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

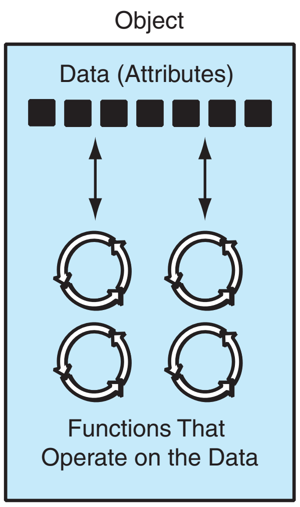
Usually the variables and data structures in a procedural program are passed to the functions that perform the desired operations. As you might imagine, the focus of procedural programming is on creating the functions that operate on the program's data.

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

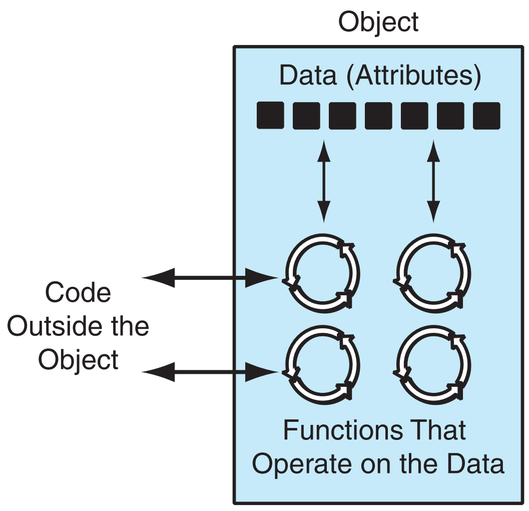
Figure 13-1



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.

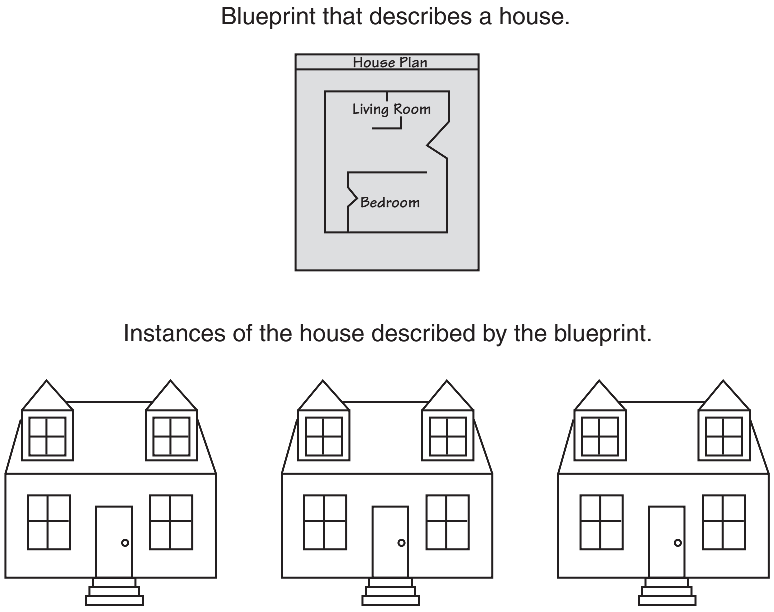
Object Reusability

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.

