collaborate with another class to fulfill that responsibility. Ask yourself questions such as

- Will an object of this class need to get data from another object in order to fulfill this responsibility?
- Will an object of this class need to request another object to perform an operation in order to fulfill this responsibility?

If collaboration is required, write the name of the collaborating class in the right column, next to the responsibility that requires it. If no collaboration is required for a responsibility, simply write "None" in the right column, or leave it blank. Figure 14-6 shows an example CRC card for the `StockPurchase` class.

**Figure 14-6**




From the CRC card shown in the figure, we can see that the `StockPurchase` class has the following responsibilities and collaborations:

- Responsibility: To know the stock to purchase
  Collaboration: The `Stock` class
- Responsibility: To know the number of shares to purchase
  Collaboration: None
- Responsibility: To calculate the cost of the purchase
  Collaboration: The `Stock` class

When you have completed a CRC card for each class in the application, you will have a good idea of each class's responsibilities and how the classes must interact.

### Checkpoint

14.31 What are the benefits of having operator functions that perform object conversion?

14.32 Why are no return types listed in the prototypes or headers of operator functions that perform data type conversion?

14.33 Assume there is a class named `BlackBox`. Write the header for a member function that converts a `BlackBox` object to an `int`.

14.34 Assume there are two classes, `Big` and `Small`. The `Big` class has, as a member, an instance of the `Small` class. Write a sentence that describes the relationship between the two classes.

## 14.9 Focus on Object-Oriented Programming: Simulating the Game of Cho-Han

Cho-Han is a traditional Japanese gambling game in which a dealer uses a cup to roll two six-sided dice. The cup is placed upside down on a table so the value of the dice is concealed. Players then wager on whether the sum of the dice values is even (Cho) or odd (Han). The winner, or winners, take all of the wagers, or the house takes them if there are no winners.

We will develop a program that simulates a simplified variation of the game. The simulated game will have a dealer and two players. The players will not wager money, but will simply guess whether the sum of the dice values is even (Cho) or odd (Han). One point will be awarded to the player, or players, for correctly guessing the outcome. The game is played for five rounds, and the player with the most points is the grand winner.

In the program, we will use the `Die` class that was introduced in Chapter 13 to simulate the dice. We will create two instances of the `Die` class to represent two six-sided dice. In addition to the `Die` class, we will write the following classes:

- `Dealer` class: We will create an instance of this class to represent the dealer. It will have the ability to roll the dice, report the value of the dice, and report whether the total dice value is Cho or Han.
- `Player` class: We will create two instances of this class to represent the players. Instances of the `Player` class can store the player's name, make a guess between Cho or Han, and be awarded points.

First, let's look at the `Dealer` class.

### Contents of `Dealer.h`

```
1  // Specification file for the Dealer class
2  #ifndef DEALER_H
3  #define DEALER_H
4  #include <string>
5  #include "Die.h"
6  using namespace std;
7
```

```
class Dealer
{
private:
     Die die1;              // Object for die #1
     Die die2;              // Object for die #2
     int die1Value;         // Value of die #1
     int die2Value;         // Value of die #2

public:
     Dealer();              // Constructor
     void rollDice();       // To roll the dice
     string getChoOrHan();  // To get the result (Cho or Han)
     int getDie1Value();    // To get the value of die #1
     int getDie2Value();    // To get the value of die #2
};
#endif
```

### Contents of `Dealer.cpp`

```
// Implementation file for the Dealer class
#include "Dealer.h"
#include "Die.h"
#include <string>
using namespace std;

//*****************************************
// Constructor                            *
//*****************************************
Dealer::Dealer()
{
     // Set the intial dice values to 0.
     // (We will not use these values.)
     die1Value = 0;
     die2Value = 0;
}

//*****************************************
// The rollDice member function rolls the  *
// dice and saves their values.            *
//*****************************************
void Dealer::rollDice()
{
     // Roll the dice.
     die1.roll();
     die2.roll();

     // Save the dice values.
     die1Value = die1.getValue();
     die2Value = die2.getValue();
}

```

```
33  //*****************************************
34  // The getChoOrHan member function returns  *
35  // the result of the dice roll, Cho (even) *
36  // or Han (odd).                           *
37  //*****************************************
38  string Dealer::getChoOrHan()
39  {
40      string result; // To hold the result
41
42      // Get the sum of the dice.
43      int sum = die1Value + die2Value;
44
45      // Determine even or odd.
46      if (sum % 2 == 0)
47          result = "Cho (even)";
48      else
49          result = "Han (odd)";
50
51      // Return the result.
52      return result;
53  }
54
55  //*****************************************
56  // The getDie1Value member function returns *
57  // the value of die #1.                    *
58  //*****************************************
59  int Dealer::getDie1Value()
60  {
61      return die1Value;
62  }
63
64  //*****************************************
65  // The getDie2Value member function returns *
66  // the value of die #2.                    *
67  //*****************************************
68  int Dealer::getDie2Value()
69  {
70      return die2Value;
71  }
```

Here is a synopsis of the class members:

| | |
|---|---|
| `die1` | Declared in line 11 of Dealer.h. This is an instance of the `Die` class, to represent die #1. |
| `die2` | Declared in line 12 of Dealer.h. This is an instance of the `Die` class, to represent die #2. |
| `die1Value` | Declared in line 13 of Dealer.h. This member variable will hold the value of die #1 after it has been rolled. |
| `die2Value` | Declared in line 14 of Dealer.h. This member variable will hold the value of die #1 after it has been rolled. |
| Constructor | Lines 10 through 16 of Dealer.cpp initializes the `die1Value` and `die2Value` fields to 0. |

`rollDice` Lines 22 through 31 of Dealer.cpp. This member function simulates the rolling of the dice. Lines 25 and 26 call the `Die` objects' `roll` method. Lines 29 and 30 save the value of the dice in the `die1Value` and `die2Value` member variables.

`getChoOrHan` Lines 38 through 53 of Dealer.cpp. This member function returns a string indicating whether the sum of the dice is Cho (even) or Han (odd).

`getDie1Value` Lines 59 through 62 of Dealer.cpp. This member function returns the value of first die (stored in the `die1Value` member variable).

`getDie2Value` Lines 68 through 71 of Dealer.cpp. This member function returns the value of first die (stored in the `die2Value` member variable).

Now, let's look at the `Player` class.

### Contents of `Player.h`

```
1  // Specification file for the Player class
2  #ifndef PLAYER_H
3  #define PLAYER_H
4  #include <string>
5  using namespace std;
6
7  class Player
8  {
9  private:
10     string name;           // The player's name
11     string guess;          // The player's guess
12     int points;            // The player's points
13
14 public:
15     Player(string);        // Constructor
16     void makeGuess();      // Causes player to make a guess
17     void addPoints(int);   // Adds points to the player
18     string getName();      // Returns the player's name
19     string getGuess();     // Returns the player's guess
20     int getPoints();       // Returns the player's points
21 };
22 #endif
```

### Contents of `Player.cpp`

```
1  // Implementation file for the Player class
2  #include "Player.h"
3  #include <cstdlib>
4  #include <ctime>
5  #include <string>
6  using namespace std;
7
8  //*****************************************
9  // Constructor                            *
10 //*****************************************
11 Player::Player(string playerName)
```

```
12  {
13      // Seed the random number generator.
14      srand(time(0));
15
16      name = playerName;
17      guess = "";
18      points = 0;
19  }
20
21  //*********************************************
22  // The makeGuess member function causes the    *
23  // player to make a guess, either "Cho (even)" *
24  // or "Han (odd)".                             *
25  //*********************************************
26  void Player::makeGuess()
27  {
28      const int MIN_VALUE = 0;
29      const int MAX_VALUE = 1;
30
31      int guessNumber; // For the user's guess
32
33      // Get a random number, either 0 or 1.
34      guessNumber = (rand() % (MAX_VALUE - MIN_VALUE + 1)) + MIN_VALUE;
35
36      // Convert the random number to Cho or Han.
37      if (guessNumber == 0)
38          guess = "Cho (even)";
39      else
40          guess = "Han (odd)";
41  }
42
43  //*********************************************
44  // The addPoints member function adds a       *
45  // specified number of points to the player's *
46  // current balance.                           *
47  //*********************************************
48  void Player::addPoints(int newPoints)
49  {
50      points += newPoints;
51  }
52
53  //*********************************************
54  // The getName member function returns a      *
55  // player's name.                             *
56  //*********************************************
57  string Player::getName()
58  {
59      return name;
60  }
61
62  //*********************************************
63  // The getGuess member function returns a     *
64  // player's guess.                            *
65  //*********************************************
```

```
66  string Player::getGuess()
67  {
68      return guess;
69  }
70
71  //*******************************************
72  // The getPoints member function returns a     *
73  // player's points.                            *
74  //*******************************************
75  int Player::getPoints()
76  {
77      return points;
78  }
```

Here is a synopsis of the class members:

| | |
|---|---|
| `name` | Declared in line 10 of Player.h. This member variable will hold the player's name. |
| `guess` | Declared in line 11 of Player.h. This member variable will hold the player's guess. |
| `points` | Declared in line 12 of Player.h. This member variable will hold the player's points. |
| Constructor | Lines 11 through 19 of Player.cpp, accepts an argument for the player's name, which is assigned to the `name` field. The `guess` field is assigned an empty string, and the `points` field is set to 0. |
| `makeGuess` | Lines 26 through 41 of Player.cpp. This member function causes the player to make a guess. The function generates a random number that is either a 0 or a 1. The `if` statement that begins at line 37 assigns the string `"Cho (even)"` to the `guess` member variable if the random number is 0, or it assigns the string `"Han (odd)"` to the `guess` member variable if the random number is 1. |
| `addPoints` | Lines 48 through 51 of Player.cpp. This member function adds the number of points specified by the argument to the player's `point` member variable. |
| `getName` | Lines 57 through 60 of Player.cpp. This member function returns the player's name. |
| `getGuess` | Lines 66 through 69 of Player.cpp. This member function returns the player's guess. |
| `getPoints` | Lines 75 through 78 of Player.cpp. This member function returns the player's points. |

Program 14-17 uses these classes to simulate the game. The `main` function simulates five rounds of the game, displaying the results of each round, and then displays the overall game results.

**Program 14-17**

```
1  // This program simulates the game of Cho-Han.
2  #include <iostream>
3  #include <string>
4  #include "Dealer.h"
5  #include "Player.h"
6  using namespace std;
7
```

```
// Function prototypes
void roundResults(Dealer &, Player &, Player &);
void checkGuess(Player &, Dealer &);
void displayGrandWinner(Player, Player);

int main()
{
    const int MAX_ROUNDS = 5;  // Number of rounds
    string player1Name;        // First player's name
    string player2Name;        // Second player's name

    // Get the player's names.
    cout << "Enter the first player's name: ";
    cin >> player1Name;
    cout << "Enter the second player's name: ";
    cin >> player2Name;

    // Create the dealer.
    Dealer dealer;

    // Create the two players.
    Player player1(player1Name);
    Player player2(player2Name);

    // Play the rounds.
    for (int round = 0; round < MAX_ROUNDS; round++)
    {
        cout << "----------------------------\n";
        cout << "Now playing round " << (round + 1)
             << endl;

        // Roll the dice.
        dealer.rollDice();

        // The players make their guesses.
        player1.makeGuess();
        player2.makeGuess();

        // Determine the winner of this round.
        roundResults(dealer, player1, player2);
    }

    // Display the grand winner.
    displayGrandWinner(player1, player2);
    return 0;
}

//************************************************
// The roundResults function detremines the results *
// of the current round.                            *
//************************************************
void roundResults(Dealer &dealer, Player &player1, Player &player2)
```

*(program continues)*

**Program 14-17** *(continued)*

```
{
     // Show the dice values.
     cout << "The dealer rolled " << dealer.getDie1Value()
          << " and " << dealer.getDie2Value() << endl;

     // Show the result (Cho or Han).
     cout << "Result: " << dealer.getChoOrHan() << endl;

     // Check each player's guess and award points.
     checkGuess(player1, dealer);
     checkGuess(player2, dealer);
}

//************************************************
// The checkGuess function checks a player's guess *
// against the dealer's result.                    *
//************************************************
void checkGuess(Player &player, Dealer &dealer)
{
     const int POINTS_TO_ADD = 1; // Points to award winner

     // Get the player's guess
     string guess = player.getGuess();

     // Get the result (Cho or Han).
     string choHanResult = dealer.getChoOrHan();

     // Display the player's guess.
     cout << "The player " << player.getName()
          << " guessed " << player.getGuess() << endl;

     // Award points if the player guessed correctly.
     if (guess == choHanResult)
     {
          player.addPoints(POINTS_TO_ADD);
          cout << "Awarding " << POINTS_TO_ADD
               << " point(s) to " << player.getName()
               << endl;
     }
}

//************************************************
// The displayGrandWinner function displays the    *
// game's grand winner.                            *
//************************************************
void displayGrandWinner(Player player1, Player player2)
{
     cout << "---------------------------\n";
     cout << "Game over. Here are the results:\n";

```

```
110     // Display player #1's results.
111     cout << player1.getName() << ": "
112          << player1.getPoints() << " points\n";
113
114     // Display player #2's results.
115     cout << player2.getName() << ": "
116          << player2.getPoints() << " points\n";
117
118     // Determine the grand winner.
119     if (player1.getPoints() > player2.getPoints())
120     {
121         cout << player1.getName()
122              << " is the grand winner!\n";
123     }
124     else if (player2.getPoints() > player1.getPoints())
125     {
126         cout << player2.getName()
127              << " is the grand winner!\n";
128     }
129     else
130     {
131         cout << "Both players are tied!\n";
132     }
133 }
```

**Program Output with Example Input Shown in Bold**

```
Enter the first player's name: Bill [Enter]
Enter the second player's name: Jill [Enter]
---------------------------
Now playing round 1
The dealer rolled 4 and 6
Result: Cho (even)
The player Bill guessed Cho (even)
Awarding 1 point(s) to Bill
The player Jill guessed Cho (even)
Awarding 1 point(s) to Jill
---------------------------
Now playing round 2
The dealer rolled 5 and 2
Result: Han (odd)
The player Bill guessed Han (odd)
Awarding 1 point(s) to Bill
The player Jill guessed Han (odd)
Awarding 1 point(s) to Jill
---------------------------
Now playing round 3
The dealer rolled 4 and 2
Result: Cho (even)
The player Bill guessed Cho (even)
Awarding 1 point(s) to Bill
The player Jill guessed Cho (even)
Awarding 1 point(s) to Jill
---------------------------
```

*(program output continues)*

**Program 14-17** *(continued)*

```
Now playing round 4
The dealer rolled 3 and 2
Result: Han (odd)
The player Bill guessed Han (odd)
Awarding 1 point(s) to Bill
The player Jill guessed Cho (even)
---------------------------
Now playing round 5
The dealer rolled 4 and 6
Result: Cho (even)
The player Bill guessed Han (odd)
The player Jill guessed Han (odd)
---------------------------
Game over. Here are the results:
Bill: 4 points
Jill: 3 points
Bill is the grand winner!
```

Let's look at the code. Here is a summary of the `main` function:

- Lines 15 through 17 make the following declarations: `MAX_ROUNDS`—the number of rounds to play, `player1Name`—to hold the name of player #1, and `player2Name`—to hold the name of player #2.
- Lines 20 through 23 prompt the user to enter the player's names.
- Line 26 creates an instance of the `Dealer` class to represent the dealer.
- Line 29 creates an instance of the `Player` class to represent player #1. Notice that `player1Name` is passed as an argument to the constructor.
- Line 30 creates another instance of the `Player` class to represent player #2. Notice that `player2Name` is passed as an argument to the constructor.
- The `for` loop that begins in line 33 iterates five times, causing the simulation of five rounds of the game. The loop performs the following actions:
  - Line 40 causes the dealer to roll the dice.
  - Line 43 causes player #1 to make a guess (Cho or Han).
  - Line 44 causes player #2 to make a guess (Cho or Han).
  - Line 47 passes the `dealer`, `player1`, and `player2` objects to the `roundResults` function. The function displays the results of this round.
- Line 51 passes the `player1` and `player2` objects to the `displayGrandWinner` function, which displays the grand winner of the game.

The `roundResults` function, which displays the results of a round, appears in lines 59 through 71. Here is a summary of the function:

- The function accepts references to the `dealer`, `player1`, and `player2` objects as arguments.
- The statement in lines 62 through 63 displays the value of the two dice.
- Line 66 calls the `dealer` object's `getChoOrHan` function to display the results, Cho or Han.
- Line 69 calls the `checkGuess` function, passing the `player1` and `dealer` objects as arguments. The `checkGuess` function compares a player's guess to the dealer's result (Cho or Han), and awards points to the player if the guess is correct.
- Line 70 calls the `checkGuess` function, passing the `player2` and `dealer` objects as arguments.

The `checkGuess` function, which compares a player's guess to the dealer's result, awarding points to the player for a correct guess, appears in lines 77 through 99. Here is a summary of the function:

- The function accepts references to a `Player` object and the `Dealer` object as arguments.
- Line 79 declares the constant `POINTS_TO_ADD`, set to the value 1, which is the number of points to add to the player's balance if the player's guess is correct.
- Line 82 assigns the player's guess to the `string` object `guess`.
- Line 85 assigns the dealer's results (Cho or Han) to the `string` object `choHanResult`.
- The statement in lines 88 and 89 displays the player's name and guess.
- The `if` statement in line 92 compares the player's guess to the dealer's result. If they match, then the player guessed correctly, and line 94 awards points to the player.

The `displayGrandWinner` function, which displays the grand winner of the game, appears in lines 105 through 133. Here is a summary of the function:

- The function accepts the `player1` and `player2` objects as arguments.
- The statements in lines 111 through 116 display both players' names and points.
- The `if-else-if` statement that begins in line 119 determines which of the two players has the highest score and displays that player's name as the grand winner. If both players have the same score, a tie is declared.

## Review Questions and Exercises

### Short Answer

1. Describe the difference between an instance member variable and a static member variable.
2. Assume that a class named `Numbers` has the following static member function declaration:

   ```
   static void showTotal();
   ```

   Write a statement that calls the `showTotal` function.
3. A static member variable is declared in a class. Where is the static member variable defined?
4. What is a friend function?
5. Why is it not always a good idea to make an entire class a friend of another class?
6. What is memberwise assignment?
7. When is a copy constructor called?
8. How can the compiler determine if a constructor is a copy constructor?
9. Describe a situation where memberwise assignment is not desirable.
10. Why must the parameter of a copy constructor be a reference?
11. What is a default copy constructor?
12. Why would a programmer want to overload operators rather than use regular member functions to perform similar operations?
13. What is passed to the parameter of a class's `operator=` function?
14. Why shouldn't a class's overloaded = operator be implemented with a `void` operator function?

15. How does the compiler know whether an overloaded ++ operator should be used in prefix or postfix mode?
16. What is the `this` pointer?
17. What type of value should be returned from an overloaded relational operator function?
18. The class `Stuff` has both a copy constructor and an overloaded = operator. Assume that `blob` and `clump` are both instances of the `Stuff` class. For each statement below, indicate whether the copy constructor or the overloaded = operator will be called.

```
Stuff blob = clump;
clump = blob;
blob.operator=(clump);
showValues(blob);     // blob is passed by value.
```

19. Explain the programming steps necessary to make a class's member variable static.
20. Explain the programming steps necessary to make a class's member function static.
21. Consider the following class declaration:

```
class Thing
{
private:
   int x;
   int y;
   static int z;
public:
   Thing()
      { x = y = z; }
   static void putThing(int a)
      { z = a; }
};
```

Assume a program containing the class declaration defines three Thing objects with the following statement:

```
Thing one, two, three;
```

How many separate instances of the `x` member exist?

How many separate instances of the `y` member exist?

How many separate instances of the `z` member exist?

What value will be stored in the `x` and `y` members of each object?

Write a statement that will call the `PutThing` member function *before* the objects above are defined.

22. Describe the difference between making a class a member of another class (object aggregation), and making a class a friend of another class.
23. What is the purpose of a forward declaration of a class?
24. Explain why memberwise assignment can cause problems with a class that contains a pointer member.
25. Why is a class's copy constructor called when an object of that class is passed by value into a function?

## Fill-in-the-Blank

26. If a member variable is declared _____, all objects of that class have access to the same variable.

27. Static member variables are defined _____ the class.
28. A(n) _____ member function cannot access any nonstatic member variables in its own class.
29. A static member function may be called _____ any instances of its class are defined.
30. A(n) _____ function is not a member of a class, but has access to the private members of the class.
31. A(n) _____ tells the compiler that a specific class will be declared later in the program.
32. _____ is the default behavior when an object is assigned the value of another object of the same class.
33. A(n) _____ is a special constructor, called whenever a new object is initialized with another object's data.
34. _____ is a special built-in pointer that is automatically passed as a hidden argument to all nonstatic member functions.
35. An operator may be _____ to work with a specific class.
36. When overloading the _____ operator, its function must have a dummy parameter.
37. Making an instance of one class a member of another class is called _____.
38. Object aggregation is useful for creating a(n) _____ relationship between two classes.