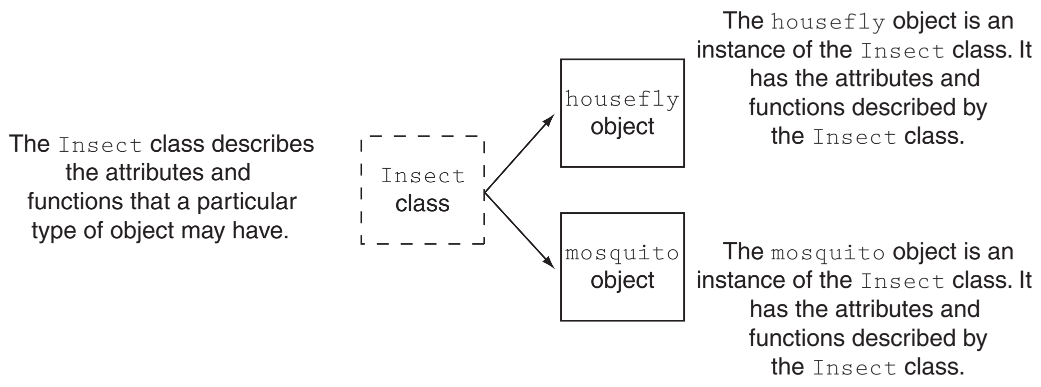
Figure 13-3


Blueprint that describes a house.
House Plan
Living Room
Bedroom
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.

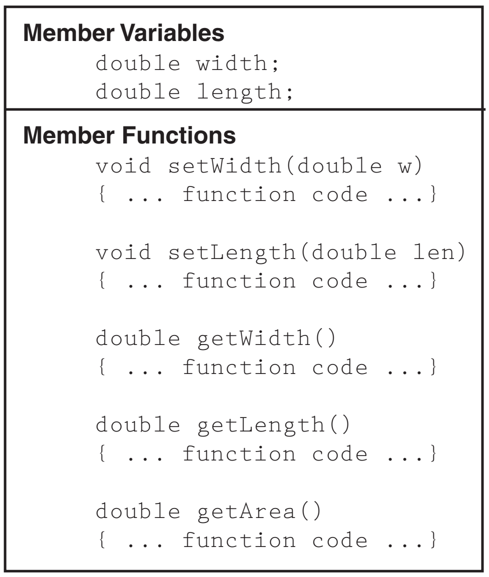
For example, Jessica is an entomologist (someone who studies insects), and she also enjoys writing computer programs. She designs a program to catalog different types of insects. As part of the program, she creates a class named `Insect`, which specifies attributes and member functions for holding and 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


The Insect class describes the attributes and functions that a particular type of object may have.
Insect class
housefly object
mosquito object
The housefly object is an instance of the Insect class. It has the attributes and functions described by the Insect class.
The mosquito object is an instance of the Insect class. It has the attributes and functions described by the Insect class.

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



Member Variables

```
double width;
double length;
```

Member Functions

```
void setWidth(double w)
{ ... function code ...}

void setLength(double len)
{ ... function code ...}

double getWidth()
{ ... function code ...}

double getLength()
{ ... function code ...}

double getArea()
{ ... function code ...}
```

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

```
// This program demonstrates a simple class.
#include <iostream>
using namespace std;

// Rectangle 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;
};

//**************************************************
// setWidth assigns a value to the width member.   *
//**************************************************

```

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

//**************************************************
// setLength assigns a value to the length member. *
//**************************************************

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

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

double Rectangle::getWidth() const
{
    return width;
}

//****************************************************
// getLength returns the value in the length member. *
//****************************************************

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

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

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

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

int main()
{
    Rectangle box;      // Define an instance of the Rectangle class
    double rectWidth;   // Local variable for width
    double rectLength; // Local variable for length

    // Get the rectangle's width and length from the user.
    cout << "This program will calculate the area of a\n";
    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
```

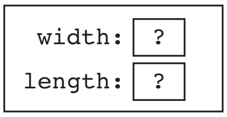
The `Rectangle` class declaration, along with the class's member functions, appears in lines 6 through 62. Inside the `main` function, in line 70, the following statement creates a `Rectangle` object named `box`.

```
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



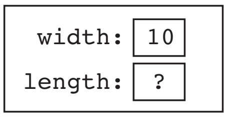
In lines 75 through 79 the program prompts the user to enter the width and length of a rectangle. The width that is entered is stored in the `rectWidth` variable, and the length that is entered is stored in the `rectLength` variable. In line 83 the following statement uses the `box` object to call the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



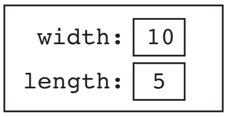
In line 84 the following statement uses the `box` object to call the `setLength` member function, passing the value of the `rectLength` variable as an argument.

```
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
```

```
        return 0;
    }
```

Program Output with Example Input Shown in Bold

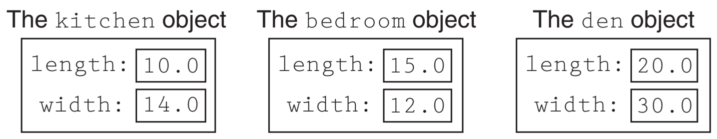
```
What is the kitchen's length? 10 [Enter]
What is the kitchen's width? 14 [Enter]
What is the bedroom's length? 15 [Enter]
What is the bedroom's width? 12 [Enter]
What is the den's length? 20 [Enter]
What is the den's width? 30 [Enter]
The total area of the three rooms is 920
```

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`:

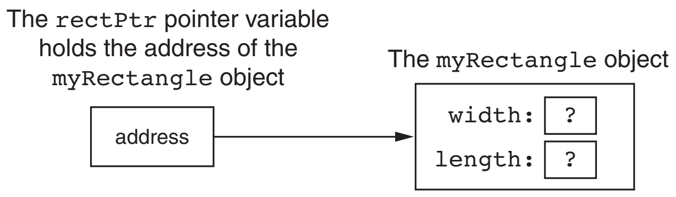
```
Rectangle *rectPtr = nullptr;
```

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

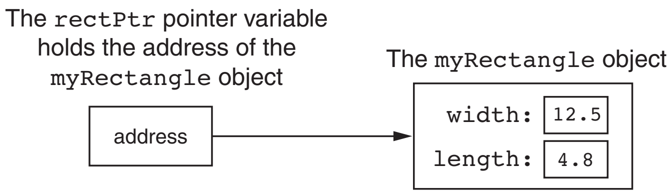

The rectPtr pointer variable holds the address of the myRectangle object
The myRectangle object
address
width: ?
length: ?

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

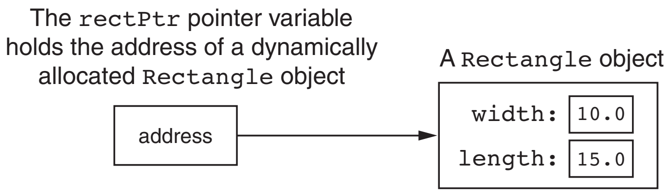

The rectPtr pointer variable holds the address of the myRectangle object
address
The myRectangle object
width: 12.5
length: 4.8

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


The rectPtr pointer variable holds the address of a dynamically allocated Rectangle object
address
A Rectangle object
width: 10.0
length: 15.0

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.
```

```
    totalArea = kitchen->getArea() + bedroom->getArea() +
                den->getArea();

    // Display the total area of the three rooms.
    cout << "The total area of the three rooms is "
         << totalArea << endl;

    // Delete the objects from memory.
    delete kitchen;
    delete bedroom;
    delete den;
    kitchen = nullptr;   // Make kitchen a null pointer.
    bedroom = nullptr;   // Make bedroom a null pointer.
    den = nullptr;       // Make den a null pointer.

    return 0;
}
```

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

```
  1 // This program uses smart pointers to allocate three
  2 // instances of the Rectangle class.
  3 #include <iostream>
  4 #include <memory>
  5 using namespace std;
  6 
  7 // Rectangle class declaration.
```

Lines 8 through 64 have been left out.