## Algorithm Workbench

39. Assume a class named `Bird` exists. Write the header for a member function that overloads the = operator for that class.
40. Assume a class named `Dollars` exists. Write the headers for member functions that overload the prefix and postfix ++ operators for that class.
41. Assume a class named `Yen` exists. Write the header for a member function that overloads the < operator for that class.
42. Assume a class named `Length` exists. Write the header for a member function that overloads cout's << operator for that class.
43. Assume a class named `Collection` exists. Write the header for a member function that overloads the `[]` operator for that class.

## True or False

44. T F Static member variables cannot be accessed by nonstatic member functions.
45. T F Static member variables are defined outside their class declaration.
46. T F A static member function may refer to nonstatic member variables of the same class, but only after an instance of the class has been defined.
47. T F When a function is declared a `friend` by a class, it becomes a member of that class.
48. T F A `friend` function has access to the private members of the class declaring it a `friend`.
49. T F An entire class may be declared a `friend` of another class.
50. T F In order for a function or class to become a friend of another class, it must be declared as such by the class granting it access.
51. T F If a class has a pointer as a member, it's a good idea to also have a copy constructor.

52. T F You cannot use the = operator to assign one object's values to another object, unless you overload the operator.
53. T F If a class doesn't have a copy constructor, the compiler generates a default copy constructor for it.
54. T F If a class has a copy constructor, and an object of that class is passed by value into a function, the function's parameter will *not* call its copy constructor.
55. T F The `this` pointer is passed to static member functions.
56. T F All functions that overload unary operators must have a dummy parameter.
57. T F For an object to perform automatic type conversion, an operator function must be written.
58. T F It is possible to have an instance of one class as a member of another class.

## Find the Error

Each of the following class declarations has errors. Locate as many as you can.

59.
```
class Box
{
   private:
      double width;
      double length;
      double height;
   public:
      Box(double w, l, h)
         { width = w; length = l; height = h; }
      Box(Box b) // Copy constructor
         { width = b.width;
           length = b.length;
           height = b.height; }

    ... Other member functions follow ...
};
```

60.
```
class Circle
{
   private:
      double diameter;
      int centerX;
      int centerY;
   public:
      Circle(double d, int x, int y)
        { diameter = d; centerX = x; centerY = y; }
      // Overloaded = operator
      void Circle=(Circle &right)
        { diameter = right.diameter;
          centerX = right.centerX;
          centerY = right.centerY; }

   ... Other member functions follow ...
};
```

61.
```
class Point
{
   private:
      int xCoord;
      int yCoord;
   public:
      Point (int x, int y)
         { xCoord = x; yCoord = y; }
      // Overloaded + operator
      void operator+(const &Point right)
         { xCoord += right.xCoord;
           yCoord += right.yCoord;
         }

   ... Other member functions follow ...
};
```

62.
```
class Box
{
   private:
      double width;
      double length;
      double height;
   public:
      Box(double w, l, h)
         { width = w; length = l; height = h; }
      // Overloaded prefix ++ operator
      void operator++()
         { ++width; ++length;}
      // Overloaded postfix ++ operator
      void operator++()
         { width++; length++;}

   ... Other member functions follow ...
};
```

63.
```
class Yard
{
   private:
      float length;
   public:
      yard(float l)
         { length = l; }
      // float conversion function
      void operator float()
         { return length; }

   ... Other member functions follow ...
};
```

## Programming Challenges

1. **`Numbers` Class**

   Design a class `Numbers` that can be used to translate whole dollar amounts in the range 0 through 9999 into an English description of the number. For example, the number 713 would be translated into the string *seven hundred thirteen*, and 8203 would be translated into *eight thousand two hundred three*. The class should have a single integer member variable:

   ```
   int number;
   ```

   and a static array of `string` objects that specify how to translate key dollar amounts into the desired format. For example, you might use static strings such as

   ```
   string lessThan20[20] = {"zero", "one", ..., "eighteen", "nineteen"};
   string hundred = "hundred";
   string thousand = "thousand";
   ```

   The class should have a constructor that accepts a nonnegative integer and uses it to initialize the `Numbers` object. It should have a member function `print()` that prints the English description of the `Numbers` object. Demonstrate the class by writing a main program that asks the user to enter a number in the proper range and then prints out its English description.

2. **Day of the Year**

   Assuming that a year has 365 days, write a class named `DayOfYear` that takes an integer representing a day of the year and translates it to a string consisting of the month followed by day of the month. For example,

   Day 2 would be *January 2*.
   Day 32 would be *February 1*.
   Day 365 would be *December 31*.

   The constructor for the class should take as parameter an integer representing the day of the year, and the class should have a member function `print()` that prints the day in the month–day format. The class should have an integer member variable to represent the day and should have static member variables holding `string` objects that can be used to assist in the translation from the integer format to the month-day format.

   Test your class by inputting various integers representing days and printing out their representation in the month–day format.

3. **Day of the Year Modification**

   Modify the `DayOfYear` class, written in Programming Challenge 2, to add a constructor that takes two parameters: a `string` object representing a month and an integer in the range 0 through 31 representing the day of the month. The constructor should then initialize the integer member of the class to represent the day specified by the month and day of month parameters. The constructor should terminate the program with an appropriate error message if the number entered for a day is outside the range of days for the month given.

Add the following overloaded operators:

`++` **prefix and postfix increment operators.** These operators should modify the `DayOfYear` object so that it represents the next day. If the day is already the end of the year, the new value of the object will represent the first day of the year.

`--` **prefix and postfix decrement operators**. These operators should modify the `DayOfYear` object so that it represents the previous day. If the day is already the first day of the year, the new value of the object will represent the last day of the year.

### 4. `NumDays` Class

VideoNote
**Solving the `NumDays` Problem**

Design a class called `NumDays`. The class's purpose is to store a value that represents a number of work hours and convert it to a number of days. For example, 8 hours would be converted to 1 day, 12 hours would be converted to 1.5 days, and 18 hours would be converted to 2.25 days. The class should have a constructor that accepts a number of hours, as well as member functions for storing and retrieving the hours and days. The class should also have the following overloaded operators:

`+` ***Addition operator.*** When two `NumDays` objects are added together, the overloaded + operator should return the sum of the two objects' hours members.

`-` ***Subtraction operator.*** When one `NumDays` object is subtracted from another, the overloaded – operator should return the difference of the two objects' hours members.

`++` ***Prefix and postfix increment operators.*** These operators should increment the number of hours stored in the object. When incremented, the number of days should be automatically recalculated.

`--` ***Prefix and postfix decrement operators.*** These operators should decrement the number of hours stored in the object. When decremented, the number of days should be automatically recalculated.

### 5. Time Off

**NOTE:** This assignment assumes you have already completed Programming Challenge 4.

Design a class named `TimeOff`. The purpose of the class is to track an employee's sick leave, vacation, and unpaid time off. It should have, as members, the following instances of the `NumDays` class described in Programming Challenge 4:

| | |
|---|---|
| `maxSickDays` | A `NumDays` object that records the maximum number of days of sick leave the employee may take. |
| `sickTaken` | A `NumDays` object that records the number of days of sick leave the employee has already taken. |
| `maxVacation` | A `NumDays` object that records the maximum number of days of paid vacation the employee may take. |
| `vacTaken` | A `NumDays` object that records the number of days of paid vacation the employee has already taken. |

| | |
|---|---|
| `maxUnpaid` | A `NumDays` object that records the maximum number of days of unpaid vacation the employee may take. |
| `unpaidTaken` | A `NumDays` object that records the number of days of unpaid leave the employee has taken. |

Additionally, the class should have members for holding the employee's name and identification number. It should have an appropriate constructor and member functions for storing and retrieving data in any of the member objects.

*Input Validation: Company policy states that an employee may not accumulate more than 240 hours of paid vacation. The class should not allow the* `maxVacation` *object to store a value greater than this amount.*

**6. Personnel Report**

**NOTE:** This assignment assumes you have already completed Programming Challenges 4 and 5.

Write a program that uses an instance of the `TimeOff` class you designed in Programming Challenge 5. The program should ask the user to enter the number of months an employee has worked for the company. It should then use the `TimeOff` object to calculate and display the employee's maximum number of sick leave and vacation days. Employees earn 12 hours of vacation leave and 8 hours of sick leave per month.

**7. `Month` Class**

Design a class named `Month`. The class should have the following private members:

- `name` A `string` object that holds the name of a month, such as "January," "February," etc.
- `monthNumber` An integer variable that holds the number of the month. For example, January would be 1, February would be 2, etc. Valid values for this variable are 1 through 12.

In addition, provide the following member functions:

- A default constructor that sets `monthNumber` to 1 and `name` to "January."
- A constructor that accepts the name of the month as an argument. It should set `name` to the value passed as the argument and set `monthNumber` to the correct value.
- A constructor that accepts the number of the month as an argument. It should set `monthNumber` to the value passed as the argument and set `name` to the correct month name.
- Appropriate set and get functions for the `name` and `monthNumber` member variables.
- Prefix and postfix overloaded `++` operator functions that increment `monthNumber` and set name to the name of next month. If `monthNumber` is set to 12 when these functions execute, they should set `monthNumber` to 1 and `name` to "January."
- Prefix and postfix overloaded `--` operator functions that decrement `monthNumber` and set `name` to the name of previous month. If `monthNumber` is set to 1 when these functions execute, they should set `monthNumber` to 12 and `name` to "December."

Also, you should overload `cout`'s `<<` operator and `cin`'s `>>` operator to work with the `Month` class. Demonstrate the class in a program.

8. **`Date` Class Modification**

   Modify the `Date` class in Programming Challenge 1 of Chapter 13. The new version should have the following overloaded operators:

   `++` ***Prefix and postfix increment operators.*** These operators should increment the object's `day` member.

   `--` ***Prefix and postfix decrement operators.*** These operators should decrement the object's `day` member.

   `-` ***Subtraction operator.*** If one `Date` object is subtracted from another, the operator should give the number of days between the two dates. For example, if April 10, 2014 is subtracted from April 18, 2014, the result will be 8.

   `<<` **`cout`*'s stream insertion operator.*** This operator should cause the date to be displayed in the form

   ```
   April 18, 2014
   ```

   `>>` `cin`***'s stream extraction operator.*** This operator should prompt the user for a date to be stored in a `Date` object.

   The class should detect the following conditions and handle them accordingly:

   - When a date is set to the last day of the month and incremented, it should become the first day of the following month.
   - When a date is set to December 31 and incremented, it should become January 1 of the following year.
   - When a day is set to the first day of the month and decremented, it should become the last day of the previous month.
   - When a date is set to January 1 and decremented, it should become December 31 of the previous year.

   Demonstrate the class's capabilities in a simple program.

   *Input Validation: The overloaded >> operator should not accept invalid dates. For example, the date 13/45/2014 should not be accepted.*

9. **`FeetInches` Modification**

   Modify the `FeetInches` class discussed in this chapter so it overloads the following operators:

   ```
   <=
   >=
   !=
   ```

   Demonstrate the class's capabilities in a simple program.

10. **Corporate Sales**

    A corporation has six divisions, each responsible for sales to different geographic locations. Design a `DivSales` class that keeps sales data for a division, with the following members:

    - An array with four elements for holding four quarters of sales figures for the division.
    - A private static variable for holding the total corporate sales for all divisions for the entire year.

- A member function that takes four arguments, each assumed to be the sales for a quarter. The value of the arguments should be copied into the array that holds the sales data. The total of the four arguments should be added to the static variable that holds the total yearly corporate sales.
- A function that takes an integer argument within the range of 0–3. The argument is to be used as a subscript into the division quarterly sales array. The function should return the value of the array element with that subscript.

Write a program that creates an array of six `DivSales` objects. The program should ask the user to enter the sales for four quarters for each division. After the data are entered, the program should display a table showing the division sales for each quarter. The program should then display the total corporate sales for the year.

*Input Validation: Only accept positive values for quarterly sales figures.*

**11. `FeetInches` Class Copy Constructor and `multiply` Function**

Add a copy constructor to the `FeetInches` class. This constructor should accept a `FeetInches` object as an argument. The constructor should assign to the `feet` attribute the value in the argument's `feet` attribute, and assign to the `inches` attribute the value in the argument's `inches` attribute. As a result, the new object will be a copy of the argument object.

Next, add a `multiply` member function to the `FeetInches` class. The `multiply` function should accept a `FeetInches` object as an argument. The argument object's `feet` and `inches` attributes will be multiplied by the calling object's `feet` and `inches` attributes, and a `FeetInches` object containing the result will be returned.

**12. `LandTract` Class**

Make a `LandTract` class that is composed of two `FeetInches` objects, one for the tract's length and one for the width. The class should have a member function that returns the tract's area. Demonstrate the class in a program that asks the user to enter the dimensions for two tracts of land. The program should display the area of each tract of land and indicate whether the tracts are of equal size.

**13. Carpet Calculator**

The Westfield Carpet Company has asked you to write an application that calculates the price of carpeting for rectangular rooms. To calculate the price, you multiply the area of the floor (width times length) by the price per square foot of carpet. For example, the area of floor that is 12 feet long and 10 feet wide is 120 square feet. To cover that floor with carpet that costs $8 per square foot would cost $960. ($12 \times 10 \times 8 = 960$.)

First, you should create a class named `RoomDimension` that has two `FeetInches` objects as attributes: one for the length of the room and one for the width. (You should use the version of the `FeetInches` class that you created in Programming Challenge 11 with the addition of a `multiply` member function. You can use this function to calculate the area of the room.) The `RoomDimension` class should have a member function that returns the area of the room as a `FeetInches` object.

Next, you should create a `RoomCarpet` class that has a `RoomDimension` object as an attribute. It should also have an attribute for the cost of the carpet per square foot. The `RoomCarpet` class should have a member function that returns the total cost of the carpet.

Once you have written these classes, use them in an application that asks the user to enter the dimensions of a room and the price per square foot of the desired carpeting. The application should display the total cost of the carpet.

**14. Parking Ticket Simulator**

For this assignment you will design a set of classes that work together to simulate a police officer issuing a parking ticket. The classes you should design are:

- **The `ParkedCar` Class:** This class should simulate a parked car. The class's responsibilities are:
  - To know the car's make, model, color, license number, and the number of minutes that the car has been parked
- **The `ParkingMeter` Class:** This class should simulate a parking meter. The class's only responsibility is:
  - To know the number of minutes of parking time that has been purchased
- **The `ParkingTicket` Class:** This class should simulate a parking ticket. The class's responsibilities are:
  - To report the make, model, color, and license number of the illegally parked car
  - To report the amount of the fine, which is $25 for the first hour or part of an hour that the car is illegally parked, plus $10 for every additional hour or part of an hour that the car is illegally parked
  - To report the name and badge number of the police officer issuing the ticket
- **The `PoliceOfficer` Class:** This class should simulate a police officer inspecting parked cars. The class's responsibilities are:
  - To know the police officer's name and badge number
  - To examine a `ParkedCar` object and a `ParkingMeter` object, and determine whether the car's time has expired
  - To issue a parking ticket (generate a `ParkingTicket` object) if the car's time has expired

Write a program that demonstrates how these classes collaborate.

**15. Car Instrument Simulator**

For this assignment you will design a set of classes that work together to simulate a car's fuel gauge and odometer. The classes you will design are:

- The `FuelGauge` Class: This class will simulate a fuel gauge. Its responsibilities are
  - To know the car's current amount of fuel, in gallons.
  - To report the car's current amount of fuel, in gallons.
  - To be able to increment the amount of fuel by 1 gallon. This simulates putting fuel in the car. (The car can hold a maximum of 15 gallons.)
  - To be able to decrement the amount of fuel by 1 gallon, if the amount of fuel is greater than 0 gallons. This simulates burning fuel as the car runs.
- The `Odometer` Class: This class will simulate the car's odometer. Its responsibilities are:
  - To know the car's current mileage.
  - To report the car's current mileage.

- To be able to increment the current mileage by 1 mile. The maximum mileage the odometer can store is 999,999 miles. When this amount is exceeded, the odometer resets the current mileage to 0.
- To be able to work with a `FuelGauge` object. It should decrease the `FuelGauge` object's current amount of fuel by 1 gallon for every 24 miles traveled. (The car's fuel economy is 24 miles per gallon.)

Demonstrate the classes by creating instances of each. Simulate filling the car up with fuel, and then run a loop that increments the odometer until the car runs out of fuel. During each loop iteration, print the car's current mileage and amount of fuel.

CHAPTER 15

# Inheritance, Polymorphism, and Virtual Functions

**TOPICS**



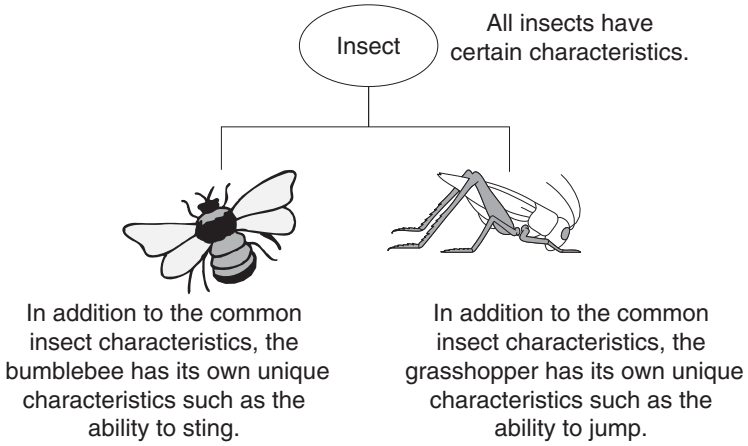
## 15.1 What Is Inheritance?

**CONCEPT:** **Inheritance allows a new class to be based on an existing class. The new class inherits all the member variables and functions (except the constructors and destructor) of the class it is based on.**

### Generalization and Specialization

In the real world you can find many objects that are specialized versions of other more general objects. For example, the term "insect" describes a very general type of creature with numerous characteristics. Because grasshoppers and bumblebees are insects, they have all the general characteristics of an insect. In addition, they have special characteristics of their own. For example, the grasshopper has its jumping ability, and the bumblebee has its stinger. Grasshoppers and bumblebees are specialized versions of an insect. This is illustrated in Figure 15-1.

**Figure 15-1**




## Inheritance and the "Is a" Relationship

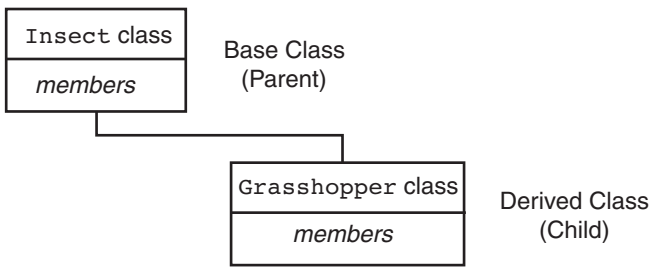
When one object is a specialized version of another object, there is an *"is a" relationship* between them. For example, a grasshopper *is an* insect. Here are a few other examples of the "is a" relationship.

- A poodle *is a* dog.
- A car *is a* vehicle.
- A tree *is a* plant.
- A rectangle *is a* shape.
- A football player *is an* athlete.

When an "is a" relationship exists between classes, it means that the specialized class has all of the characteristics of the general class, plus additional characteristics that make it special. In object-oriented programming, *inheritance* is used to create an "is a" relationship between classes.

Inheritance involves a base class and a derived class. The *base class* is the general class and the *derived class* is the specialized class. The derived class is based on, or derived from, the base class. You can think of the base class as the parent and the derived class as the child. This is illustrated in Figure 15-2.

**Figure 15-2**



The derived class inherits the member variables and member functions of the base class without any of them being rewritten. Furthermore, new member variables and functions may be added to the derived class to make it more specialized than the base class.

Let's look at an example of how inheritance can be used. Most teachers assign various graded activities for their students to complete. A graded activity can receive a numeric score such as 70, 85, 90, and so on, and a letter grade such as A, B, C, D, or F. The following `GradedActivity` class is designed to hold the numeric score and letter grade of a graded activity. When a numeric score is stored by the class, it automatically determines the letter grade. (These files are stored in the Student Source Code Folder `Chapter 15\GradedActivity Version 1`.)

### Contents of `GradedActivity.h` (Version 1)

```
#ifndef GRADEDACTIVITY_H
#define GRADEDACTIVITY_H

// GradedActivity class declaration

class GradedActivity
{
private:
   double score; // To hold the numeric score
public:
   // Default constructor
   GradedActivity()
      { score = 0.0; }

   // Constructor
   GradedActivity(double s)
      { score = s; }

   // Mutator function
   void setScore(double s)
      { score = s; }

   // Accessor functions
   double getScore() const
      { return score; }

   char getLetterGrade() const;
};
#endif
```

### Contents of `GradedActivity.cpp` (Version 1)

```
#include "GradedActivity.h"

//*****************************************************
// Member function GradedActivity::getLetterGrade     *
//*****************************************************

```

```
 7  char GradedActivity::getLetterGrade() const
 8  {
 9     char letterGrade; // To hold the letter grade
10
11     if (score > 89)
12        letterGrade = 'A';
13     else if (score > 79)
14        letterGrade = 'B';
15     else if (score > 69)
16        letterGrade = 'C';
17     else if (score > 59)
18        letterGrade = 'D';
19     else
20        letterGrade = 'F';
21
22     return letterGrade;
23  }
```

The `GradedActivity` class has a default constructor that initializes the `score` member variable to 0.0. A second constructor accepts an argument for `score`. The `setScore` member function also accepts an argument for the `score` variable, and the `getLetterGrade` member function returns the letter grade that corresponds to the value in `score`. Program 15-1 demonstrates the `GradedActivity` class. (This file is also stored in the Student Source Code Folder `Chapter 15\GradedActivity Version 1`.)

**Program 15-1**

```
 1  // This program demonstrates the GradedActivity class.
 2  #include <iostream>
 3  #include "GradedActivity.h"
 4  using namespace std;
 5
 6  int main()
 7  {
 8     double testScore;  // To hold a test score
 9
10     // Create a GradedActivity object for the test.
11     GradedActivity test;
12
13     // Get a numeric test score from the user.
14     cout << "Enter your numeric test score: ";
15     cin >> testScore;
16
17     // Store the numeric score in the test object.
18     test.setScore(testScore);
19
20     // Display the letter grade for the test.
21     cout << "The grade for that test is "
22          << test.getLetterGrade() << endl;
23
24     return 0;
25  }
```

**Program Output with Example Input Shown in Bold**

```
Enter your numeric test score: 89 [Enter]
The grade for that test is B
```

**Program Output with Different Example Input Shown in Bold**

```
Enter your numeric test score: 75 [Enter]
The grade for that test is C
```

The `GradedActivity` class represents the general characteristics of a student's graded activity. Many different types of graded activities exist, however, such as quizzes, midterm exams, final exams, lab reports, essays, and so on. Because the numeric scores might be determined differently for each of these graded activities, we can create derived classes to handle each one. For example, the following code shows the `FinalExam` class, which is derived from the `GradedActivity` class. It has member variables for the number of questions on the exam (`numQuestions`), the number of points each question is worth (`pointsEach`), and the number of questions missed by the student (`numMissed`). These files are also stored in the Student Source Code Folder `Chapter 15\GradedActivity Version 1`.

### Contents of `FinalExam.h`

```
1  #ifndef FINALEXAM_H
2  #define FINALEXAM_H
3  #include "GradedActivity.h"
4
5  class FinalExam : public GradedActivity
6  {
7  private:
8     int numQuestions;    // Number of questions
9     double pointsEach;   // Points for each question
10    int numMissed;       // Number of questions missed
11 public:
12    // Default constructor
13    FinalExam()
14       { numQuestions = 0;
15         pointsEach = 0.0;
16         numMissed = 0; }
17
18    // Constructor
19    FinalExam(int questions, int missed)
20       { set(questions, missed); }
21
22    // Mutator function
23    void set(int, int);  // Defined in FinalExam.cpp
24
25    // Accessor functions
26    double getNumQuestions() const
27       { return numQuestions; }
28
29    double getPointsEach() const
30       { return pointsEach; }
31
```

```
32      int getNumMissed() const
33         { return numMissed; }
34  };
35  #endif
```

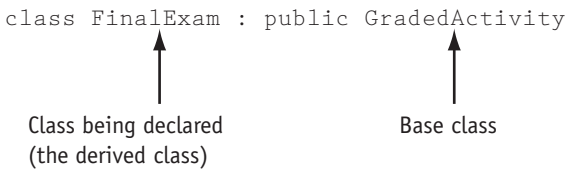
### Contents of `FinalExam.cpp`

```
 1  #include "FinalExam.h"
 2
 3  //*******************************************************
 4  // set function                                         *
 5  // The parameters are the number of questions and the   *
 6  // number of questions missed.                          *
 7  //*******************************************************
 8
 9  void FinalExam::set(int questions, int missed)
10  {
11     double numericScore;  // To hold the numeric score
12
13     // Set the number of questions and number missed.
14     numQuestions = questions;
15     numMissed = missed;
16
17     // Calculate the points for each question.
18     pointsEach = 100.0 / numQuestions;
19
20     // Calculate the numeric score for this exam.
21     numericScore = 100.0 - (missed * pointsEach);
22
23     // Call the inherited setScore function to set
24     // the numeric score.
25     setScore(numericScore);
26  }
```

The only new notation in this code is in line 5 of the `FinalExam.h` file, which reads

```
class FinalExam : public GradedActivity
```

This line indicates the name of the class being declared and the name of the base class it is derived from. `FinalExam` is the name of the class being declared, and `GradedActivity` is the name of the base class it inherits from.




If we want to express the relationship between the two classes, we can say that a `FinalExam` *is a* `GradedActivity`.

The word `public`, which precedes the name of the base class in line 5 of the `FinalExam.h` file, is the *base class access specification*. It affects how the members of the base class are inherited by the derived class. When you create an object of a derived class, you can think

of it as being built on top of an object of the base class. The members of the base class object become members of the derived class object. How the base class members appear in the derived class is determined by the base class access specification.

Although we will discuss this topic in more detail in the next section, let's see how it works in this example. The `GradedActivity` class has both private members and public members. The `FinalExam` class is derived from the `GradedActivity` class, using `public` access specification. This means that the public members of the `GradedActivity` class will become public members of the `FinalExam` class. The private members of the `GradedActivity` class cannot be accessed directly by code in the `FinalExam` class. Although the private members of the `GradedActivity` class are inherited, it's as though they are invisible to the code in the `FinalExam` class. They can only be accessed by the member functions of the `GradedActivity` class. Here is a list of the members of the `FinalExam` class:

Private Members:

| | |
|---|---|
| `int numQuestions` | Declared in the `FinalExam` class |
| `double pointsEach` | Declared in the `FinalExam` class |
| `int numMissed` | Declared in the `FinalExam` class |

Public Members:

| | |
|---|---|
| `FinalExam()` | Defined in the `FinalExam` class |
| `FinalExam(int, int)` | Defined in the `FinalExam` class |
| `set(int, int)` | Defined in the `FinalExam` class |
| `getNumQuestions()` | Defined in the `FinalExam` class |
| `getPointsEach()` | Defined in the `FinalExam` class |
| `getNumMissed()` | Defined in the `FinalExam` class |
| `setScore(double)` | Inherited from `GradedActivity` |
| `getScore()` | Inherited from `GradedActivity` |
| `getLetterGrade()` | Inherited from `GradedActivity` |

The `GradedActivity` class has one private member, the variable `score`. Notice that it is not listed as a member of the `FinalExam` class. It is still inherited by the derived class, but because it is a private member of the base class, only member functions of the base class may access it. It is truly private to the base class. Because the functions `setScore`, `getScore`, and `getLetterGrade` are public members of the base class, they also become public members of the derived class.

You will also notice that the `GradedActivity` class constructors are not listed among the members of the `FinalExam` class. Although the base class constructors still exist, it makes sense that they are not members of the derived class because their purpose is to construct objects of the base class. In the next section we discuss in more detail how base class constructors operate.

Let's take a closer look at the `FinalExam` class constructors. The default constructor appears in lines 13 through 16 of the `FinalExam.h` file. It simply assigns 0 to each of the class's member variables. Another constructor appears in lines 19 through 20. This constructor accepts two arguments, one for the number of questions on the exam, and one for the number of questions missed. This constructor merely passes those values as arguments to the `set` function.

The `set` function is defined in `FinalExam.cpp`. It accepts two arguments: the number of questions on the exam, and the number of questions missed by the student. In lines 14 and 15 these values are assigned to the `numQuestions` and `numMissed` member variables. In line 18 the number of points for each question is calculated. In line 21 the numeric test score is calculated. In line 25, the last statement in the function reads:

```
setScore(numericScore);
```

This is a call to the `setScore` function. Although no `setScore` function appears in the `FinalExam` class, it is inherited from the `GradedActivity` class. Program 15-2 demonstrates the `FinalExam` class.

**Program 15-2**

```
1  // This program demonstrates a base class and a derived class.
2  #include <iostream>
3  #include <iomanip>
4  #include "FinalExam.h"
5  using namespace std;
6
7  int main()
8  {
9     int questions; // Number of questions on the exam
10    int missed;    // Number of questions missed by the student
11
12    // Get the number of questions on the final exam.
13    cout << "How many questions are on the final exam? ";
14    cin >> questions;
15
16    // Get the number of questions the student missed.
17    cout << "How many questions did the student miss? ";
18    cin >> missed;
19
20    // Define a FinalExam object and initialize it with
21    // the values entered.
22    FinalExam test(questions, missed);
23
24    // Display the test results.
25    cout << setprecision(2);
26    cout << "\nEach question counts " << test.getPointsEach()
27         << " points.\n";
28    cout << "The exam score is " << test.getScore() << endl;
29    cout << "The exam grade is " << test.getLetterGrade() << endl;
30
31    return 0;
32 }
```

**Program Output with Example Input Shown in Bold**

```
How many questions are on the final exam? 20 [Enter]
How many questions did the student miss? 3 [Enter]

Each question counts 5 points.
The exam score is 85
The exam grade is B
```

Notice in lines 28 and 29 that the public member functions of the `GradedActivity` class may be directly called by the `test` object:

```
cout << "The exam score is " << test.getScore() << endl;
cout << "The exam grade is " << test.getLetterGrade() << endl;
```

The `getScore` and `getLetterGrade` member functions are inherited as public members of the `FinalExam` class, so they may be accessed like any other public member.

Inheritance does not work in reverse. It is not possible for a base class to call a member function of a derived class. For example, the following classes will not compile in a program because the `BadBase` constructor attempts to call a function in its derived class:

```
class BadBase
{
   private:
      int x;
   public:
      BadBase() { x = getVal(); }  // Error!
};

class Derived : public BadBase
{
   private:
      int y;
   public:
      Derived(int z) { y = z; }
      int getVal() { return y; }
};
```

## Checkpoint

15.1 Here is the first line of a class declaration. Circle the name of the base class:

```
class Truck : public Vehicle
```

15.2 Circle the name of the derived class in the following declaration line:

```
class Truck : public Vehicle
```

15.3 Suppose a program has the following class declarations:

```
class Shape
{
private:
   double area;
public:
   void setArea(double a)
      { area = a; }

   double getArea()
      { return area; }
};

class Circle : public Shape
{
private:
   double radius;
```

```
public:
   void setRadius(double r)
      { radius = r;
        setArea(3.14 * r * r); }
   double getRadius()
      { return radius; }
};
```

Answer the following questions concerning these classes:

A) When an object of the `Circle` class is created, what are its private members?

B) When an object of the `Circle` class is created, what are its public members?

C) What members of the `Shape` class are not accessible to member functions of the `Circle` class?

## 15.2 Protected Members and Class Access

**CONCEPT:** **Protected members of a base class are like private members, but they may be accessed by derived classes. The base class access specification determines how private, public, and protected base class members are accessed when they are inherited by the derived classes.**

Until now you have used two access specifications within a class: `private` and `public`. C++ provides a third access specification, `protected`. Protected members of a base class are like private members, except they may be accessed by functions in a derived class. To the rest of the program, however, protected members are inaccessible.

The following code shows a modified version of the `GradedActivity` class declaration. The private member of the class has been made protected. This file is stored in the Student Source Code Folder `Chapter 15\GradedActivity Version 2`. The implementation file, `GradedActivity.cpp` has not changed, so it is not shown again in this example.

### Contents of `GradedActivity.h` (Version 2)

```
1  #ifndef GRADEDACTIVITY_H
2  #define GRADEDACTIVITY_H
3
4  // GradedActivity class declaration
5
6  class GradedActivity
7  {
8  protected:
9     double score;  // To hold the numeric score
10 public:
11    // Default constructor
12    GradedActivity()
13       { score = 0.0; }
14
15    // Constructor
16    GradedActivity(double s)
17       { score = s; }
18
```

```
19      // Mutator function
20      void setScore(double s)
21         { score = s; }
22
23      // Accessor functions
24      double getScore() const
25         { return score; }
26
27      char getLetterGrade() const;
28   };
29   #endif
```

Now we will look at a modified version of the `FinalExam` class, which is derived from this version of the `GradedActivity` class. This version of the `FinalExam` class has a new member function named `adjustScore`. This function directly accesses the `GradedActivity` class's `score` member variable. If the content of the `score` variable has a fractional part of 0.5 or greater, the function rounds `score` up to the next whole number. The `set` function calls the `adjustScore` function after it calculates the numeric score. (These files are stored in the Student Source Code Folder `Chapter 15\GradedActivity Version 2`.)

### Contents of `FinalExam.h` (Version 2)

```
1   #ifndef FINALEXAM_H
2   #define FINALEXAM_H
3   #include "GradedActivity.h"
4
5   class FinalExam : public GradedActivity
6   {
7   private:
8      int numQuestions;   // Number of questions
9      double pointsEach;  // Points for each question
10     int numMissed;      // Number of questions missed
11  public:
12     // Default constructor
13     FinalExam()
14        { numQuestions = 0;
15          pointsEach = 0.0;
16          numMissed = 0; }
17
18     // Constructor
19     FinalExam(int questions, int missed)
20        { set(questions, missed); }
21
22     // Mutator functions
23     void set(int, int); // Defined in FinalExam.cpp
24     void adjustScore(); // Defined in FinalExam.cpp
25
26     // Accessor functions
27     double getNumQuestions() const
28        { return numQuestions; }
29
30     double getPointsEach() const
31        { return pointsEach; }
32
```

```
33      int getNumMissed() const
34         { return numMissed; }
35  };
36  #endif
```

### Contents of `FinalExam.cpp` (Version 2)

```
 1  #include "FinalExam.h"
 2
 3  //********************************************************
 4  // set function                                          *
 5  // The parameters are the number of questions and the    *
 6  // number of questions missed.                           *
 7  //********************************************************
 8
 9  void FinalExam::set(int questions, int missed)
10  {
11     double numericScore;  // To hold the numeric score
12
13     // Set the number of questions and number missed.
14     numQuestions = questions;
15     numMissed = missed;
16
17     // Calculate the points for each question.
18     pointsEach = 100.0 / numQuestions;
19
20     // Calculate the numeric score for this exam.
21     numericScore = 100.0 - (missed * pointsEach);
22
23     // Call the inherited setScore function to set
24     // the numeric score.
25     setScore(numericScore);
26
27     // Call the adjustScore function to adjust
28     // the score.
29     adjustScore();
30  }
31
32  //************************************************************
33  // Definition of Test::adjustScore. If score is within        *
34  // 0.5 points of the next whole point, it rounds the score up *
35  // and recalculates the letter grade.                         *
36  //************************************************************
37
38  void FinalExam::adjustScore()
39  {
40     double fraction = score - static_cast<int>(score);
41
42     if (fraction >= 0.5)
43     {
44        // Adjust the score variable in the GradedActivity class.
45        score += (1.0 - fraction);
46     }
47  }
```

Program 15-3 demonstrates these versions of the `GradedActivity` and `FinalExam` classes. (This file is also stored in the Student Source Code Folder `Chapter 15\GradedActivity Version 2`.)

**Program 15-3**

```
// This program demonstrates a base class with a
// protected member.
#include <iostream>
#include <iomanip>
#include "FinalExam.h"
using namespace std;

int main()
{
   int questions; // Number of questions on the exam
   int missed;    // Number of questions missed by the student

   // Get the number of questions on the final exam.
   cout << "How many questions are on the final exam? ";
   cin >> questions;

   // Get the number of questions the student missed.
   cout << "How many questions did the student miss? ";
   cin >> missed;

   // Define a FinalExam object and initialize it with
   // the values entered.
   FinalExam test(questions, missed);

   // Display the adjusted test results.
   cout << setprecision(2) << fixed;
   cout << "\nEach question counts "
        << test.getPointsEach() << " points.\n";
   cout << "The adjusted exam score is "
        << test.getScore() << endl;
   cout << "The exam grade is "
        << test.getLetterGrade() << endl;

   return 0;
}
```

**Program Output with Example Input Shown in Bold**

```
How many questions are on the final exam? 16 [Enter]
How many questions did the student miss? 5 [Enter]

Each question counts 6.25 points.
The adjusted exam score is 69.00
The exam grade is D
```

The program works as planned. In the example run, the student missed five questions, which are worth 6.25 points each. The unadjusted score would be 68.75. The score was adjusted to 69.

## More About Base Class Access Specification

The first line of the `FinalExam` class declaration reads:

```
class FinalExam : public GradedActivity
```

This declaration gives public access specification to the base class. Recall from our earlier discussion that base class access specification affects how inherited base class members are accessed. Be careful not to confuse base class access specification with member access specification. Member access specification determines how members that are *defined* within the class are accessed. Base class access specification determines how *inherited* members are accessed.

When you create an object of a derived class, it inherits the members of the base class. The derived class can have its own private, protected, and public members, but what is the access specification of the inherited members? This is determined by the base class access specification. Table 15-1 summarizes how base class access specification affects the way that base class members are inherited.

**Table 15-1**

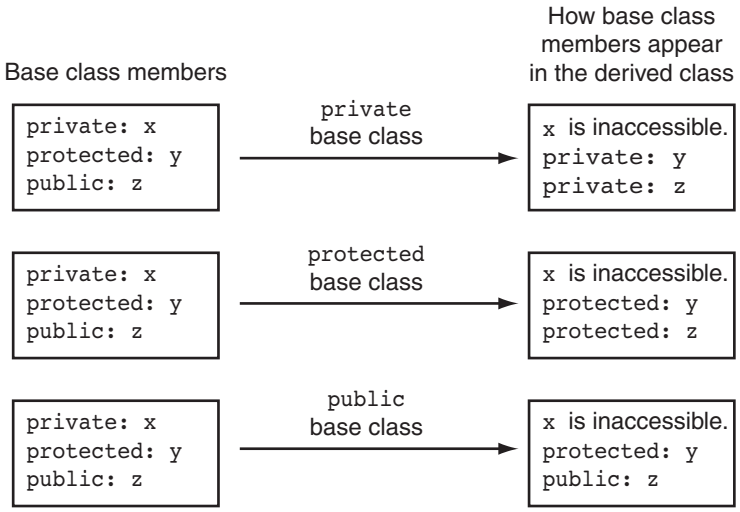
| Base Class Access Specification | How Members of the Base Class Appear in the Derived Class |
|---|---|
| `private` | Private members of the base class are inaccessible to the derived class. |
| | Protected members of the base class become private members of the derived class. |
| | Public members of the base class become private members of the derived class. |
| `protected` | Private members of the base class are inaccessible to the derived class. |
| | Protected members of the base class become protected members of the derived class. |
| | Public members of the base class become protected members of the derived class. |
| `public` | Private members of the base class are inaccessible to the derived class. |
| | Protected members of the base class become protected members of the derived class. |
| | Public members of the base class become public members of the derived class. |

As you can see from Table 15-1, class access specification gives you a great deal of flexibility in determining how base class members will appear in the derived class. Think of a base class's access specification as a filter that base class members must pass through when becoming inherited members of a derived class. This is illustrated in Figure 15-3.

**NOTE:** If the base class access specification is left out of a declaration, the default access specification is `private`. For example, in the following declaration, `Grade` is declared as a `private` base class:

```
class Test : Grade
```

**Figure 15-3**




## Checkpoint

15.4 What is the difference between private members and protected members?

15.5 What is the difference between member access specification and class access specification?

15.6 Suppose a program has the following class declaration:

```
// Declaration of CheckPoint class.
class CheckPoint
{
   private:
      int a;
   protected:
      int b;
      int c;
      void setA(int x) { a = x;}
   public:
      void setB(int y) { b = y;}
      void setC(int z) { c = z;}
};
```

Answer the following questions regarding the class:

A) Suppose another class, `Quiz`, is derived from the `CheckPoint` class. Here is the first line of its declaration:

```
class Quiz : private CheckPoint
```

Indicate whether each member of the `CheckPoint` class is `private`, `protected`, `public`, or inaccessible:

```
a
b
c
setA
setB
setC
```

B) Suppose the `Quiz` class, derived from the `CheckPoint` class, is declared as

```
class Quiz : protected Checkpoint
```

Indicate whether each member of the `CheckPoint` class is `private`, `protected`, `public`, or inaccessible:

```
a
b
c
setA
setB
setC
```

C) Suppose the `Quiz` class, derived from the `CheckPoint` class, is declared as

```
class Quiz : public Checkpoint
```

Indicate whether each member of the `CheckPoint` class is `private`, `protected`, `public`, or inaccessible:

```
a
b
c
setA
setB
setC
```

D) Suppose the `Quiz` class, derived from the `CheckPoint` class, is declared as

```
class Quiz : Checkpoint
```

Is the `CheckPoint` class a `private`, `public`, or `protected` base class?