```
 65 
 66 //****************************************************
 67 // Function main                                     *
 68 //****************************************************
 69 
 70 int main()
 71 {
 72     double number;            // To hold a number
 73     double totalArea;         // The total area
 74 
 75     // Dynamically allocate the objects.
 76     unique_ptr<Rectangle> kitchen(new Rectangle);
 77     unique_ptr<Rectangle> bedroom(new Rectangle);
 78     unique_ptr<Rectangle> den(new Rectangle);
 79 
 80     // Get the kitchen dimensions.
 81     cout << "What is the kitchen's length? ";
 82     cin >> number;                          // Get the length
 83     kitchen->setLength(number);             // Store in kitchen object
 84     cout << "What is the kitchen's width? ";
 85     cin >> number;                          // Get the width
 86     kitchen->setWidth(number);              // Store in kitchen object
 87 
 88     // Get the bedroom dimensions.
 89     cout << "What is the bedroom's length? ";
 90     cin >> number;                          // Get the length
 91     bedroom->setLength(number);             // Store in bedroom object
 92     cout << "What is the bedroom's width? ";
 93     cin >> number;                          // Get the width
 94     bedroom->setWidth(number);              // Store in bedroom object
 95 
 96     // Get the den dimensions.
 97     cout << "What is the den's length? ";
 98     cin >> number;                          // Get the length
 99     den->setLength(number);                 // Store in den object
100     cout << "What is the den's width? ";
101     cin >> number;                          // Get the width
102     den->setWidth(number);                  // Store in den object
103 
104     // Calculate the total area of the three rooms.
105     totalArea = kitchen->getArea() + bedroom->getArea() +
106                 den->getArea();
107 
```

```
    // Display the total area of the three rooms.
    cout << "The total area of the three rooms is "
         << totalArea << endl;

    return 0;
}
```

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?

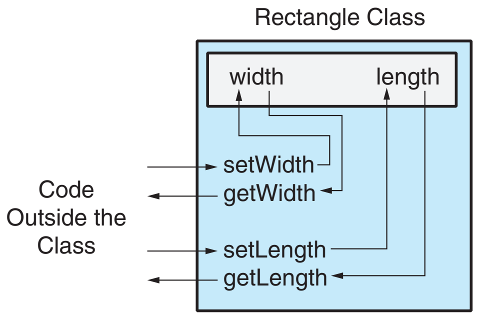
CONCEPT: In object-oriented programming, an object should protect its important data by making it private and providing a public interface to access that data.

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)

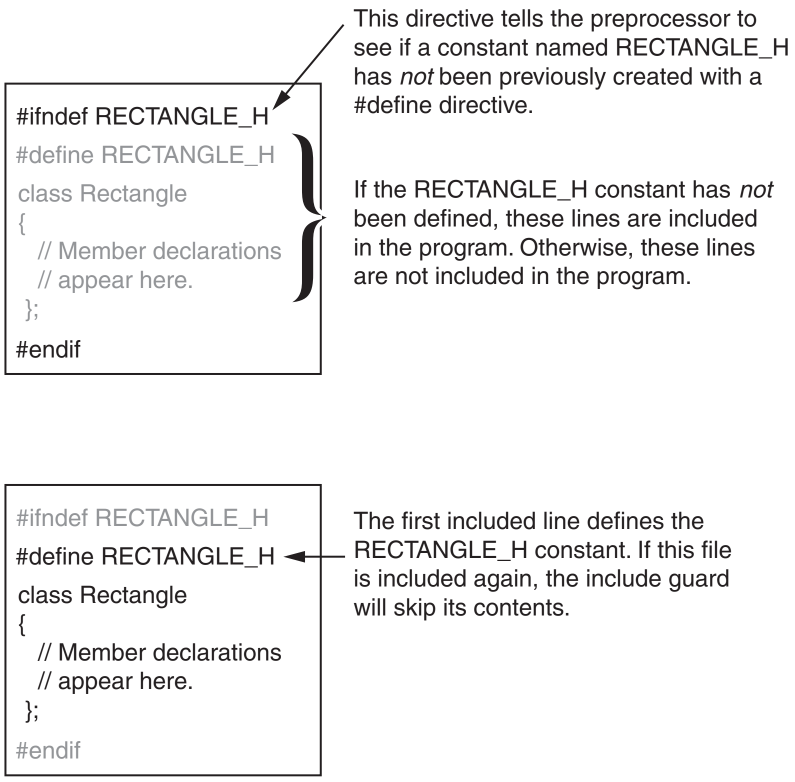
```
 1  // Specification file for the Rectangle class.
 2  #ifndef RECTANGLE_H
 3  #define RECTANGLE_H
 4
 5  // Rectangle class declaration.
 6
 7  class Rectangle
 8  {
 9      private:
10          double width;
11          double length;
12      public:
13          void setWidth(double);
14          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