## 15.3 Constructors and Destructors in Base and Derived Classes

**CONCEPT:** **The base class's constructor is called before the derived class's constructor. The destructors are called in reverse order, with the derived class's destructor being called first.**

In inheritance, the base class constructor is called before the derived class constructor. Destructors are called in reverse order. Program 15-4 shows a simple set of demonstration classes, each with a default constructor and a destructor. The `DerivedClass` class is derived from the `BaseClass` class. Messages are displayed by the constructors and destructors to demonstrate when each is called.

**Program 15-4**

```
1  // This program demonstrates the order in which base and
2  // derived class constructors and destructors are called.
3  #include <iostream>
4  using namespace std;
5
6  //*********************************
7  // BaseClass declaration         *
8  //*********************************
9
```

```
10  class BaseClass
11  {
12  public:
13     BaseClass()  // Constructor
14        { cout << "This is the BaseClass constructor.\n"; }
15
16     ~BaseClass()  // Destructor
17        { cout << "This is the BaseClass destructor.\n"; }
18  };
19
20  //*******************************
21  // DerivedClass declaration     *
22  //*******************************
23
24  class DerivedClass : public BaseClass
25  {
26  public:
27     DerivedClass()  // Constructor
28        { cout << "This is the DerivedClass constructor.\n"; }
29
30     ~DerivedClass()  // Destructor
31        { cout << "This is the DerivedClass destructor.\n"; }
32  };
33
34  //*******************************
35  // main function                *
36  //*******************************
37
38  int main()
39  {
40     cout << "We will now define a DerivedClass object.\n";
41
42     DerivedClass object;
43
44     cout << "The program is now going to end.\n";
45     return 0;
46  }
```

**Program Output**

```
We will now define a DerivedClass object.
This is the BaseClass constructor.
This is the DerivedClass constructor.
The program is now going to end.
This is the DerivedClass destructor.
This is the BaseClass destructor.
```

## Passing Arguments to Base Class Constructors

In Program 15-4, both the base class and derived class have default constructors, that are called automatically. But what if the base class's constructor takes arguments? What if there is more than one constructor in the base class? The answer to these questions is to let the derived class constructor pass arguments to the base class constructor. For example, consider the following class:

### Contents of `Rectangle.h`

```
1  #ifndef RECTANGLE_H
2  #define RECTANGLE_H
3
4  class Rectangle
5  {
6  private:
7     double width;
8     double length;
9  public:
10    // Default constructor
11    Rectangle()
12       { width = 0.0;
13         length = 0.0; }
14
15    // Constructor #2
16    Rectangle(double w, double len)
17       { width = w;
18         length = len; }
19
20    double getWidth() const
21       { return width; }
22
23    double getLength() const
24       { return length; }
25
26    double getArea() const
27       { return width * length; }
28 };
29 #endif
```

This class is designed to hold data about a rectangle. It specifies two constructors. The default constructor, in lines 11 through 13, simply initializes the `width` and `length` member variables to 0.0. The second constructor, in lines 16 through 18, takes two arguments, which are assigned to the `width` and `length` member variables. Now let's look at a class that is derived from the `Rectangle` class:

### Contents of `Cube.h`

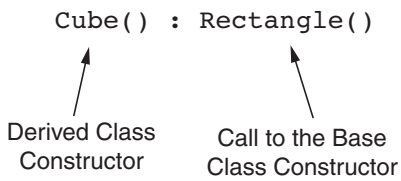
```
1  #ifndef CUBE_H
2  #define CUBE_H
3  #include "Rectangle.h"
4
5  class Cube : public Rectangle
6  {
7  protected:
8     double height;
9     double volume;
10 public:
11    // Default constructor
12    Cube() : Rectangle()
13       { height = 0.0; volume = 0.0; }
14
```

```
15      // Constructor #2
16      Cube(double w, double len, double h) : Rectangle(w, len)
17          { height = h;
18            volume = getArea() * h; }
19
20      double getHeight() const
21          { return height; }
22
23      double getVolume() const
24          { return volume; }
25  };
26  #endif
```

The `Cube` class is designed to hold data about cubes, which not only have a length and width, but a height and volume as well. Look at line 12, which is the first line of the `Cube` class's default constructor:

```
Cube() : Rectangle()
```

Notice the added notation in the header of the constructor. A colon is placed after the derived class constructor's parentheses, followed by a function call to a base class constructor. In this case, the base class's default constructor is being called. When this `Cube` class constructor executes, it will first call the `Rectangle` class's default constructor. This is illustrated here:




The general format of this type of constructor declaration is

```
ClassName::ClassName(ParameterList) : BaseClassName(ArgumentList)
```

You can also pass arguments to the base class constructor, as shown in the `Cube` class's second constructor. Look at line 16:

```
Cube(double w, double len, double h) : Rectangle(w, len)
```

This `Cube` class constructor has three parameters: `w`, `len`, and `h`. Notice that the `Rectangle` class's constructor is called, and the `w` and `len` parameters are passed as arguments. This causes the `Rectangle` class's second constructor to be called.

You only write this notation in the definition of a constructor, not in a prototype. In this example, the derived class constructor is written inline (inside the class declaration), so the notation that contains the call to the base class constructor appears there. If the constructor were defined outside the class, the notation would appear in the function header. For example, the `Cube` class could appear as follows.

```
class Cube : public Rectangle
{
protected:
   double height;
   double volume;
public:
    // Default constructor
    Cube() : Rectangle()
       { height = 0.0; volume = 0.0; }

    // Constructor #2
    Cube(double, double, double);

    double getHeight() const
       { return height; }

    double getVolume() const
       { return volume; }
};

// Cube class constructor #2
Cube::Cube(double w, double len, double h) : Rectangle(w, len)
{
   height = h;
   volume = getArea() * h;
}
```

The base class constructor is always executed before the derived class constructor. When the `Rectangle` constructor finishes, the `Cube` constructor is then executed.

Any literal value or variable that is in scope may be used as an argument to the derived class constructor. Usually, one or more of the arguments passed to the derived class constructor are, in turn, passed to the base class constructor. The values that may be used as base class constructor arguments are

- Derived class constructor parameters
- Literal values
- Global variables that are accessible to the file containing the derived class constructor definition
- Expressions involving any of these items

Program 15-5 shows the `Rectangle` and `Cube` classes in use.

**Program 15-5**

```
1  // This program demonstrates passing arguments to a base
2  // class constructor.
3  #include <iostream>
4  #include "Cube.h"
5  using namespace std;
6
```

```
 7  int main()
 8  {
 9     double cubeWidth;  // To hold the cube's width
10     double cubeLength; // To hold the cube's length
11     double cubeHeight; // To hold the cube's height
12
13     // Get the width, length, and height of
14     // the cube from the user.
15     cout << "Enter the dimensions of a cube:\n";
16     cout << "Width: ";
17     cin >> cubeWidth;
18     cout << "Length: ";
19     cin >> cubeLength;
20     cout << "Height: ";
21     cin >> cubeHeight;
22
23     // Define a Cube object and use the dimensions
24     // entered by the user.
25     Cube myCube(cubeWidth, cubeLength, cubeHeight);
26
27     // Display the Cube object's properties.
28     cout << "Here are the cube's properties:\n";
29     cout << "Width: " << myCube.getWidth() << endl;
30     cout << "Length: " << myCube.getLength() << endl;
31     cout << "Height: " << myCube.getHeight() << endl;
32     cout << "Base area: " << myCube.getArea() << endl;
33     cout << "Volume: " << myCube.getVolume() << endl;
34
35     return 0;
36  }
```

**Program Output with Example Input Shown in Bold**

```
Enter the dimensions of a cube:
Width: 10 [Enter]
Length: 15 [Enter]
Height: 12 [Enter]
Here are the cube's properties:
Width: 10
Length: 15
Height: 12
Base area: 150
Volume: 1800
```

**NOTE:** If the base class has no default constructor, then the derived class must have a constructor that calls one of the base class constructors.

## In the Spotlight:

### The `Automobile`, `Car`, `Truck`, and `SUV` classes

Suppose we are developing a program that a car dealership can use to manage its inventory of used cars. The dealership's inventory includes three types of automobiles: cars, pickup trucks, and sport-utility vehicles (SUVs). Regardless of the type, the dealership keeps the following data about each automobile:

- Make
- Year model
- Mileage
- Price

Each type of vehicle that is kept in inventory has these general characteristics, plus its own specialized characteristics. For cars, the dealership keeps the following additional data:

- Number of doors (2 or 4)

For pickup trucks, the dealership keeps the following additional data:

- Drive type (two-wheel drive or four-wheel drive)

And, for SUVs, the dealership keeps the following additional data:

- Passenger capacity

In designing this program, one approach would be to write the following three classes:

- A `Car` class with attributes for the make, year model, mileage, price, and number of doors.
- A `Truck` class with attributes for the make, year model, mileage, price, and drive type.
- An `SUV` class with attributes for the make, year model, mileage, price, and passenger capacity.

This would be an inefficient approach, however, because all three classes have a large number of common data attributes. As a result, the classes would contain a lot of duplicated code. In addition, if we discover later that we need to add more common attributes, we would have to modify all three classes.

A better approach would be to write an `Automobile` base class to hold all the general data about an automobile, and then write derived classes for each specific type of automobile. The following code shows the `Automobile` class. (This file is stored in the Student Source Code Folder `Chapter 15\Automobile`.)

#### Contents of `Automobile.h`

```
1  #ifndef AUTOMOBILE_H
2  #define AUTOMOBILE_H
3  #include <string>
4  using namespace std;
5
6  // The Automobile class holds general data
7  // about an automobile in inventory.
8  class Automobile
9  {
```

```
10  private:
11     string make;   // The auto's make
12     int model;     // The auto's year model
13     int mileage;   // The auto's mileage
14     double price;  // The auto's price
15
16  public:
17     // Default constructor
18     Automobile()
19     {  make = "";
20        model = 0;
21        mileage = 0;
22        price = 0.0; }
23
24     // Constructor
25     Automobile(string autoMake, int autoModel,
26                int autoMileage, double autoPrice)
27     {  make = autoMake;
28        model = autoModel;
29        mileage = autoMileage;
30        price = autoPrice; }
31
32     // Accessors
33     string getMake() const
34     { return make; }
35
36     int getModel() const
37     { return model; }
38
39     int getMileage() const
40     { return mileage; }
41
42     double getPrice() const
43     { return price; }
44  };
45  #endif
```

Notice that the class has a default constructor in lines 18 through 22, and a constructor that accepts arguments for all of the class's attributes in lines 25 through 30. The `Automobile` class is a complete class that we can create objects from. If we wish, we can write a program that creates instances of the `Automobile` class. However, the `Automobile` class holds only general data about an automobile. It does not hold any of the specific pieces of data that the dealership wants to keep about cars, pickup trucks, and SUVs. To hold data about those specific types of automobiles we will write derived classes that inherit from the `Automobile` class. The following shows the code for the `Car` class. (This file is also stored in the Student Source Code Folder `Chapter 15\Automobile`.)

### Contents of `Car.h`

```
1  #ifndef CAR_H
2  #define CAR_H
3  #include "Automobile.h"
4  #include <string>
5  using namespace std;
```

```
 6
 7  // The Car class represents a car.
 8  class Car : public Automobile
 9  {
10  private:
11     int doors;
12
13  public:
14     // Default constructor
15     Car() : Automobile()
16     { doors = 0;}
17
18     // Constructor #2
19     Car(string carMake, int carModel, int carMileage,
20         double carPrice, int carDoors) :
21         Automobile(carMake, carModel, carMileage, carPrice)
22     { doors = carDoors; }
23
24     // Accessor for doors attribute
25     int getDoors()
26     {return doors;}
27  };
28  #endif
```

The `Car` class defines a `doors` attribute in line 11 to hold the car's number of doors. The class has a default constructor in lines 15 through 16 that sets the `doors` attribute to 0. Notice in line 15 that the default constructor calls the `Automobile` class's default constructor, which initializes all of the inherited attributes to their default values.

The `Car` class also has an overloaded constructor, in lines 19 through 22, that accepts arguments for the car's make, model, mileage, price, and number of doors. Line 21 calls the `Automobile` class's constructor, passing the make, model, mileage, and price as arguments. Line 22 sets the value of the `doors` attribute.

Now let's look at the `Truck` class, which also inherits from the `Automobile` class. (This file is also stored in the Student Source Code Folder `Chapter 15\Automobile`.)

### Contents of `Truck.h`

```
 1  #ifndef TRUCK_H
 2  #define TRUCK_H
 3  #include "Automobile.h"
 4  #include <string>
 5  using namespace std;
 6
 7  // The Truck class represents a truck.
 8  class Truck : public Automobile
 9  {
10  private:
11     string driveType;
12
13  public:
14     // Default constructor
15     Truck() : Automobile()
16     { driveType = ""; }
17
```

```
18      // Constructor #2
19      Truck(string truckMake, int truckModel, int truckMileage,
20         double truckPrice, string truckDriveType) :
21         Automobile(truckMake, truckModel, truckMileage, truckPrice)
22      { driveType = truckDriveType; }
23
24      // Accessor for driveType attribute
25      string getDriveType()
26      { return driveType; }
27  };
28  #endif
```

The `Truck` class defines a `driveType` attribute in line 11 to hold a string describing the truck's drive type. The class has a default constructor in lines 15 through 16 that sets the `driveType` attribute to an empty string. Notice in line 15 that the default constructor calls the `Automobile` class's default constructor, which initializes all of the inherited attributes to their default values.

The `Truck` class also has an overloaded constructor, in lines 19 through 22, that accepts arguments for the truck's make, model, mileage, price, and drive type. Line 21 calls the `Automobile` class's constructor, passing the make, model, mileage, and price as arguments. Line 22 sets the value of the `driveType` attribute.

Now let's look at the `SUV` class, which also inherits from the `Automobile` class. (This file is also stored in the Student Source Code Folder `Chapter 15\Automobile`.)

### Contents of `SUV.h`

```
1   #ifndef SUV_H
2   #define SUV_H
3   #include "Automobile.h"
4   #include <string>
5   using namespace std;
6
7   // The SUV class represents a SUV.
8   class SUV : public Automobile
9   {
10  private:
11      int passengers;
12
13  public:
14      // Default constructor
15      SUV() : Automobile()
16      { passengers = 0; }
17
18      // Constructor #2
19      SUV(string SUVMake, int SUVModel, int SUVMileage,
20          double SUVPrice, int SUVPassengers) :
21          Automobile(SUVMake, SUVModel, SUVMileage, SUVPrice)
22      { passengers = SUVPassengers; }
23
24      // Accessor for passengers attribute
25      int getPassengers()
26      {return passengers;}
27  };
28  #endif
```

The SUV class defines a passengers attribute in line 11 to hold the number of passengers that the vehicle can accommodate. The class has a default constructor in lines 15 through 16 that sets the passengers attribute to 0. Notice in line 15 that the default constructor calls the Automobile class's default constructor, which initializes all of the inherited attributes to their default values.

The SUV class also has an overloaded constructor, in lines 19 through 22, that accepts arguments for the SUV's make, model, mileage, price, and number of passengers. Line 21 calls the Automobile class's constructor, passing the make, model, mileage, and price as arguments. Line 22 sets the value of the passengers attribute.

Program 15-6 demonstrates each of the derived classes. It creates a Car object, a Truck object, and an SUV object. (This file is also stored in the Student Source Code Folder Chapter 15\Automobile.)

**Program 15-6**

```
1  // This program demonstrates the Car, Truck, and SUV
2  // classes that are derived from the Automobile class.
3  #include <iostream>
4  #include <iomanip>
5  #include "Car.h"
6  #include "Truck.h"
7  #include "SUV.h"
8  using namespace std;
9
10 int main()
11 {
12     // Create a Car object for a used 2007 BMW with
13     // 50,000 miles, priced at $15,000, with 4 doors.
14     Car car("BMW", 2007, 50000, 15000.0, 4);
15
16     // Create a Truck object for a used 2006 Toyota
17     // pickup with 40,000 miles, priced at $12,000,
18     // with 4-wheel drive.
19     Truck truck("Toyota", 2006, 40000, 12000.0, "4WD");
20
21     // Create an SUV object for a used 2005 Volvo
22     // with 30,000 miles, priced at $18,000, with
23     // 5 passenger capacity.
24     SUV suv("Volvo", 2005, 30000, 18000.0, 5);
25
26     // Display the automobiles we have in inventory.
27     cout << fixed << showpoint << setprecision(2);
28     cout << "We have the following car in inventory:\n"
29          << car.getModel() << " " << car.getMake()
30          << " with " << car.getDoors() << " doors and "
31          << car.getMileage() << " miles.\nPrice: $"
32          << car.getPrice() << endl << endl;
33
```

```
34      cout << "We have the following truck in inventory:\n"
35           << truck.getModel() << " " << truck.getMake()
36           << " with " << truck.getDriveType()
37           << " drive type and " << truck.getMileage()
38           << " miles.\nPrice: $" << truck.getPrice()
39           << endl << endl;
40
41      cout << "We have the following SUV in inventory:\n"
42           << suv.getModel() << " " << suv.getMake()
43           << " with " << suv.getMileage() << " miles and "
44           << suv.getPassengers() << " passenger capacity.\n"
45           << "Price: $" << suv.getPrice() << endl;
46
47      return 0;
48  }
```

**Program Output**

```
We have the following car in inventory:
2007 BMW with 4 doors and 50000 miles.
Price: $15000.00

We have the following truck in inventory:
2006 Toyota with 4WD drive type and 40000 miles.
Price: $12000.00

We have the following SUV in inventory:
2005 Volvo with 30000 miles and 5 passenger capacity.
Price: $18000.00
```

## Checkpoint

15.7 What will the following program display?

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

class Sky
{
public:
   Sky()
      { cout << "Entering the sky.\n"; }
   ~Sky()
      { cout << "Leaving the sky.\n"; }
};

class Ground : public Sky
{
public:
   Ground()
      { cout << "Entering the Ground.\n"; }
   ~Ground()
      { cout << "Leaving the Ground.\n"; }
};
```

```
int main()
{
   Ground object;
   return 0;
}
```

15.8 What will the following program display?

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

class Sky
{
public:
   Sky()
      { cout << "Entering the sky.\n"; }
   Sky(string color)
      { cout << "The sky is " << color << endl; }
   ~Sky()
      { cout << "Leaving the sky.\n"; }
};

class Ground : public Sky
{
public:
   Ground()
      { cout << "Entering the Ground.\n"; }
   Ground(string c1, string c2) : Sky(c1)
      { cout << "The ground is " << c2 << endl; }
   ~Ground()
      { cout << "Leaving the Ground.\n"; }
};

int main()
{
   Ground object;
   return 0;
}
```

## 15.4 Redefining Base Class Functions

**CONCEPT: A base class member function may be redefined in a derived class.**

VideoNote
**Redefining a Base Class Function in a Derived Class**

Inheritance is commonly used to extend a class or give it additional capabilities. Sometimes it may be helpful to overload a base class function with a function of the same name in the derived class. For example, recall the `GradedActivity` class that was presented earlier in this chapter:

```
class GradedActivity
{
protected:
   char letter;              // To hold the letter grade
   double score;             // To hold the numeric score
   void determineGrade();    // Determines the letter grade
```

```
public:
   // Default constructor
   GradedActivity()
      { letter = ' '; score = 0.0; }

   // Mutator function
   void setScore(double s)
      { score = s;
        determineGrade(); }

   // Accessor functions
   double getScore() const
      { return score; }

   char getLetterGrade() const
      { return letter; }
};
```

This class holds a numeric score and determines a letter grade based on that score. The `setScore` member function stores a value in score, then calls the `determineGrade` member function to determine the letter grade.

Suppose a teacher wants to "curve" a numeric score before the letter grade is determined. For example, Dr. Harrison determines that in order to curve the grades in her class she must multiply each student's score by a certain percentage. This gives an adjusted score, which is used to determine the letter grade.

The following `CurvedActivity` class is derived from the `GradedActivity` class. It multiplies the numeric score by a percentage, and passes that value as an argument to the base class's `setScore` function. (This file is stored in the Student Source Code Folder `Chapter 15\CurvedActivity`.)

### Contents of `CurvedActivity.h`

```
 1  #ifndef CURVEDACTIVITY_H
 2  #define CURVEDACTIVITY_H
 3  #include "GradedActivity.h"
 4
 5  class CurvedActivity : public GradedActivity
 6  {
 7  protected:
 8       double rawScore;     // Unadjusted score
 9       double percentage;   // Curve percentage
10  public:
11       // Default constructor
12       CurvedActivity() : GradedActivity()
13          { rawScore = 0.0; percentage = 0.0; }
14
15       // Mutator functions
16       void setScore(double s)
17          { rawScore = s;
18            GradedActivity::setScore(rawScore * percentage); }
19
```

```
20       void setPercentage(double c)
21          { percentage = c; }
22
23       // Accessor functions
24       double getPercentage() const
25          { return percentage; }
26
27       double getRawScore() const
28          { return rawScore; }
29    };
30    #endif
```

This `CurvedActivity` class has the following member variables:

- `rawScore` This variable holds the student's unadjusted score.
- `percentage` This variable holds the value that the unadjusted score must be multiplied by to get the curved score.

It also has the following member functions:

- A default constructor that calls the `GradedActivity` default constructor, then sets `rawScore` and `percentage` to 0.0.
- `setScore` This function accepts an argument that is the student's unadjusted score. The function stores the argument in the `rawScore` variable, then passes `rawScore * percentage` as an argument to the base class's `setScore` function.
- `setPercentage` This function stores a value in the `percentage` variable.
- `getPercentage` This function returns the value in the `percentage` variable.
- `getRawScore` This function returns the value in the `rawScore` variable.

**NOTE:** Although we are not using the `CurvedActivity` class as a base class, it still has a protected member section. This is because we might want to use the `CurvedActivity` class itself as a base class, as you will see in the next section.

Notice that the `CurvedActivity` class has a `setScore` member function. This function has the same name as one of the base class member functions. When a derived class's member function has the same name as a base class member function, it is said that the derived class function *redefines* the base class function. When an object of the derived class calls the function, it calls the derived class's version of the function.

There is a distinction between redefining a function and overloading a function. An overloaded function is one with the same name as one or more other functions, but with a different parameter list. The compiler uses the arguments passed to the function to tell which version to call. Overloading can take place with regular functions that are not members of a class. Overloading can also take place inside a class when two or more member functions *of the same class* have the same name. These member functions must have different parameter lists for the compiler to tell them apart in function calls.

Redefining happens when a derived class has a function with the same name as a base class function. The parameter lists of the two functions can be the same because the derived class function is always called by objects of the derived class type.

Let's continue our look at the `CurvedActivity` class. Here is the `setScore` member function:

```
void setScore(double s)
   { rawScore = s;
     GradedActivity::setScore(rawScore * percentage); }
```

This function accepts an argument that should be the student's unadjusted numeric score, into the parameter `s`. This value is stored in the `rawScore` variable. Then the following statement is executed:

```
GradedActivity::setScore(rawScore * percentage);
```

This statement calls the base class's version of the `setScore` function with the expression `rawScore * percentage` passed as an argument. Notice that the name of the base class and the scope resolution operator precede the name of the function. This specifies that the base class's version of the `setScore` function is being called. A derived class function may call a base class function of the same name using this notation, which takes this form:

```
BaseClassName::functionName(ArgumentList);
```

Program 15-7 shows the `GradedActivity` and `CurvedActivity` classes used in a complete program. (This file is stored in the Student Source Code Folder `Chapter 15\CurvedActivity`.)

**Program 15-7**

```
1  // This program demonstrates a class that redefines
2  // a base class function.
3  #include <iostream>
4  #include <iomanip>
5  #include "CurvedActivity.h"
6  using namespace std;
7
8  int main()
9  {
10     double numericScore;  // To hold the numeric score
11     double percentage;    // To hold curve percentage
12
13     // Define a CurvedActivity object.
14     CurvedActivity exam;
15
16     // Get the unadjusted score.
17     cout << "Enter the student's raw numeric score: ";
18     cin >> numericScore;
19
20     // Get the curve percentage.
21     cout << "Enter the curve percentage for this student: ";
22     cin >> percentage;
23
24     // Send the values to the exam object.
25     exam.setPercentage(percentage);
26     exam.setScore(numericScore);
27
```

*(program continues)*

**Program 15-7** *(continued)*

```
28       // Display the grade data.
29       cout << fixed << setprecision(2);
30       cout << "The raw score is "
31            << exam.getRawScore() << endl;
32       cout << "The curved score is "
33            << exam.getScore() << endl;
34       cout << "The curved grade is "
35            << exam.getLetterGrade() << endl;
36
37       return 0;
38   }
```

**Program Output with Example Input Shown in Bold**

```
Enter the student's raw numeric score: 87 [Enter]
Enter the curve percentage for this student: 1.06 [Enter]
The raw score is 87.00
The curved score is 92.22
The curved grade is A
```

It is important to note that even though a derived class may redefine a function in the base class, objects that are defined of the base class type still call the base class version of the function. This is demonstrated in Program 15-8.

**Program 15-8**

```
1   // This program demonstrates that when a derived class function
2   // overrides a   class function, objects of the base class
3   // still call the base class version of the function.
4   #include <iostream>
5   using namespace std;
6
7   class BaseClass
8   {
9   public:
10      void showMessage()
11         { cout << "This is the Base class.\n";}
12  };
13
14  class DerivedClass : public BaseClass
15  {
16  public:
17      void showMessage()
18         { cout << "This is the Derived class.\n"; }
19  };
20
21  int main()
22  {
23      BaseClass b;
24      DerivedClass d;
```

```
25
26       b.showMessage();
27       d.showMessage();
28
29       return 0;
30   }
```

**Program Output**

```
This is the Base class.
This is the Derived class.
```

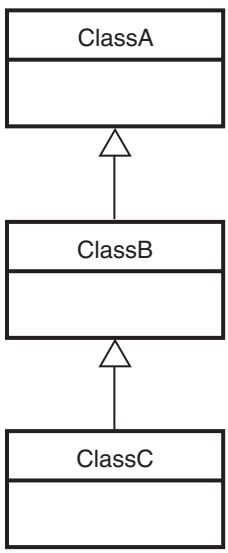
In Program 15-8, a class named `BaseClass` is declared with a member function named `showMessage`. A class named `DerivedClass` is then declared, also with a `showMessage` member function. As their names imply, `DerivedClass` is derived from `BaseClass`. Two objects, `b` and `d`, are defined in function `main`. The object `b` is a `BaseClass` object, and d is a `DerivedClass` object. When `b` is used to call the `showMessage` function, it is the `BaseClass` version that is executed. Likewise, when `d` is used to call `showMessage`, the `DerivedClass` version is used.

## 15.5 Class Hierarchies

**CONCEPT: A base class can also be derived from another class.**

Sometimes it is desirable to establish a hierarchy of classes in which one class inherits from a second class, which in turn inherits from a third class, as illustrated by Figure 15-4. In some cases, the inheritance of classes goes on for many layers.

**Figure 15-4**



In Figure 15-4, `ClassC` inherits `ClassB`'s members, including the ones that `ClassB` inherited from `ClassA`. Let's look at an example of such a chain of inheritance. Consider the following `PassFailActivity` class, which inherits from the `GradedActivity` class. The class is intended to determine a letter grade of 'P' for passing, or 'F' for failing. (This file is stored in the Student Source Code Folder `Chapter 15\PassFailActivity`.)

### Contents of `PassFailActivity.h`

```
1  #ifndef PASSFAILACTIVITY_H
2  #define PASSFAILACTIVITY_H
3  #include "GradedActivity.h"
4
5  class PassFailActivity : public GradedActivity
6  {
7  protected:
8      double minPassingScore;  // Minimum passing score.
9  public:
10     // Default constructor
11     PassFailActivity() : GradedActivity()
12        { minPassingScore = 0.0; }
13
14     // Constructor
15     PassFailActivity(double mps) : GradedActivity()
16          { minPassingScore = mps; }
17
18     // Mutator
19     void setMinPassingScore(double mps)
20          { minPassingScore = mps; }
21
22     // Accessors
23     double getMinPassingScore() const
24          { return minPassingScore; }
25
26     char getLetterGrade() const;
27 };
28 #endif
```

The `PassFailActivity` class has a private member variable named `minPassingScore`. This variable holds the minimum passing score for an activity. The default constructor, in lines 11 through 12, sets `minPassingScore` to 0.0. An overloaded constructor in lines 15 through 16 accepts a `double` argument that is the minimum passing grade for the activity. This value is stored in the `minPassingScore` variable. The `getLetterGrade` member function is defined in the following `PassFailActivity.cpp` file. (This file is also stored in the Student Source Code Folder `Chapter 15\PassFailActivity`.)

### Contents of `PassFailActivity.cpp`

```
1  #include "PassFailActivity.h"
2
3  //*****************************************************
4  // Member function PassFailActivity::getLetterGrade    *
5  // This function returns 'P' if the score is passing, *
6  // otherwise it returns 'F'.                           *
7  //*****************************************************
```

```
 8
 9  char PassFailActivity::getLetterGrade() const
10  {
11      char letterGrade;
12
13      if (score >= minPassingScore)
14          letterGrade = 'P';
15      else
16          letterGrade = 'F';
17
18      return letterGrade;
19  }
```

This `getLetterGrade` member function redefines the `getLetterGrade` member function of `GradedActivity` class. This version of the function returns a grade of `'P'` if the numeric score is greater than or equal to `minPassingScore`. Otherwise, the function returns a grade of `'F'`.

The `PassFailActivity` class represents the general characteristics of a student's pass-or-fail activity. There might be numerous types of pass-or-fail activities, however. Suppose we need a more specialized class, such as one that determines a student's grade for a pass-or-fail exam. The following `PassFailExam` class is an example. This class is derived from the `PassFailActivity` class. It inherits all of the members of `PassFailActivity`, including the ones that `PassFailActivity` inherits from `GradedActivity`. The `PassFailExam` class calculates the number of points that each question on the exam is worth, as well as the student's numeric score. (These files are stored in the Student Source Code Folder `Chapter 15\PassFailActivity`.)

### Contents of `PassFailExam.h`

```
 1  #ifndef PASSFAILEXAM_H
 2  #define PASSFAILEXAM_H
 3  #include "PassFailActivity.h"
 4
 5  class PassFailExam : public PassFailActivity
 6  {
 7  private:
 8      int numQuestions;     // Number of questions
 9      double pointsEach;    // Points for each question
10      int numMissed;        // Number of questions missed
11  public:
12      // Default constructor
13      PassFailExam() : PassFailActivity()
14          { numQuestions = 0;
15            pointsEach = 0.0;
16            numMissed = 0; }
17
18      // Constructor
19      PassFailExam(int questions, int missed, double mps) :
20          PassFailActivity(mps)
21          { set(questions, missed); }
22
```

```
23      // Mutator function
24      void set(int, int);  // Defined in PassFailExam.cpp
25
26      // Accessor functions
27      double getNumQuestions() const
28          { return numQuestions; }
29
30      double getPointsEach() const
31          { return pointsEach; }
32
33      int getNumMissed() const
34          { return numMissed; }
35  };
36  #endif
```

### Contents of `PassFailExam.cpp`

```
1   #include "PassFailExam.h"
2
3   //*********************************************************
4   // set function                                           *
5   // The parameters are the number of questions and the     *
6   // number of questions missed.                            *
7   //*********************************************************
8
9   void PassFailExam::set(int questions, int missed)
10  {
11      double numericScore; // To hold the numeric score
12
13      // Set the number of questions and number missed.
14      numQuestions = questions;
15      numMissed = missed;
16
17      // Calculate the points for each question.
18      pointsEach = 100.0 / numQuestions;
19
20      // Calculate the numeric score for this exam.
21      numericScore = 100.0 - (missed * pointsEach);
22
23      // Call the inherited setScore function to set
24      // the numeric score.
25      setScore(numericScore);
26  }
```

The `PassFailExam` class inherits all of the `PassFailActivity` class's members, including the ones that `PassFailActivity` inherited from `GradedActivity`. Because the `public` base class access specification is used, all of the protected members of `PassFailActivity` become protected members of `PassFailExam`, and all of the `public` members of `PassFailActivity` become `public` members of `PassFailExam`. Table 15-2 lists all of the member variables of the `PassFailExam` class, and Table 15-3 lists all the member functions. These include the members that were inherited from the base classes.

**Table 15-2**

| Member Variable of the `PassFailExam` Class | Access | Inherited? |
|---|---|---|
| `numQuestions` | protected | No |
| `pointsEach` | protected | No |
| `numMissed` | protected | No |
| `minPassingScore` | protected | Yes, from `PassFailActivity` |
| `score` | protected | Yes, from `PassFailActivity`, which inherited it from `GradedActivity` |

**Table 15-3**

| Member Function of the `PassFailExam` Class | Access | Inherited? |
|---|---|---|
| `set` | public | No |
| `getNumQuestions` | public | No |
| `getPointsEach` | public | No |
| `getNumMissed` | public | No |
| `setMinPassingScore` | public | Yes, from `PassFailActivity` |
| `getMinPassingScore` | public | Yes, from `PassFailActivity` |
| `getLetterGrade` | public | Yes, from `PassFailActivity` |
| `setScore` | public | Yes, from `PassFailActivity`, which inherited it from `GradedActivity` |
| `getScore` | public | Yes, from `PassFailActivity`, which inherited it from `GradedActivity` |

Program 15-9 demonstrates the `PassFailExam` class. This file is also stored in the student source code folder `Chapter 15\PassFailActivity`.

**Program 15-9**

```
1  // This program demonstrates the PassFailExam class.
2  #include <iostream>
3  #include <iomanip>
4  #include "PassFailExam.h"
5  using namespace std;
6
7  int main()
8  {
9      int questions;        // Number of questions
10     int missed;           // Number of questions missed
11     double minPassing;    // The minimum passing score
12
13     // Get the number of questions on the exam.
```

*(program continues)*

**Program 15-9** *(continued)*

```
14       cout << "How many questions are on the exam? ";
15       cin >> questions;
16
17       // Get the number of questions the student missed.
18       cout << "How many questions did the student miss? ";
19       cin >> missed;
20
21       // Get the minimum passing score.
22       cout << "Enter the minimum passing score for this test: ";
23       cin >> minPassing;
24
25       // Define a PassFailExam object.
26       PassFailExam exam(questions, missed, minPassing);
27
28       // Display the test results.
29       cout << fixed << setprecision(1);
30       cout << "\nEach question counts "
31            << exam.getPointsEach() << " points.\n";
32       cout << "The minimum passing score is "
33            << exam.getMinPassingScore() << endl;
34       cout << "The student's exam score is "
35            << exam.getScore() << endl;
36       cout << "The student's grade is "
37            << exam.getLetterGrade() << endl;
38       return 0;
39  }
```

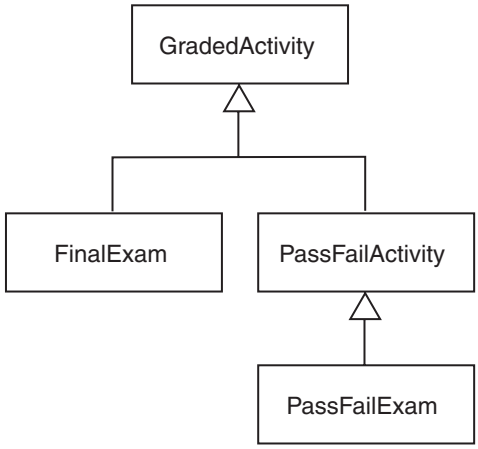
**Program Output with Example Input Shown in Bold**

```
How many questions are on the exam? 100 [Enter]
How many questions did the student miss? 25 [Enter]
Enter the minimum passing score for this test: 60 [Enter]

Each question counts 1.0 points.
The minimum passing score is 60.0
The student's exam score is 75.0
The student's grade is P
```

This program uses the `PassFailExam` object to call the `getLetterGrade` member function in line 37. Recall that the `PassFailActivity` class redefines the `getLetterGrade` function to report only grades of 'P' or 'F'. Because the `PassFailExam` class is derived from the `PassFailActivity` class, it inherits the redefined `getLetterGrade` function.

Software designers often use class hierarchy diagrams. Like a family tree, a class hierarchy diagram shows the inheritance relationships between classes. Figure 15-5 shows a class hierarchy for the `GradedActivity`, `FinalExam`, `PassFailActivity`, and `PassFailExam` classes. The more general classes are toward the top of the tree and the more specialized classes are toward the bottom.

**Figure 15-5**




## 15.6 Polymorphism and Virtual Member Functions

**CONCEPT:** **Polymorphism allows an object reference variable or an object pointer to reference objects of different types and to call the correct member functions, depending upon the type of object being referenced.**

VideoNote
**Polymorphism**

Look at the following code for a function named `displayGrade`:

```
void displayGrade(const GradedActivity &activity)
{
   cout << setprecision(1) << fixed;
   cout << "The activity's numeric score is "
        << activity.getScore() << endl;
   cout << "The activity's letter grade is "
        << activity.getLetterGrade() << endl;
}
```

This function uses a `const GradedActivity` reference variable as its parameter. When a `GradedActivity` object is passed as an argument to this function, the function calls the object's `getScore` and `getLetterGrade` member functions to display the numeric score and letter grade. The following code shows how we might call the function.

```
GradedActivity test(88.0);   // The score is 88
displayGrade(test);          // Pass test to displayGrade
```

This code will produce the following output:

```
The activity's numeric score is 88.0
The activity's letter grade is B
```

Recall that the `GradedActivity` class is also the base class for the `FinalExam` class. Because of the "is-a" relationship between a base class and a derived class, an object of the `FinalExam` class is not just a `FinalExam` object. It is also a `GradedActivity` object.

(A final exam *is a* graded activity.) Because of this relationship, we can also pass a `FinalExam` object to the `displayGrade` function. For example, look at the following code:

```
// There are 100 questions. The student missed 25.
FinalExam test2(100, 25);
displayGrade(test2);
```

This code will produce the following output:

```
The activity's numeric score is 75.0
The activity's letter grade is C
```

Because the parameter in the `displayGrade` function is a `GradedActivity` reference variable, it can reference any object that is derived from `GradedActivity`. A problem can occur with this type of code, however, when redefined member functions are involved. For example, recall that the `PassFailActivity` class is derived from the `GradedActivity` class. The `PassFailActivity` class redefines the `getLetterGrade` function. Although we can pass a `PassFailActivity` object as an argument to the `displayGrade` function, we will not get the results we wish. This is demonstrated in Program 15-10. (This file is stored in the Student Source Code Folder `Chapter 15\PassFailActivity`.)

**Program 15-10**

```
1  #include <iostream>
2  #include <iomanip>
3  #include "PassFailActivity.h"
4  using namespace std;
5
6  // Function prototype
7  void displayGrade(const GradedActivity &);
8
9  int main()
10 {
11     // Create a PassFailActivity object. Minimum passing
12     // score is 70.
13     PassFailActivity test(70);
14
15     // Set the score to 72.
16     test.setScore(72);
17
18     // Display the object's grade data. The letter grade
19     // should be 'P'. What will be displayed?
20     displayGrade(test);
21     return 0;
22 }
23
24 //************************************************************
25 // The displayGrade function displays a GradedActivity object's *
26 // numeric score and letter grade.                             *
27 //************************************************************
28
29 void displayGrade(const GradedActivity &activity)
```

```
30  {
31       cout << setprecision(1) << fixed;
32       cout << "The activity's numeric score is "
33            << activity.getScore() << endl;
34       cout << "The activity's letter grade is "
35            << activity.getLetterGrade() << endl;
36  }
```

**Program Output**

```
The activity's numeric score is 72.0
The activity's letter grade is C
```

As you can see from the example output, the `getLetterGrade` member function returned 'C' instead of 'P'. This is because the `GradedActivity` class's `getLetterGrade` function was executed instead of the `PassFailActivity` class's version of the function.

This behavior happens because of the way C++ matches function calls with the correct function. This process is known as *binding*. In Program 15-10, C++ decides at compile time which version of the `getLetterGrade` function to execute when it encounters the call to the function in line 35. Even though we passed a `PassFailActivity` object to the `displayGrade` function, the `activity` parameter in the `displayGrade` function is a `GradedActivity` reference variable. Because it is of the `GradedActivity` type, the compiler binds the function call in line 35 with the `GradedActivity` class's `getLetterGrade` function. When the program executes, it has already been determined by the compiler that the `GradedActivity` class's `getLetterGrade` function will be called. The process of matching a function call with a function at compile time is called *static binding*.