#ifndef RECTANGLE_H
#define RECTANGLE_H
class Rectangle
{
// Member declarations
// appear here.
};
#endif
This directive tells the preprocessor to see if a constant named RECTANGLE_H has *not* been previously created with a #define directive.
If the RECTANGLE_H constant has *not* been defined, these lines are included in the program. Otherwise, these lines are not included in the program.
#ifndef RECTANGLE_H
#define RECTANGLE_H
class Rectangle
{
// Member declarations
// appear here.
};
#endif
The first included line defines the RECTANGLE_H constant. If this file is included again, the include guard will skip its contents.

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
```

```
void Rectangle::setWidth(double w)
{
    if (w >= 0)
        width = w;
    else
    {
        cout << "Invalid width\n";
        exit(EXIT_FAILURE);
    }
}

//**********************************************************
// setLength sets the value of the member variable length. *
//**********************************************************

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

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

double Rectangle::getWidth() const
{
    return width;
}

//*************************************************************
// getLength returns the value in the member variable length. *
//*************************************************************

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

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

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

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

```
// This program uses the Rectangle class, which is declared in
// the Rectangle.h file. The member Rectangle class's member
// functions are defined in the Rectangle.cpp file. This program
// should be compiled with those files in a project.
#include <iostream>
#include "Rectangle.h"  // Needed for Rectangle class
using namespace std;

int main()
{
    Rectangle box;     // Define an instance of the Rectangle class
    double rectWidth;  // Local variable for width
    double rectLength; // Local variable for length

    // Get the rectangle's width and length from the user.
    cout << "This program will calculate the area of a\n";
    cout << "rectangle. What is the width? ";
    cin >> rectWidth;
    cout << "What is the length? ";
    cin >> rectLength;

```

(program continues)

Program 13-5 *(continued)*

```
    // Store the width and length of the rectangle
    // in the box object.
    box.setWidth(rectWidth);
    box.setLength(rectLength);

    // Display the rectangle's data.
    cout << "Here is the rectangle's data:\n";
    cout << "Width: " << box.getWidth() << endl;
    cout << "Length: " << box.getLength() << endl;
    cout << "Area: " << box.getArea() << endl;
    return 0;
}
```

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`.

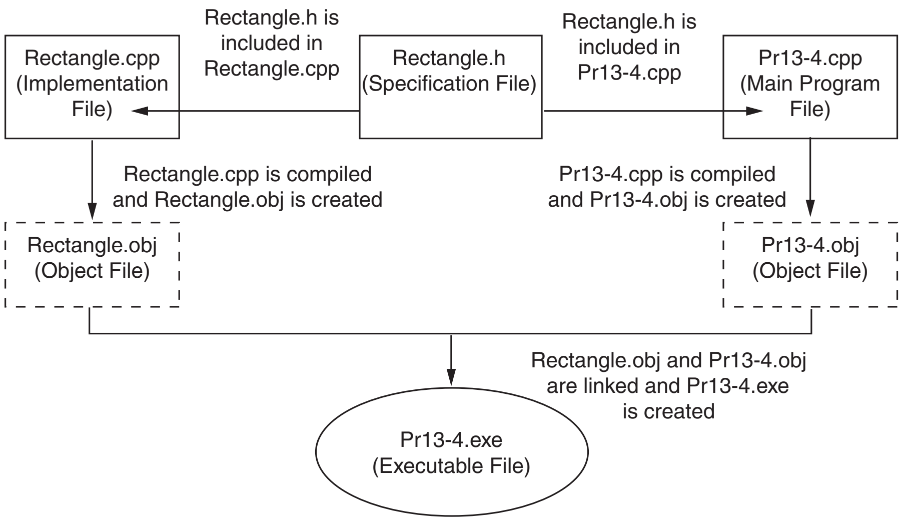
This process is illustrated in Figure 13-15.