To remedy this, the `getLetterGrade` function can be made *virtual*. A *virtual function* is a member function that is dynamically bound to function calls. In *dynamic binding*, C++ determines which function to call at runtime, depending on the type of the object responsible for the call. If a `GradedActivity` object is responsible for the call, C++ will execute the `GradedActivity::getLetterGrade` function. If a `PassFailActivity` object is responsible for the call, C++ will execute the `PassFailActivity::getLetterGrade` function.

Virtual functions are declared by placing the key word `virtual` before the return type in the base class's function declaration, such as

```
virtual char getLetterGrade() const;
```

This declaration tells the compiler to expect `getLetterGrade` to be redefined in a derived class. The compiler does not bind calls to the function with the actual function. Instead, it allows the program to bind calls, at runtime, to the version of the function that belongs to the same class as the object responsible for the call.

**NOTE:** You place the `virtual` key word only in the function's declaration or prototype. If the function is defined outside the class, you do not place the `virtual` key word in the function header.

The following code shows an updated version of the `GradedActivity` class, with the `getLetterGrade` function declared `virtual`. This file is stored in the Student Source Code Folder `Chapter 15\GradedActivity Version 3`. The `GradedActivity.cpp` file has not changed, so it is not shown again.

### Contents of `GradedActivity.h` (Version 3)

```
1  #ifndef GRADEDACTIVITY_H
2  #define GRADEDACTIVITY_H
3
4  // GradedActivity class declaration
5
6  class GradedActivity
7  {
8  protected:
9       double score;  // To hold the numeric score
10 public:
11      // Default constructor
12      GradedActivity()
13         { score = 0.0; }
14
15      // Constructor
16      GradedActivity(double s)
17         { score = s; }
18
19      // Mutator function
20      void setScore(double s)
21         { score = s; }
22
23      // Accessor functions
24      double getScore() const
25         { return score; }
26
27      virtual char getLetterGrade() const;
28 };
29 #endif
```

The only change we have made to this class is to declare `getLetterGrade` as `virtual` in line 27. This tells the compiler not to bind calls to `getLetterGrade` with the function at compile time. Instead, calls to the function will be bound dynamically to the function at runtime.

When a member function is declared `virtual` in a base class, any redefined versions of the function that appear in derived classes automatically become `virtual`. So, it is not necessary to declare the `getLetterGrade` function in the `PassFailActivity` class as `virtual`. It is still a good idea to declare the function `virtual` in the `PassFailActivity` class for documentation purposes. A new version of the `PassFailActivity` class is shown here. This file is stored in the Student Source Code Folder `Chapter 15\GradedActivity Version 3`. The `PassFailActivity.cpp` file has not changed, so it is not shown again.

### Contents of `PassFailActivity.h`

```
1  #ifndef PASSFAILACTIVITY_H
2  #define PASSFAILACTIVITY_H
3  #include "GradedActivity.h"
4
5  class PassFailActivity : public GradedActivity
```

```
 6  {
 7  protected:
 8      double minPassingScore;  // Minimum passing score
 9   public:
10      // Default constructor
11      PassFailActivity() : GradedActivity()
12         { minPassingScore = 0.0; }
13
14      // Constructor
15      PassFailActivity(double mps) : GradedActivity()
16         { minPassingScore = mps; }
17
18      // Mutator
19      void setMinPassingScore(double mps)
20         { minPassingScore = mps; }
21
22      // Accessors
23      double getMinPassingScore() const
24         { return minPassingScore; }
25
26      virtual char getLetterGrade() const;
27  };
28  #endif
```

The only change we have made to this class is to declare `getLetterGrade` as `virtual` in line 26. Program 15-11 is identical to Program 15-10, except it uses the corrected version of the `GradedActivity` and `PassFailActivity` classes. This file is also stored in the student source code folder `Chapter 15\GradedActivity Version 3`.

**Program 15-11**

```
 1  #include <iostream>
 2  #include <iomanip>
 3  #include "PassFailActivity.h"
 4  using namespace std;
 5
 6  // Function prototype
 7  void displayGrade(const GradedActivity &);
 8
 9  int main()
10  {
11      // Create a PassFailActivity object. Minimum passing
12      // score is 70.
13      PassFailActivity test(70);
14
15      // Set the score to 72.
16      test.setScore(72);
17
18      // Display the object's grade data. The letter grade
19      // should be 'P'. What will be displayed?
20      displayGrade(test);
21      return 0;
22  }
```

*(program continues)*

**Program 15-11** *(continued)*

```
23
24  //************************************************************
25  // The displayGrade function displays a GradedActivity object's *
26  // numeric score and letter grade.                             *
27  //************************************************************
28
29  void displayGrade(const GradedActivity &activity)
30  {
31       cout << setprecision(1) << fixed;
32       cout << "The activity's numeric score is "
33            << activity.getScore() << endl;
34       cout << "The activity's letter grade is "
35            << activity.getLetterGrade() << endl;
36  }
```

**Program Output**

```
The activity's numeric score is 72.0
The activity's letter grade is P
```

Now that the `getLetterGrade` function is declared `virtual`, the program works properly. This type of behavior is known as polymorphism. The term *polymorphism* means the ability to take many forms. Program 15-12 demonstrates polymorphism by passing objects of the `GradedActivity` and `PassFailExam` classes to the `displayGrade` function. This file is stored in the Student Source Code Folder `Chapter 15\GradedActivity Version 3`.

**Program 15-12**

```
1   #include <iostream>
2   #include <iomanip>
3   #include "PassFailExam.h"
4   using namespace std;
5
6   // Function prototype
7   void displayGrade(const GradedActivity &);
8
9   int main()
10  {
11       // Create a GradedActivity object. The score is 88.
12       GradedActivity test1(88.0);
13
14       // Create a PassFailExam object. There are 100 questions,
15       // the student missed 25 of them, and the minimum passing
16       // score is 70.
17       PassFailExam test2(100, 25, 70.0);
18
19       // Display the grade data for both objects.
20       cout << "Test 1:\n";
21       displayGrade(test1);     // GradedActivity object
22       cout << "\nTest 2:\n";
```

```
23      displayGrade(test2);  // PassFailExam object
24      return 0;
25  }
26
27  //***************************************************************
28  // The displayGrade function displays a GradedActivity object's *
29  // numeric score and letter grade.                               *
30  //***************************************************************
31
32  void displayGrade(const GradedActivity &activity)
33  {
34      cout << setprecision(1) << fixed;
35      cout << "The activity's numeric score is "
36           << activity.getScore() << endl;
37      cout << "The activity's letter grade is "
38           << activity.getLetterGrade() << endl;
39  }
```

**Program Output**

```
Test 1:
The activity's numeric score is 88.0
The activity's letter grade is B

Test 2:
The activity's numeric score is 75.0
The activity's letter grade is P
```

## Polymorphism Requires References or Pointers

The `displayGrade` function in Programs 15-11 and 15-12 uses a `GradedActivity` reference variable as its parameter. When we call the function, we pass an object by reference. Polymorphic behavior is not possible when an object is passed by value, however. For example, suppose the `displayGrade` function had been written as shown here:

```
// Polymorphic behavior is not possible with this function.
void displayGrade(const GradedActivity activity)
{
   cout << setprecision(1) << fixed;
   cout << "The activity's numeric score is "
        << activity.getScore() << endl;
   cout << "The activity's letter grade is "
        << activity.getLetterGrade() << endl;
}
```

In this version of the function the `activity` parameter is an object variable, not a reference variable. Suppose we call this version of the function with the following code:

```
// Create a GradedActivity object. The score is 88.
GradedActivity test1(88.0);

// Create a PassFailExam object. There are 100 questions,
// the student missed 25 of them, and the minimum passing
// score is 70.
PassFailExam test2(100, 25, 70.0);
```

```
// Display the grade data for both objects.
cout << "Test 1:\n";
displayGrade(test1);  // Pass the GradedActivity object
cout << "\nTest 2:\n";
displayGrade(&test2); // Pass the PassFailExam object
```

This code will produce the following output:

```
Test 1:
The activity's numeric score is 88.0
The activity's letter grade is B

Test 2:
The activity's numeric score is 75.0
The activity's letter grade is C
```

Even though the `getLetterGrade` function is declared `virtual`, static binding still takes place because `activity` is not a reference variable or a pointer.

Alternatively we could have used a `GradedActivity` pointer in the `displayGrade` function, as shown in Program 15-13. This file is also stored in the Student Source Code Folder `Chapter 15\GradedActivity Version 3`.

**Program 15-13**

```
1  #include <iostream>
2  #include <iomanip>
3  #include "PassFailExam.h"
4  using namespace std;
5
6  // Function prototype
7  void displayGrade(const GradedActivity *);
8
9  int main()
10 {
11     // Create a GradedActivity object. The score is 88.
12     GradedActivity test1(88.0);
13
14     // Create a PassFailExam object. There are 100 questions,
15     // the student missed 25 of them, and the minimum passing
16     // score is 70.
17     PassFailExam test2(100, 25, 70.0);
18
19     // Display the grade data for both objects.
20     cout << "Test 1:\n";
21     displayGrade(&test1);  // Address of the GradedActivity object
22     cout << "\nTest 2:\n";
23     displayGrade(&test2);  // Address of the PassFailExam object
24     return 0;
25 }
26
27 //***************************************************************
28 // The displayGrade function displays a GradedActivity object's *
29 // numeric score and letter grade. This version of the function *
30 // uses a GradedActivity pointer as its parameter.              *
31 //***************************************************************
```

```
32
33  void displayGrade(const GradedActivity *activity)
34  {
35       cout << setprecision(1) << fixed;
36       cout << "The activity's numeric score is "
37            << activity->getScore() << endl;
38       cout << "The activity's letter grade is "
39            << activity->getLetterGrade() << endl;
40  }
```

**Program Output**

```
Test 1:
The activity's numeric score is 88.0
The activity's letter grade is B

Test 2:
The activity's numeric score is 75.0
The activity's letter grade is P
```

## Base Class Pointers

Pointers to a base class may be assigned the address of a derived class object. For example, look at the following code:

```
GradedActivity *exam = new PassFailExam(100, 25, 70.0);
```

This statement dynamically allocates a `PassFailExam` object and assigns its address to `exam`, which is a `GradedActivity` pointer. We can then use the `exam` pointer to call member functions, as shown here:

```
cout << exam->getScore() << endl;
cout << exam->getLetterGrade() << endl;
```

Program 15-14 is an example that uses base class pointers to reference derived class objects. This file is also stored in the Student Source Code Folder `Chapter 15\ GradedActivity Version 3`.

**Program 15-14**

```
1  #include <iostream>
2  #include <iomanip>
3  #include "PassFailExam.h"
4  using namespace std;
5
6  // Function prototype
7  void displayGrade(const GradedActivity *);
8
9  int main()
10 {
11      // Constant for the size of an array.
12      const int NUM_TESTS = 4;
13
```

*(program continues)*

**Program 15-14** *(continued)*

```
14       // tests is an array of GradedActivity pointers.
15       // Each element of tests is initialized with the
16       // address of a dynamically allocated object.
17       GradedActivity *tests[NUM_TESTS] =
18            { new GradedActivity(88.0),
19              new PassFailExam(100, 25, 70.0),
20              new GradedActivity(67.0),
21              new PassFailExam(50, 12, 60.0)
22            };
23
24       // Display the grade data for each element in the array.
25       for (int count = 0; count < NUM_TESTS; count++)
26       {
27            cout << "Test #" << (count + 1) << ":\n";
28            displayGrade(tests[count]);
29            cout << endl;
30       }
31       return 0;
32  }
33
34  //************************************************************
35  // The displayGrade function displays a GradedActivity object's *
36  // numeric score and letter grade. This version of the function *
37  // uses a GradedActivity pointer as its parameter.             *
38  //************************************************************
39
40  void displayGrade(const GradedActivity *activity)
41  {
42       cout << setprecision(1) << fixed;
43       cout << "The activity's numeric score is "
44            << activity->getScore() << endl;
45       cout << "The activity's letter grade is "
46            << activity->getLetterGrade() << endl;
47  }
```

**Program Output**

```
Test #1:
The activity's numeric score is 88.0
The activity's letter grade is B

Test #2:
The activity's numeric score is 75.0
The activity's letter grade is P

Test #3:
The activity's numeric score is 67.0
The activity's letter grade is D

Test #4:
The activity's numeric score is 76.0
The activity's letter grade is P
```

Let's take a closer look at this program. An array named `tests` is defined in lines 17 through 22. This is an array of `GradedActivity` pointers. The array elements are initialized with the addresses of dynamically allocated objects. The `tests[0]` element is initialized with the address of the `GradedActivity` object returned from this expression:

```
new GradedActivity(88.0)
```

The `tests[1]` element is initialized with the address of the `GradedActivity` object returned from this expression:

```
new PassFailExam(100, 25, 70.0)
```

The `tests[2]` element is initialized with the address of the `GradedActivity` object returned from this expression:

```
new GradedActivity(67.0)
```

Finally, the `tests[3]` element is initialized with the address of the `GradedActivity` object returned from this expression:

```
new PassFailExam(50, 12, 60.0)
```

Although each element in the array is a `GradedActivity` pointer, some of the elements point to `GradedActivity` objects and some point to `PassFailExam` objects. The loop in lines 25 through 30 steps through the array, passing each pointer element to the `displayGrade` function.

## Base Class Pointers and References Know Only About Base Class Members

Although a base class pointer can reference objects of any class that derives from the base class, there are limits to what the pointer can do with those objects. Recall that the `GradedActivity` class has, other than its constructors, only three member functions: `setScore`, `getScore`, and `getLetterGrade`. So, a `GradedActivity` pointer can be used to call only those functions, regardless of the type of object it points to. For example, look at the following code.

```
GradedActivity *exam = new PassFailExam(100, 25, 70.0);
cout << exam->getScore() << endl;         // This works.
cout << exam->getLetterGrade() << endl;   // This works.
cout << exam->getPointsEach() << endl;    // ERROR! Won't work!
```

In this code, `exam` is a `GradedActivity` pointer, and is assigned the address of a `PassFailExam` object. The `GradedActivity` class has only the `setScore`, `getScore`, and `getLetterGrade` member functions, so those are the only member functions that the `exam` variable knows how to execute. The last statement in this code is a call to the `getPointsEach` member function, which is defined in the `PassFailExam` class. Because the `exam` variable only knows about member functions in the `GradedActivity` class, it cannot execute this function.

## The "Is-a" Relationship Does Not Work in Reverse

It is important to note that the "is-a" relationship does not work in reverse. Although the statement "a final exam is a graded activity" is true, the statement "a graded activity is a

final exam" is not true. This is because not all graded activities are final exams. Likewise, not all `GradedActivity` objects are `FinalExam` objects. So, the following code will not work.

```
// Create a GradedActivity object.
GradedActivity *gaPointer = new GradedActivity(88.0);

// Error! This will not work.
FinalExam *fePointer = gaPointer;
```

You cannot assign the address of a `GradedActivity` object to a `FinalExam` pointer. This makes sense because `FinalExam` objects have capabilities that go beyond those of a `GradedActivity` object. Interestingly, the C++ compiler will let you make such an assignment if you use a type cast, as shown here:

```
// Create a GradedActivity object.
GradedActivity *gaPointer = new GradedActivity(88.0);

// This will work, but with limitations.
FinalExam *fePointer = static_cast<FinalExam *>(gaPointer);
```

After this code executes, the derived class pointer `fePointer` will be pointing to a base class object. We can use the pointer to access members of the object, but only the members that exist. The following code demonstrates:

```
// This will work. The object has a getScore function.
cout << fePointer->getScore() << endl;

// This will work. The object has a getLetterGrade function.
cout << fePointer->getLetterGrade() << endl;

// This will compile, but an error will occur at runtime.
// The object does not have a getPointsEach function.
cout << fePointer->getPointsEach() << endl;
```

In this code `fePointer` is a `FinalExam` pointer, and it points to a `GradedActivity` object. The first two `cout` statements work because the `GradedActivity` object has `getScore` and a `getLetterGrade` member functions. The last `cout` statement will cause an error, however, because it calls the `getPointsEach` member function. The `GradedActivity` object does not have a `getPointsEach` member function.

## Redefining vs. Overriding

Earlier in this chapter you learned how a derived class can redefine a base class member function. When a class redefines a virtual function, it is said that the class *overrides* the function. In C++, the difference between overriding and redefining base class functions is that overridden functions are dynamically bound, and redefined functions are statically bound. Only virtual functions can be overridden.

## Virtual Destructors

When you write a class with a destructor, and that class could potentially become a base class, you should always declare the destructor `virtual`. This is because the compiler will perform static binding on the destructor if it is not declared `virtual`. This can lead to problems when a base class pointer or reference variable references a derived class object. If

the derived class has its own destructor, it will not execute when the object is destroyed or goes out of scope. Only the base class destructor will execute. Program 15-15 demonstrates.

**Program 15-15**

```
1  #include <iostream>
2  using namespace std;
3
4  // Animal is a base class.
5  class Animal
6  {
7  public:
8       // Constructor
9       Animal()
10           { cout << "Animal constructor executing.\n"; }
11
12      // Destructor
13      ~Animal()
14           { cout << "Animal destructor executing.\n"; }
15 };
16
17 // The Dog class is derived from Animal
18 class Dog : public Animal
19 {
20 public:
21      // Constructor
22      Dog() : Animal()
23           { cout << "Dog constructor executing.\n"; }
24
25      // Destructor
26      ~Dog()
27           { cout << "Dog destructor executing.\n"; }
28 };
29
30 //************************************************
31 // main function                                 *
32 //************************************************
33
34 int main()
35 {
36      // Create a Dog object, referenced by an
37      // Animal pointer.
38      Animal *myAnimal = new Dog;
39
40      // Delete the dog object.
41      delete myAnimal;
42      return 0;
43 }
```

**Program Output**

```
Animal constructor executing.
Dog constructor executing.
Animal destructor executing.
```

This program declares two classes: `Animal` and `Dog`. `Animal` is the base class and `Dog` is the derived class. Each class has its own constructor and destructor. In line 38, a `Dog` object is created, and its address is stored in an `Animal` pointer. Both the `Animal` and the `Dog` constructors execute. In line 41 the object is deleted. When this statement executes, however, only the `Animal` destructor executes. The `Dog` destructor does not execute because the object is referenced by an `Animal` pointer. We can fix this problem by declaring the `Animal` class destructor `virtual`, as shown in Program 15-16.

**Program 15-16**

```
1  #include <iostream>
2  using namespace std;
3
4  // Animal is a base class.
5  class Animal
6  {
7  public:
8       // Constructor
9       Animal()
10           { cout << "Animal constructor executing.\n"; }
11
12      // Destructor
13      virtual ~Animal()
14           { cout << "Animal destructor executing.\n"; }
15 };
16
17 // The Dog class is derived from Animal
18 class Dog : public Animal
19 {
20 public:
21      // Constructor
22      Dog() : Animal()
23           { cout << "Dog constructor executing.\n"; }
24
25      // Destructor
26      ~Dog()
27           { cout << "Dog destructor executing.\n"; }
28 };
29
30 //************************************************
31 // main function                                 *
32 //************************************************
33
34 int main()
35 {
36      // Create a Dog object, referenced by an
37      // Animal pointer.
38      Animal *myAnimal = new Dog;
39
40      // Delete the dog object.
41      delete myAnimal;
42      return 0;
43 }
```

**Program Output**

```
Animal constructor executing.
Dog constructor executing.
Dog destructor executing.
Animal destructor executing.
```

The only thing that has changed in this program is that the `Animal` class destructor is declared virtual in line 13. As a result, the destructor is dynamically bound at runtime. When the `Dog` object is destroyed, both the `Animal` and `Dog` destructors execute.

A good programming practice to follow is that any class that has a virtual member function should also have a virtual destructor. If the class doesn't require a destructor, it should have a virtual destructor that performs no statements. Remember, when a base class function is declared `virtual`, all overridden versions of the function in derived classes automatically become virtual. Including a virtual destructor in a base class, even one that does nothing, will ensure that any derived class destructors will also be virtual.

## C++ 11's `override` and `final` Key Words

C++ 11 introduces the `override` key word to help prevent subtle errors when overriding virtual functions. For example, can you find the mistake in Program 15-17?

**Program 15-17**

```
1  // This program has a subtle error in the virtual functions.
2  #include <iostream>
3  using namespace std;
4
5  class Base
6  {
7  public:
8      virtual void functionA(int arg) const
9          { cout << "This is Base::functionA" << endl; }
10 };
11
12 class Derived : public Base
13 {
14 public:
15     virtual void functionA(long arg) const
16         { cout << "This is Derived::functionA" << endl; }
17 };
18
19 int main()
20 {
21     // Allocate instances of the Derived class.
22     Base *b = new Derived();
23     Derived *d = new Derived();
24
25     // Call functionA with the two pointers.
26     b->functionA(99);
```

*(program continues)*

**Program 15-17** *(continued)*

```
27      d->functionA(99);
28
29      return 0;
30  }
```

**Program Output**

```
This is Base::functionA
This is Derived::functionA
```

Both the `Base` class and the `Derived` class have a virtual member function named `functionA`.