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


Rectangle.cpp (Implementation File)
Rectangle.h is included in Rectangle.cpp
Rectangle.h (Specification File)
Rectangle.h is included in Pr13-4.cpp
Pr13-4.cpp (Main Program File)
Rectangle.cpp is compiled and Rectangle.obj is created
Pr13-4.cpp is compiled and Pr13-4.obj is created
Rectangle.obj (Object File)
Pr13-4.obj (Object File)
Rectangle.obj and Pr13-4.obj are linked and Pr13-4.exe is created
Pr13-4.exe (Executable File)

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)

```
// Specification file for the Rectangle class
// This version uses some inline member functions.
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
   private:
      double width;
      double length;
```

```
    public:
       void setWidth(double);
       void setLength(double);

       double getWidth() const
           { return width; }

       double getLength() const
           { return length; }

       double getArea() const
           { return width * length; }
};
#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)

```
// Implementation file for the Rectangle class.
// In this version of the class, the getWidth, getLength,
// and getArea functions are written inline in Rectangle.h.
#include "Rectangle.h"    // Needed for the Rectangle class
#include <iostream>       // Needed for cout
#include <cstdlib>        // Needed for the exit function
using namespace std;

//***********************************************************
// setWidth sets the value of the member variable width.   *
//***********************************************************

void Rectangle::setWidth(double w)
{
    if (w >= 0)
       width = w;
    else
    {
       cout << "Invalid width\n";
       exit(EXIT_FAILURE);
    }
}

//***********************************************************
// setLength sets the value of the member variable length. *
//***********************************************************

```

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

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

```
// This program demonstrates a constructor.
#include <iostream>
using namespace std;

// Demo class declaration.

class Demo
{
public:
     Demo();     // Constructor
};

Demo::Demo()
{
     cout << "Welcome to the constructor!\n";
}

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

```

```
int main()
{
    Demo demoObject; // Define a Demo object;

    cout << "This program demonstrates an object\n";
    cout << "with a constructor.\n";
    return 0;
}
```

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)

```
// Specification file for the Rectangle class
// This version has a constructor.
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
   private:
      double width;
      double length;
   public:
       Rectangle();                 // Constructor
       void setWidth(double);
       void setLength(double);

       double getWidth() const
          { return width; }

       double getLength() const
          { return length; }

       double getArea() const
          { return width * length; }
};
#endif
```

Contents of `Rectangle.cpp` (Version 3)

```
// Implementation file for the Rectangle class.
// This version has a constructor.
#include "Rectangle.h"    // Needed for the Rectangle class
#include <iostream>       // Needed for cout
#include <cstdlib>        // Needed for the exit function
using namespace std;

//**********************************************************
// The constructor initializes width and length to 0.0.    *
//**********************************************************

Rectangle::Rectangle()
{
    width = 0.0;
    length = 0.0;
}

//**********************************************************
// setWidth sets the value of the member variable width.   *
//**********************************************************

```

```
void Rectangle::setWidth(double w)
{
     if (w >= 0)
          width = w;
     else
     {
          cout << "Invalid width\n";
          exit(EXIT_FAILURE);
     }
}

//***********************************************************
// setLength sets the value of the member variable length. *
//***********************************************************

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

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

```
// This program uses the Rectangle class's constructor.
#include <iostream>
#include "Rectangle.h" // Needed for Rectangle class
using namespace std;

int main()
{
     Rectangle box;     // Define an instance of the Rectangle class

     // Display the rectangle's data.
     cout << "Here is the rectangle's data:\n";
     cout << "Width: " << box.getWidth() << endl;
     cout << "Length: " << box.getLength() << endl;
     cout << "Area: " << box.getArea() << endl;
     return 0;
}
```