Notice that in lines 22 and 23 in the `main` function, we allocate two instances of the `Derived` class. We reference one of the instances with a `Base` class pointer (`b`), and we reference the other instance with a `Derived` class pointer (`d`). When we call `functionA` in lines 26 and 27, we might expect that the `Derived` class's `functionA` would be called in both lines. This is not the case, however, as you can see from the program's output.

The `functionA` in the `Derived` class does not override the `functionA` in the `Base` class because the function signatures are different. The `functionA` in the `Base` class takes an `int` argument, but the one in the `Derived` class takes a `long` argument. So, `functionA` in the `Derived` class merely overloads `functionA` in the `Base` class.

To make sure that a member function in a derived class overrides a virtual member function in a base class, you can use the `override` key word in the derived class's function prototype (or the function header, if the function is written inline). The `override` key word tells the compiler that the function is supposed to override a function in the base class. It will cause a compiler error if the function does not actually override any functions. Program 15-18 demonstrates how Program 15-17 can be fixed so that the `Derived` class function does, in fact, override the `Base` class function. Notice in line 15 that we have changed the parameter in the `Derived` class function to an `int`, and we have added the `override` key word to the function header.

**Program 15-18**

```
1   // This program demonstrates the override key word.
2   #include <iostream>
3   using namespace std;
4
5   class Base
6   {
7   public:
8       virtual void functionA(int arg) const
9       { cout << "This is Base::functionA" << endl; }
10  };
11
12  class Derived : public Base
13  {
14  public:
```

```
15      virtual void functionA(int arg) const override
16      { cout << "This is Derived::functionA" << endl; }
17  };
18
19  int main()
20  {
21      // Allocate instances of the Derived class.
22      Base *b = new Derived();
23      Derived *d = new Derived();
24
25      // Call functionA with the two pointers.
26      b->functionA(99);
27      d->functionA(99);
28
29      return 0;
30  }
```

**Program Output**

```
This is Derived::functionA
This is Derived::functionA
```

### Preventing a Member Function from Being Overridden

In some derived classes, you might want to make sure that a virtual member function cannot be overridden any further down the class hierarchy. When a member function is declared with the `final` key word, it cannot be overridden in a derived class. The following member function prototype is an example that uses the `final` key word:

```
virtual void message() const final;
```

If a derived class attempts to override a `final` member function, the compiler generates an error.

## 15.7 Abstract Base Classes and Pure Virtual Functions

**CONCEPT:** **An abstract base class cannot be instantiated, but other classes are derived from it. A pure virtual function is a virtual member function of a base class that must be overridden. When a class contains a pure virtual function as a member, that class becomes an abstract base class.**

Sometimes it is helpful to begin a class hierarchy with an *abstract base class*. An abstract base class is not instantiated itself, but serves as a base class for other classes. The abstract base class represents the generic, or abstract, form of all the classes that are derived from it.

For example, consider a factory that manufactures airplanes. The factory does not make a generic airplane, but makes three specific types of planes: two different models of prop-driven planes, and one commuter jet model. The computer software that catalogs the planes might use an abstract base class called `Airplane`. That class has members representing the common characteristics of all airplanes. In addition, it has classes for each of the three specific airplane models the factory manufactures. These classes have members representing

the unique characteristics of each type of plane. The base class, `Airplane`, is never instantiated, but is used to derive the other classes.

A class becomes an abstract base class when one or more of its member functions is a *pure virtual function*. A pure virtual function is a virtual member function declared in a manner similar to the following:

```
virtual void showInfo() = 0;
```

The `= 0` notation indicates that `showInfo` is a pure virtual function. Pure virtual functions have no body, or definition, in the base class. They must be overridden in derived classes. Additionally, the presence of a pure virtual function in a class prevents a program from instantiating the class. The compiler will generate an error if you attempt to define an object of an abstract base class.

For example, look at the following abstract base class `Student`. It holds data common to all students, but does not hold all the data needed for students of specific majors.

### Contents of `Student.h`

```
// Specification file for the Student class
#ifndef STUDENT_H
#define STUDENT_H
#include <string>
using namespace std;

class Student
{
protected:
     string name;         // Student name
     string idNumber;     // Student ID
     int yearAdmitted;    // Year student was admitted
public:
     // Default constructor
     Student()
          { name = "";
            idNumber = "";
            yearAdmitted = 0; }

     // Constructor
     Student(string n, string id, int year)
     { set(n, id, year); }

     // The set function sets the attribute data.
     void set(string n, string id, int year)
          { name = n;               // Assign the name
            idNumber = id;          // Assign the ID number
            yearAdmitted = year; }  // Assign the year admitted

     // Accessor functions
     const string getName() const
          { return name; }

     const string getIdNum() const
          { return idNumber; }
```

```
36
37      int getYearAdmitted() const
38          { return yearAdmitted; }
39
40      // Pure virtual function
41      virtual int getRemainingHours() const = 0;
42  };
43  #endif
```

The `Student` class contains members for storing a student's name, ID number, and year admitted. It also has constructors and a mutator function for setting values in the `name`, `idNumber`, and `yearAdmitted` members. Accessor functions are provided that return the values in the `name`, `idNumber`, and `yearAdmitted` members. A pure virtual function named `getRemainingHours` is also declared.

The pure virtual function must be overridden in classes derived from the `Student` class. It was made a pure virtual function because this class is intended to be the base for classes that represent students of specific majors. For example, a `CsStudent` class might hold the data for a computer science student, and a `BiologyStudent` class might hold the data for a biology student. Computer science students must take courses in different disciplines than those taken by biology students. It stands to reason that the `CsStudent` class will calculate the number of hours taken in a different manner than the `BiologyStudent` class.

Let's look at an example of the `CsStudent` class.

### Contents of `CsStudent.h`

```
1   // Specification file for the CsStudent class
2   #ifndef CSSTUDENT_H
3   #define CSSTUDENT_H
4   #include "Student.h"
5
6   // Constants for required hours
7   const int MATH_HOURS = 20;     // Math hours
8   const int CS_HOURS = 40;       // Computer science hours
9   const int GEN_ED_HOURS = 60;   // General Ed hours
10
11  class CsStudent : public Student
12  {
13  private:
14      int mathHours;     // Hours of math taken
15      int csHours;       // Hours of Computer Science taken
16      int genEdHours;    // Hours of general education taken
17
18  public:
19      // Default constructor
20      CsStudent() : Student()
21          { mathHours = 0;
22            csHours = 0;
23            genEdHours = 0; }
24
25      // Constructor
26      CsStudent(string n, string id, int year) :
```

```
27          Student(n, id, year)
28          { mathHours = 0;
29            csHours = 0;
30            genEdHours = 0; }
31
32      // Mutator functions
33      void setMathHours(int mh)
34          { mathHours = mh; }
35
36      void setCsHours(int csh)
37          { csHours = csh; }
38
39      void setGenEdHours(int geh)
40          { genEdHours = geh; }
41
42      // Overridden getRemainingHours function,
43      // defined in CsStudent.cpp
44      virtual int getRemainingHours() const;
45  };
46  #endif
```

This file declares the following `const int` member variables in lines 7 through 9: `MATH_HOURS`, `CS_HOURS`, and `GEN_ED_HOURS`. These variables hold the required number of math, computer science, and general education hours for a computer science student. The `CsStudent` class, which derives from the `Student` class, declares the following member variables in lines 14 through 16: `mathHours`, `csHours`, and `genEdHours`. These variables hold the number of math, computer science, and general education hours taken by the student. Mutator functions are provided to store values in these variables. In addition, the class overrides the pure virtual `getRemainingHours` function in the `CsStudent.cpp` file.

### Contents of `CsStudent.cpp`

```
1   #include <iostream>
2   #include "CsStudent.h"
3   using namespace std;
4
5   //*************************************************
6   // The CsStudent::getRemainingHours function returns *
7   // the number of hours remaining to be taken.       *
8   //*************************************************
9
10  int CsStudent::getRemainingHours() const
11  {
12      int reqHours,   // Total required hours
13      remainingHours; // Remaining hours
14
15      // Calculate the required hours.
16      reqHours = MATH_HOURS + CS_HOURS + GEN_ED_HOURS;
17
18      // Calculate the remaining hours.
19      remainingHours = reqHours - (mathHours + csHours +
20                       genEdHours);
```

```
21
22       // Return the remaining hours.
23       return remainingHours;
24   }
```

Program 15-19 provides a simple demonstration of the class.

**Program 15-19**

```
1  // This program demonstrates the CsStudent class, which is
2  // derived from the abstract base class, Student.
3  #include <iostream>
4  #include "CsStudent.h"
5  using namespace std;
6
7  int main()
8  {
9       // Create a CsStudent object for a student.
10      CsStudent student("Jennifer Haynes", "167W98337", 2006);
11
12      // Store values for Math, Computer Science, and General
13      // Ed hours.
14      student.setMathHours(12);   // Student has taken 12 Math hours
15      student.setCsHours(20);     // Student has taken 20 CS hours
16      student.setGenEdHours(40);  // Student has taken 40 Gen Ed hours
17
18      // Display the number of remaining hours.
19      cout << "The student " << student.getName()
20           << " needs to take " << student.getRemainingHours()
21           << " more hours to graduate.\n";
22
23      return 0;
24 }
```

**Program Output**

```
The student Jennifer Haynes needs to take 48 more hours to graduate.
```

Remember the following points about abstract base classes and pure virtual functions:

- When a class contains a pure virtual function, it is an abstract base class.
- Pure virtual functions are declared with the `= 0` notation.
- Abstract base classes cannot be instantiated.
- Pure virtual functions have no body, or definition, in the base class.
- A pure virtual function *must* be overridden at some point in a derived class in order for it to become nonabstract.

## Checkpoint

15.9 Explain the difference between overloading a function and redefining a function.

15.10 Explain the difference between static binding and dynamic binding.

15.11 Are virtual functions statically bound or dynamically bound?

15.12 What will the following program display?

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

class First
{
protected:
   int a;
public:
   First(int x = 1)
      { a = x; }

   int getVal()
      { return a; }
};

class Second : public First
{
private:
   int b;

public:
   Second(int y = 5)
      { b = y; }
   int getVal()
      { return b; }
};

int main()
{
   First object1;
   Second object2;

   cout << object1.getVal() << endl;
   cout << object2.getVal() << endl;
   return 0;
}
```

15.13 What will the following program display?

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

class First
{
protected:
   int a;
public:
   First(int x = 1)
      { a = x; }

   void twist()
      { a *= 2; }
   int getVal()
      { twist(); return a; }
};
```

```
class Second : public First
{
private:
   int b;
public:
   Second(int y = 5)
      { b = y; }

   void twist()
      { b *= 10; }
};

int main()
{
   First object1;
   Second object2;

   cout << object1.getVal() << endl;
   cout << object2.getVal() << endl;
   return 0;
}
```

15.14 What will the following program display?

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

class First
{
protected:
   int a;
public:
   First(int x = 1)
      { a = x; }

   virtual void twist()
      { a *= 2; }

   int getVal()
      { twist(); return a; }
};

class Second : public First
{
private:
   int b;
public:
   Second(int y = 5)
      { b = y; }
   virtual void twist()
      { b *= 10; }
};

int main()
{
   First object1;
   Second object2;
```

```
    cout << object1.getVal() << endl;
    cout << object2.getVal() << endl;
    return 0;
}
```

15.15 What will the following program display?

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

class Base
{
protected:
    int baseVar;
public:
    Base(int val = 2)
        { baseVar = val; }

    int getVar()
        { return baseVar; }
};

class Derived : public Base
{
private:
    int derivedVar;

public:
    Derived(int val = 100)
        { derivedVar = val; }
    int getVar()
        { return derivedVar; }
};

int main()
{
    Base *optr = nullptr;
    Derived object;

    optr = &object;
    cout << optr->getVar() << endl;
    return 0;
}
```

## 15.8 Multiple Inheritance

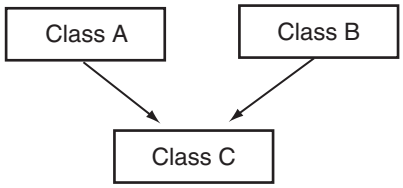
**CONCEPT: Multiple inheritance is when a derived class has two or more base classes.**

Previously we discussed how a class may be derived from a second class that is itself derived from a third class. The series of classes establishes a chain of inheritance. In such a scheme, you might be tempted to think of the lowest class in the chain as having multiple base classes. A base class, however, should be thought of as the class that another class is directly

derived from. Even though there may be several classes in a chain, each class (below the topmost class) only has one base class.

Another way of combining classes is through multiple inheritance. *Multiple inheritance* is when a class has two or more base classes. This is illustrated in Figure 15-6.

**Figure 15-6**




In Figure 15-6, class C is directly derived from classes A and B and inherits the members of both. Neither class A nor B, however, inherits members from the other. Their members are only passed down to class C. Let's look at an example of multiple inheritance. Consider the two classes declared here:

**Contents of `Date.h`**

```
1  // Specification file for the Date class
2  #ifndef DATE_H
3  #define DATE_H
4
5  class Date
6  {
7  protected:
8       int day;
9       int month;
10      int year;
11 public:
12      // Default constructor
13      Date(int d, int m, int y)
14         { day = 1; month = 1; year = 1900; }
15
16      // Constructor
17      Date(int d, int m, int y)
18         { day = d; month = m; year = y; }
19
20      // Accessors
21      int getDay() const
22         { return day; }
23
24      int getMonth() const
25         { return month; }
26
27      int getYear() const
28         { return year; }
29 };
30 #endif
```

### Contents of `Time.h`

```
// Specification file for the Time class
#ifndef TIME_H
#define TIME_H

class Time
{
protected:
     int hour;
     int min;
     int sec;
public:
     // Default constructor
     Time()
        { hour = 0; min = 0; sec = 0; }

     // Constructor
     Time(int h, int m, int s)
        { hour = h; min = m; sec = s; }

     // Accessor functions
     int getHour() const
        { return hour; }

     int getMin() const
        { return min; }

     int getSec() const
        { return sec; }
};
#endif
```

These classes are designed to hold integers that represent the date and time. They both can be used as base classes for a third class we will call `DateTime`:

### Contents of `DateTime.h`

```
// Specification file for the DateTime class
#ifndef DATETIME_H
#define DATETIME_H
#include <string>
#include "Date.h"
#include "Time.h"
using namespace std;

class DateTime : public Date, public Time
{
public:
     // Default constructor
     DateTime();

     // Constructor
     DateTime(int, int, int, int, int, int);
```

```
17
18      // The showDateTime function displays the
19      // date and the time.
20      void showDateTime() const;
21  };
22  #endif
```

In line 9, the first line in the `DateTime` declaration reads

```
class DateTime : public Date, public Time
```

Notice there are two base classes listed, separated by a *comma*. Each base class has its own access specification. The general format of the first line of a class declaration with multiple base classes is

```
class DerivedClassName : AccessSpecification BaseClassName,
                         AccessSpecification BaseClassName [, ...]
```

The notation in the square brackets indicates that the list of base classes with their access specifications may be repeated. (It is possible to have several base classes.)

### Contents of `DateTime.cpp`

```
1   // Implementation file for the DateTime class
2   #include <iostream>
3   #include <string>
4   #include "DateTime.h"
5   using namespace std;
6
7   //*********************************************
8   // Default constructor                        *
9   // Note that this constructor does nothing other *
10  // than call default base class constructors.    *
11  //*********************************************
12  DateTime::DateTime() : Date(), Time()
13  {}
14
15  //*********************************************
16  // Constructor                                *
17  // Note that this constructor does nothing other *
18  // than call base class constructors.         *
19  //*********************************************
20  DateTime::DateTime(int dy, int mon, int yr, int hr, int mt, int sc) :
21      Date(dy, mon, yr), Time(hr, mt, sc)
22  {}
23
24  //*********************************************
25  // The showDateTime member function displays the *
26  // date and the time.                         *
27  //*********************************************
28  void DateTime::showDateTime() const
29  {
30      // Display the date in the form MM/DD/YYYY.
31      cout << getMonth() << "/" << getDay() << "/" << getYear() << " ";
```

```
32
33       // Display the time in the form HH:MM:SS.
34       cout << getHour() << ":" << getMin() << ":" << getSec() << endl;
35   }
```

The class has two constructors: a default constructor and a constructor that accepts arguments for each component of a date and time. Let's look at the function header for the default constructor, in line 12:

```
DateTime::DateTime() : Date(), Time()
```

After the `DateTime` constructor's parentheses is a colon, followed by calls to the `Date` constructor and the `Time` constructor. The calls are separated by a comma. When using multiple inheritance, the general format of a derived class's constructor header is

```
DerivedClassName(ParameterList) : BaseClassName(ArgumentList),
                                  BaseClassName(ArgumentList)[, ...]
```

Look at the function header for the second constructor, which appears in lines 20 and 21:

```
DateTime::DateTime(int dy, int mon, int yr, int hr, int mt, int sc) :
    Date(dy, mon, yr), Time(hr, mt, sc)
```

This `DateTime` constructor accepts arguments for the day (`dy`), month (`mon`), year (`yr`), hour (`hr`), minute (`mt`), and second (`sc`). The `dy`, `mon`, and `yr` parameters are passed as arguments to the `Date` constructor. The `hr`, `mt`, and `sc` parameters are passed as arguments to the `Time` constructor.

The order that the base class constructor calls appear in the list does not matter. They are always called in the order of inheritance. That is, they are always called in the order they are listed in the first line of the class declaration. Here is line 9 from the `DateTime.h` file:

```
class DateTime : public Date, public Time
```

Because `Date` is listed before `Time` in the `DateTime` class declaration, the `Date` constructor will always be called first. If the classes use destructors, they are always called in reverse order of inheritance. Program 15-20 shows these classes in use.

**Program 15-20**

```
1  // This program demonstrates a class with multiple inheritance.
2  #include "DateTime.h"
3  using namespace std;
4
5  int main()
6  {
7       // Define a DateTime object and use the default
8       // constructor to initialize it.
9       DateTime emptyDay;
10
11      // Display the object's date and time.
12      emptyDay.showDateTime();
13
14      // Define a DateTime object and initialize it
15      // with the date 2/4/1960 and the time 5:32:27.
```

```
16      DateTime pastDay(2, 4, 1960, 5, 32, 27);
17
18      // Display the object's date and time.
19      pastDay.showDateTime();
20      return 0;
21  }
```

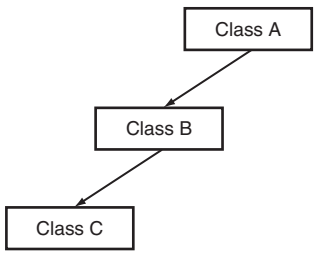
**Program Output**

```
1/1/1900 0:0:0
4/2/1960 5:32:27
```

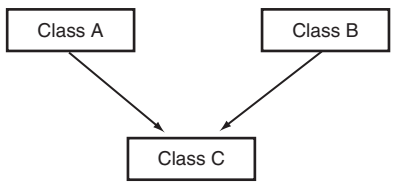
**NOTE:** It should be noted that multiple inheritance opens the opportunity for a derived class to have ambiguous members. That is, two base classes may have member variables or functions of the same name. In situations like these, the derived class should always redefine or override the member functions. Calls to the member functions of the appropriate base class can be performed within the derived class using the scope resolution operator (::). The derived class can also access the ambiguously named member variables of the correct base class using the scope resolution operator. If these steps aren't taken, the compiler will generate an error when it can't tell which member is being accessed.

## Checkpoint

15.16 Does the following diagram depict multiple inheritance or a chain of inheritance?




15.17 Does the following diagram depict multiple inheritance or a chain of inheritance?




15.18 Examine the following classes. The table lists the variables that are members of the `Third` class (some are inherited). Complete the table by filling in the access specification each member will have in the `Third` class. Write "inaccessible" if a member is inaccessible to the `Third` class.

```
class First
{
   private:
```

```
      int a;
   protected:
      double b;
   public:
      long c;
};

class Second : protected First
{
   private:
      int d;
   protected:
      double e;
   public:
      long f;
};

class Third : public Second
{
   private:
      int g;
   protected:
      double h;
   public:
      long i;
};
```

| Member Variable | Access Specification in Third Class |
|---|---|
| a | |
| b | |
| c | |
| d | |
| e | |
| f | |
| g | |
| h | |
| i | |

15.19 Examine the following class declarations:

```
class Van
{
protected:
   int passengers;
public:
   Van(int p)
   { passengers = p; }
};
```

```
class FourByFour
{
protected:
   double cargoWeight;
public:
   FourByFour(float w)
      { cargoWeight = w; }
};
```

Write the declaration of a class named `SportUtility`. The class should be derived from both the `Van` and `FourByFour` classes above. (This should be a case of multiple inheritance, where both `Van` and `FourByFour` are base classes.)