(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)

```
// Specification file for the Rectangle class
// This version has a constructor.
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
    private:
        double width;
        double length;
    public:
        Rectangle(double, double);   // Constructor
        void setWidth(double);
        void setLength(double);

        double getWidth() const
            { return width; }

        double getLength() const
            { return length; }

        double getArea() const
            { return width * length; }
};
#endif
```

Contents of `Rectangle.cpp` (Version 4)

```
// Implementation file for the Rectangle class.
// This version has a constructor that accepts arguments.
#include "Rectangle.h"   // Needed for the Rectangle class
#include <iostream>      // Needed for cout
#include <cstdlib>       // Needed for the exit function
using namespace std;

//***********************************************************
// The constructor accepts arguments for width and length. *
//***********************************************************

Rectangle::Rectangle(double w, double len)
{
    width = w;
    length = len;
}

//***********************************************************
// setWidth sets the value of the member variable width.   *
//***********************************************************

void Rectangle::setWidth(double w)
```

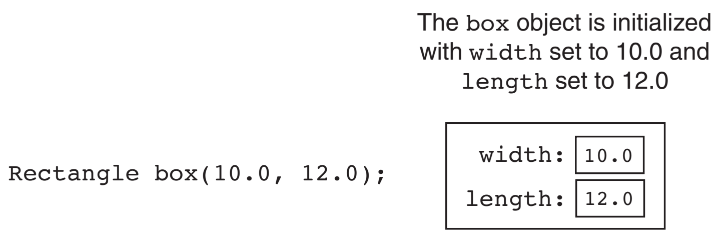
```
23  {
24      if (w >= 0)
25          width = w;
26      else
27      {
28          cout << "Invalid width\n";
29          exit(EXIT_FAILURE);
30      }
31  }
32
33  //***********************************************************
34  // setLength sets the value of the member variable length. *
35  //***********************************************************
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


The box object is initialized with width set to 10.0 and length set to 12.0
Rectangle box(10.0, 12.0);
width: 10.0
length: 12.0

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

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

```

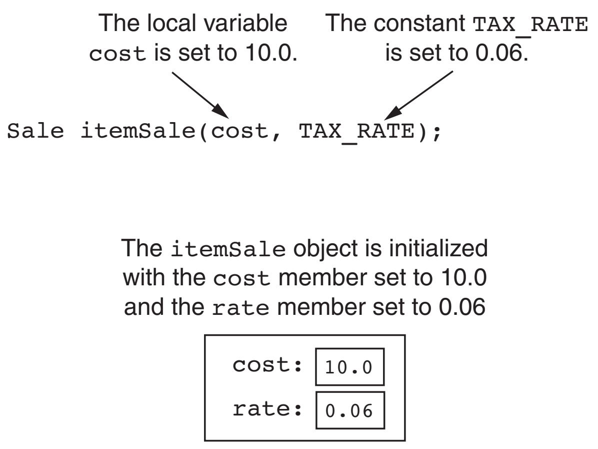
```
 7  int main()
 8  {
 9     const double TAX_RATE = 0.06;  // 6 percent sales tax rate
10     double cost;                   // To hold the item cost
11
12     // Get the cost of the item.
13     cout << "Enter the cost of the item: ";
14     cin >> cost;
15
16     // Create a Sale object for this 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


The local variable cost is set to 10.0.
The constant TAX_RATE is set to 0.06.
The itemSale object is initialized with the cost member set to 10.0 and the rate member set to 0.06
cost: 10.0
rate: 0.06

```
Sale itemSale(cost, TAX_RATE);
```

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)