## Review Questions and Exercises

### Short Answer

1. What is an "is a" relationship?
2. A program uses two classes: `Dog` and `Poodle`. Which class is the base class and which is the derived class?
3. How does base class access specification differ from class member access specification?
4. What is the difference between a protected class member and a private class member?
5. Can a derived class ever directly access the private members of its base class?
6. Which constructor is called first, that of the derived class or the base class?
7. What is the difference between redefining a base class function and overriding a base class function?
8. When does static binding take place? When does dynamic binding take place?
9. What is an abstract base class?
10. A program has a class `Potato`, which is derived from the class `Vegetable`, which is derived from the class `Food`. Is this an example of multiple inheritance? Why or why not?
11. What base class is named in the line below?

    ```
    class Pet : public Dog
    ```

12. What derived class is named in the line below?

    ```
    class Pet : public Dog
    ```

13. What is the class access specification of the base class named below?

    ```
    class Pet : public Dog
    ```

14. What is the class access specification of the base class named below?

    ```
    class Pet : Fish
    ```

15. Protected members of a base class are like _____ members, except they may be accessed by derived classes.
16. Complete the table on the next page by filling in private, protected, public, or inaccessible in the right-hand column:

| In a private base class, this base class MEMBER access specification… | …becomes this access specification in the derived class. |
|---|---|
| `private` | |
| `protected` | |
| `public` | |

17. Complete the table below by filling in private, protected, public, or inaccessible in the right-hand column:

| In a protected base class, this base class MEMBER access specification… | …becomes this access specification in the derived class. |
|---|---|
| `private` | |
| `protected` | |
| `public` | |

18. Complete the table below by filling in private, protected, public, or inaccessible in the right-hand column:

| In a public base class, this base class MEMBER access specification… | …becomes this access specification in the derived class. |
|---|---|
| `private` | |
| `protected` | |
| `public` | |

## Fill-in-the-Blank

19. A derived class inherits the _____ of its base class.
20. When both a base class and a derived class have constructors, the base class's constructor is called _____ (first/last).
21. When both a base class and a derived class have destructors, the base class's constructor is called _____ (first/last).
22. An overridden base class function may be called by a function in a derived class by using the _____ operator.
23. When a derived class redefines a function in a base class, which version of the function do objects that are defined of the base class call? _____
24. A(n) _____ member function in a base class expects to be overridden in a derived class.
25. _____ binding is when the compiler binds member function calls at compile time.
26. _____ binding is when a function call is bound at runtime.
27. _____ is when member functions in a class hierarchy behave differently, depending upon which object performs the call.
28. When a pointer to a base class is made to point to a derived class, the pointer ignores any _____ the derived class performs, unless the function is _____.

29. A(n) _____ class cannot be instantiated.
30. A(n) _____ function has no body, or definition, in the class in which it is declared.
31. A(n) _____ of inheritance is where one class is derived from a second class, which in turn is derived from a third class.
32. _____ is where a derived class has two or more base classes.
33. In multiple inheritance, the derived class should always _____ a function that has the same name in more than one base class.

## Algorithm Workbench

34. Write the first line of the declaration for a `Poodle` class. The class should be derived from the `Dog` class with public base class access.
35. Write the first line of the declaration for a `SoundSystem` class. Use multiple inheritance to base the class on the `CDplayer` class, the `Tuner` class, and the `CassettePlayer` class. Use public base class access in all cases.
36. Suppose a class named `Tiger` is derived from both the `Felis` class and the `Carnivore` class. Here is the first line of the `Tiger` class declaration:

```
class Tiger : public Felis, public Carnivore
```

Here is the function header for the Tiger constructor:

```
Tiger(int x, int y) : Carnivore(x), Felis(y)
```

Which base class constructor is called first, `Carnivore` or `Felis`?

37. Write the declaration for class `B`. The class's members should be
    - `m`, an integer. This variable should not be accessible to code outside the class or to member functions in any class derived from class `B`.
    - `n`, an integer. This variable should not be accessible to code outside the class, but should be accessible to member functions in any class derived from class `B`.
    - `setM`, `getM`, `setN`, and `getN`. These are the set and get functions for the member variables `m` and `n`. These functions should be accessible to code outside the class.
    - `calc`, a public virtual member function that returns the value of `m` times `n`.

    Next write the declaration for class `D`, which is derived from class `B`. The class's members should be
    - `q`, a `float`. This variable should not be accessible to code outside the class but should be accessible to member functions in any class derived from class `D`.
    - `r`, a `float`. This variable should not be accessible to code outside the class, but should be accessible to member functions in any class derived from class `D`.
    - `setQ`, `getQ`, `setR`, and `getR`. These are the set and get functions for the member variables `q` and `r`. These functions should be accessible to code outside the class.
    - `calc`, a public member function that overrides the base class `calc` function. This function should return the value of `q` times `r`.

## True or False

38. T F The base class's access specification affects the way base class member functions may access base class member variables.
39. T F The base class's access specification affects the way the derived class inherits members of the base class.

40. T F Private members of a private base class become inaccessible to the derived class.
41. T F Public members of a private base class become private members of the derived class.
42. T F Protected members of a private base class become public members of the derived class.
43. T F Public members of a protected base class become private members of the derived class.
44. T F Private members of a protected base class become inaccessible to the derived class.
45. T F Protected members of a public base class become public members of the derived class.
46. T F The base class constructor is called after the derived class constructor.
47. T F The base class destructor is called after the derived class destructor.
48. T F It isn't possible for a base class to have more than one constructor.
49. T F Arguments are passed to the base class constructor by the derived class constructor.
50. T F A member function of a derived class may not have the same name as a member function of the base class.
51. T F Pointers to a base class may be assigned the address of a derived class object.
52. T F A base class may not be derived from another class.

## Find the Errors

Each of the class declarations and/or member function definitions below has errors. Find as many as you can.

53.
```
class Car, public Vehicle
{
   public:
      Car();
      ~Car();
   protected:
      int passengers;
}
```

54.
```
class Truck, public : Vehicle, protected
{
   private:
      double cargoWeight;
   public:
      Truck();
      ~Truck();
};
```

55.
```
class SnowMobile : Vehicle
{
   protected:
```

```
        int horsePower;
        double weight;
    public:
        SnowMobile(int h, double w), Vehicle(h)
            { horsePower = h; }
        ~SnowMobile();
};
```

56.
```
class Table : public Furniture
{
   protected:
      int numSeats;
   public:
      Table(int n) : Furniture(numSeats)
         { numSeats = n; }
      ~Table();
};
```

57.
```
class Tank : public Cylinder
{
    private:
        int fuelType;
        double gallons;
    public:
        Tank();
        ~Tank();
        void setContents(double);
        void setContents(double);
};
```

58.
```
class Three : public Two : public One
{
    protected:
        int x;
    public:
        Three(int a, int b, int c), Two(b), Three(c)
            { x = a; }
        ~Three();
};
```

## Programming Challenges

VideoNote
**Solving the `Employee` and `ProductionWorker` Classes Problem**

1. **`Employee` and `ProductionWorker` Classes**

   Design a class named `Employee`. The class should keep the following information in

   - Employee name
   - Employee number
   - Hire date

   Write one or more constructors and the appropriate accessor and mutator functions for the class.

Next, write a class named `ProductionWorker` that is derived from the `Employee` class. The `ProductionWorker` class should have member variables to hold the following information:

- Shift (an integer)
- Hourly pay rate (a `double`)

The workday is divided into two shifts: day and night. The shift variable will hold an integer value representing the shift that the employee works. The day shift is shift 1, and the night shift is shift 2. Write one or more constructors and the appropriate accessor and mutator functions for the class. Demonstrate the classes by writing a program that uses a `ProductionWorker` object.

2. **`ShiftSupervisor` Class**

In a particular factory a shift supervisor is a salaried employee who supervises a shift. In addition to a salary, the shift supervisor earns a yearly bonus when his or her shift meets production goals. Design a `ShiftSupervisor` class that is derived from the `Employee` class you created in Programming Challenge 1. The `ShiftSupervisor` class should have a member variable that holds the annual salary and a member variable that holds the annual production bonus that a shift supervisor has earned. Write one or more constructors and the appropriate accessor and mutator functions for the class. Demonstrate the class by writing a program that uses a `ShiftSupervisor` object.

3. **`TeamLeader` Class**

In a particular factory, a team leader is an hourly paid production worker who leads a small team. In addition to hourly pay, team leaders earn a fixed monthly bonus. Team leaders are required to attend a minimum number of hours of training per year. Design a `TeamLeader` class that extends the `ProductionWorker` class you designed in Programming Challenge 1. The `TeamLeader` class should have member variables for the monthly bonus amount, the required number of training hours, and the number of training hours that the team leader has attended. Write one or more constructors and the appropriate accessor and mutator functions for the class. Demonstrate the class by writing a program that uses a `TeamLeader` object.

4. **Time Format**

In Program 15-20, the file `Time.h` contains a `Time` class. Design a class called `MilTime` that is derived from the `Time` class. The `MilTime` class should convert time in military (24-hour) format to the standard time format used by the `Time` class. The class should have the following member variables:

`milHours`: Contains the hour in 24-hour format. For example, 1:00 pm would be stored as 1300 hours, and 4:30 pm would be stored as 1630 hours.

`milSeconds`: Contains the seconds in standard format.

The class should have the following member functions:

`Constructor`: The constructor should accept arguments for the hour and seconds, in military format. The time should then be converted to standard time and stored in the `hours`, `min`, and `sec` variables of the `Time` class.

`setTime`: Accepts arguments to be stored in the `milHours` and `milSeconds` variables. The time should then be converted to standard time and stored in the `hours`, `min`, and `sec` variables of the `Time` class.

`getHour`: Returns the hour in military format.

`getStandHr`: Returns the hour in standard format.

Demonstrate the class in a program that asks the user to enter the time in military format. The program should then display the time in both military and standard format.

*Input Validation: The* `MilTime` *class should not accept hours greater than 2359, or less than 0. It should not accept seconds greater than 59 or less than 0.*

**5. Time Clock**

Design a class named `TimeClock`. The class should be derived from the `MilTime` class you designed in Programming Challenge 4. The class should allow the programmer to pass two times to it: starting time and ending time. The class should have a member function that returns the amount of time elapsed between the two times. For example, if the starting time is 900 hours (9:00 am), and the ending time is 1300 hours (1:00 pm), the elapsed time is 4 hours.

*Input Validation: The class should not accept hours greater than 2359 or less than 0.*

**6. `Essay` class**

Design an `Essay` class that is derived from the `GradedActivity` class presented in this chapter. The `Essay` class should determine the grade a student receives on an essay. The student's essay score can be up to 100, and is determined in the following manner:

- Grammar: 30 points
- Spelling: 20 points
- Correct length: 20 points
- Content: 30 points

Demonstrate the class in a simple program.

**7. `PersonData` and `CustomerData` classes**

Design a class named `PersonData` with the following member variables:

- `lastName`
- `firstName`
- `address`
- `city`
- state
- `zip`
- `phone`

Write the appropriate accessor and mutator functions for these member variables.

Next, design a class named `CustomerData`, which is derived from the `PersonData` class. The `CustomerData` class should have the following member variables:

- `customerNumber`
- `mailingList`

The `customerNumber` variable will be used to hold a unique integer for each customer. The `mailingList` variable should be a `bool`. It will be set to `true` if the customer wishes to be on a mailing list, or `false` if the customer does not wish to be on a mailing list. Write appropriate accessor and mutator functions for these member variables. Demonstrate an object of the `CustomerData` class in a simple program.

8. **`PreferredCustomer` Class**

   A retail store has a preferred customer plan where customers may earn discounts on all their purchases. The amount of a customer's discount is determined by the amount of the customer's cumulative purchases in the store.

   - When a preferred customer spends $500, he or she gets a 5% discount on all future purchases.
   - When a preferred customer spends $1,000, he or she gets a 6% discount on all future purchases.
   - When a preferred customer spends $1,500, he or she gets a 7% discount on all future purchases.
   - When a preferred customer spends $2,000 or more, he or she gets a 10% discount on all future purchases.

   Design a class named `PreferredCustomer`, which is derived from the `CustomerData` class you created in Programming Challenge 7. The `PreferredCustomer` class should have the following member variables:

   - `purchasesAmount` (a `double`)
   - `discountLevel` (a `double`)

   The `purchasesAmount` variable holds the total of a customer's purchases to date. The `discountLevel` variable should be set to the correct discount percentage, according to the store's preferred customer plan. Write appropriate member functions for this class and demonstrate it in a simple program.

   *Input Validation: Do not accept negative values for any sales figures.*

9. **File Filter**

   A file filter reads an input file, transforms it in some way, and writes the results to an output file. Write an abstract file filter class that defines a pure virtual function for transforming a character. Create one derived class of your file filter class that performs encryption, another that transforms a file to all uppercase, and another that creates an unchanged copy of the original file. The class should have the following member function:

   ```
   void doFilter(ifstream &in, ofstream &out)
   ```

   This function should be called to perform the actual filtering. The member function for transforming a single character should have the prototype:

   ```
   char transform(char ch)
   ```

   The encryption class should have a constructor that takes an integer as an argument and uses it as the encryption key.

10. **File Double-Spacer**

    Create a derived class of the abstract filter class of Programming Challenge 9 that double-spaces a file: that is, it inserts a blank line between any two lines of the file.

11. **Course Grades**

    In a course, a teacher gives the following tests and assignments:

    - A **lab activity** that is observed by the teacher and assigned a numeric score.
    - A **pass/fail exam** that has 10 questions. The minimum passing score is 70.
    - An **essay** that is assigned a numeric score.
    - A **final exam** that has 50 questions.

Write a class named `CourseGrades`. The class should have a member named `grades` that is an array of `GradedActivity` pointers. The `grades` array should have four elements, one for each of the assignments previously described. The class should have the following member functions:

`setLab`: This function should accept the address of a `GradedActivity` object as its argument. This object should already hold the student's score for the lab activity. Element 0 of the `grades` array should reference this object.

`setPassFailExam`: This function should accept the address of a `PassFailExam` object as its argument. This object should already hold the student's score for the pass/fail exam. Element 1 of the `grades` array should reference this object.

`setEssay`: This function should accept the address of an `Essay` object as its argument. (See Programming Challenge 6 for the `Essay` class. If you have not completed Programming Challenge 6, use a `GradedActivity` object instead.) This object should already hold the student's score for the essay. Element 2 of the `grades` array should reference this object.

`setPassFailExam`: This function should accept the address of a `FinalExam` object as its argument. This object should already hold the student's score for the final exam. Element 3 of the `grades` array should reference this object.

`print`: This function should display the numeric scores and grades for each element in the `grades` array.

Demonstrate the class in a program.

**12. `Ship`, `CruiseShip`, and `CargoShip` Classes**

Design a `Ship` class that has the following members:

- A member variable for the name of the ship (a string)
- A member variable for the year that the ship was built (a string)
- A constructor and appropriate accessors and mutators
- A virtual `print` function that displays the ship's name and the year it was built.

Design a `CruiseShip` class that is derived from the `Ship` class. The `CruiseShip` class should have the following members:

- A member variable for the maximum number of passengers (an `int`)
- A constructor and appropriate accessors and mutators
- A `print` function that overrides the `print` function in the base class. The `CruiseShip` class's print function should display only the ship's name and the maximum number of passengers.

Design a `CargoShip` class that is derived from the `Ship` class. The `CargoShip` class should have the following members:

- A member variable for the cargo capacity in tonnage (an `int`).
- A constructor and appropriate accessors and mutators.
- A `print` function that overrides the `print` function in the base class. The `CargoShip` class's print function should display only the ship's name and the ship's cargo capacity.

Demonstrate the classes in a program that has an array of `Ship` pointers. The array elements should be initialized with the addresses of dynamically allocated `Ship`, `CruiseShip`, and `CargoShip` objects. (See Program 15-14, lines 17 through 22, for an example of how to do this.) The program should then step through the array, calling each object's `print` function.

**13. Pure Abstract Base Class Project**

Define a pure abstract base class called `BasicShape`. The `BasicShape` class should have the following members:

Private Member Variable:

`area, a double used to hold the shape's area.`

Public Member Functions:

`getArea.` This function should return the value in the member variable area.

`calcArea.` This function should be a pure virtual function.

Next, define a class named `Circle`. It should be derived from the `BasicShape` class. It should have the following members:

Private Member Variables:

`centerX,` a long integer used to hold the x coordinate of the circle's center.

`centerY`, a long integer used to hold the y coordinate of the circle's center.

`radius`, a double used to hold the circle's radius.

Public Member Functions:

constructor—accepts values for `centerX, centerY`, and `radius.` Should call the overridden `calcArea` function described below.

`getCenterX`—returns the value in `centerX.`

`getCenterY`—returns the value in `centerY.`

`calcArea`—calculates the area of the circle (area = 3.14159 * radius * radius) and stores the result in the inherited member area.

Next, define a class named `Rectangle`. It should be derived from the `BasicShape` class. It should have the following members:

Private Member Variables:

`width`, a long integer used to hold the width of the rectangle.

`length`, a long integer used to hold the length of the rectangle.

Public Member Functions:

constructor—accepts values for width and length. Should call the overridden `calcArea` function described below.

`getWidth`—returns the value in width.

`getLength`—returns the value in length.

`calcArea`—calculates the area of the rectangle (area = length * width) and stores the result in the inherited member area.

After you have created these classes, create a driver program that defines a `Circle` object and a `Rectangle` object. Demonstrate that each object properly calculates and reports its area.

## Group Project

**14. Bank Accounts**

This program should be designed and written by a team of students. Here are some suggestions:

- One or more students may work on a single class.
- The requirements of the program should be analyzed so each student is given about the same work load.
- The parameters and return types of each function and class member function should be decided in advance.
- The program will be best implemented as a multi-file program.

Design a generic class to hold the following information about a bank account:

Balance

Number of deposits this month

Number of withdrawals

Annual interest rate

Monthly service charges

The class should have the following member functions:

Constructor: Accepts arguments for the balance and annual interest rate.

`deposit`: A virtual function that accepts an argument for the amount of the deposit. The function should add the argument to the account balance. It should also increment the variable holding the number of deposits.

`withdraw`: A virtual function that accepts an argument for the amount of the withdrawal. The function should subtract the argument from the balance. It should also increment the variable holding the number of withdrawals.

`calcInt`: A virtual function that updates the balance by calculating the monthly interest earned by the account, and adding this interest to the balance. This is performed by the following formulas:

Monthly Interest Rate = (Annual Interest Rate / 12)
Monthly Interest = Balance * Monthly Interest Rate
Balance = Balance + Monthly Interest

`monthlyProc`: A virtual function that subtracts the monthly service charges from the balance, calls the `calcInt` function, and then sets the variables that hold the number of withdrawals, number of deposits, and monthly service charges to zero.

Next, design a savings account class, derived from the generic account class. The savings account class should have the following additional member:

```
status (to represent an active or inactive account)
```

If the balance of a savings account falls below $25, it becomes inactive. (The `status` member could be a flag variable.) No more withdrawals may be made until the balance is raised above $25, at which time the account becomes active again. The savings account class should have the following member functions:

`withdraw`: A function that checks to see if the account is inactive before a withdrawal is made. (No withdrawal will be allowed if the account is not active.) A withdrawal is then made by calling the base class version of the function.

`deposit`: A function that checks to see if the account is inactive before a deposit is made. If the account is inactive and the deposit brings the balance above $25, the account becomes active again. The deposit is then made by calling the base class version of the function.

`monthlyProc`: Before the base class function is called, this function checks the number of withdrawals. If the number of withdrawals for the month is more than 4, a service charge of $1 for each withdrawal above 4 is added to the base class variable that holds the monthly service charges. (Don't forget to check the account balance after the service charge is taken. If the balance falls below $25, the account becomes inactive.)

Next, design a checking account class, also derived from the generic account class. It should have the following member functions:

`withdraw`: Before the base class function is called, this function will determine if a withdrawal (a check written) will cause the balance to go below $0. If the balance goes below $0, a service charge of $15 will be taken from the account. (The withdrawal will not be made.) If there isn't enough in the account to pay the service charge, the balance will become negative and the customer will owe the negative amount to the bank.

`monthlyProc`: Before the base class function is called, this function adds the monthly fee of $5 plus $0.10 per withdrawal (check written) to the base class variable that holds the monthly service charges.

Write a complete program that demonstrates these classes by asking the user to enter the amounts of deposits and withdrawals for a savings account and checking account. The program should display statistics for the month, including beginning balance, total amount of deposits, total amount of withdrawals, service charges, and ending balance.

**NOTE:** You may need to add more member variables and functions to the classes than those listed above.