```
// This version of the Sale class uses a default argument
// in the constructor.
#ifndef SALE_H
#define SALE_H

class Sale
{
private:
   double itemCost;    // Cost of the item
   double taxRate;     // Sales tax rate
public:
   Sale(double cost, double rate = 0.05)
      { itemCost = cost;
        taxRate = rate; }

   double getItemCost() const
      { return itemCost; }

   double getTaxRate() const
      { return taxRate; }

   double getTax() const
      { return (itemCost * taxRate); }

   double getTotal() const
      { return (itemCost + getTax()); }
};
#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

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

```

```
int main()
{
    double cost;  // To hold the item cost

    // Get the cost of the item.
    cout << "Enter the cost of the item: ";
    cin >> cost;

    // Create a Sale object for this transaction.
    // Specify the item cost, but use the default
    // tax rate of 5 percent.
    Sale itemSale(cost);

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

    // Display the sales tax and total.
    cout << "The amount of sales tax is $"
         << itemSale.getTax() << endl;
    cout << "The total of the sale is $";
    cout << itemSale.getTotal() << endl;
    return 0;
}
```

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

```
// This program demonstrates a destructor.
#include <iostream>
using namespace std;

class Demo
{
public:
    Demo();      // Constructor
    ~Demo();     // Destructor
};

Demo::Demo()
{
    cout << "Welcome to the constructor!\n";
}

Demo::~Demo()
{
    cout << "The destructor is now running.\n";
}

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

int main()
{
    Demo demoObject;  // Define a demo object;

    cout << "This program demonstrates an object\n";
    cout << "with a constructor and destructor.\n";
    return 0;
}
```

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)

```
// Specification file for the Contact class.
#ifndef CONTACTINFO_H
#define CONTACTINFO_H
#include <cstring>   // Needed for strlen and strcpy

// ContactInfo class declaration.
class ContactInfo
{
private:
    char *name;    // The name
    char *phone;   // The phone number
public:
    // Constructor
    ContactInfo(char *n, char *p)
    { // Allocate just enough memory for the name and phone number.
        name = new char[strlen(n) + 1];
        phone = new char[strlen(p) + 1];

        // Copy the name and phone number to the allocated memory.
        strcpy(name, n);
        strcpy(phone, p); }

    // Destructor
    ~ContactInfo()
    { delete [] name;
      delete [] phone; }

    const char *getName() const
    { return name; }

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

```
// This program demonstrates a class with a destructor.
#include <iostream>
#include "ContactInfo.h"
using namespace std;

int main()
{
    // Define a ContactInfo object with the following data:
    // Name: Kristen Lee Phone Number: 555-2021
    ContactInfo entry("Kristen Lee", "555-2021");

    // Display the object's data.
    cout << "Name: " << entry.getName() << endl;
    cout << "Phone Number: " << entry.getPhoneNumber() << endl;
    return 0;
}
```

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`

```
// This class has overloaded constructors.
#ifndef INVENTORYITEM_H
#define INVENTORYITEM_H
#include <string>
using namespace std;

class InventoryItem
{
private:
   string description; // The item description
   double cost;        // The item cost
   int units;          // Number of units on hand
public:
   // Constructor #1
   InventoryItem()
      { // Initialize description, cost, and units.
        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)

```
// Contact class specification file (version 2)
#ifndef CONTACTINFO_H
#define CONTACTINFO_H
#include <cstring>   // Needed for strlen and strcpy

// ContactInfo class declaration.
class ContactInfo
{
private:
   char *name;   // The contact's name
   char *phone;  // The contact's phone number

   // Private member function: initName
   // This function initializes the name attribute.
   void initName(char *n)
   { name = new char[strlen(n) + 1];
     strcpy(name, n); }

   // Private member function: initPhone
   // This function initializes the phone attribute.
   void initPhone(char *p)
   { phone = new char[strlen(p) + 1];
     strcpy(phone, p); }
public:
   // Constructor
   ContactInfo(char *n, char *p)
   { // Initialize the name attribute.
     initName(n);

     // Initialize the phone attribute.
     initPhone(n); }

   // Destructor
   ~ContactInfo()
   { delete [] name;
     delete [] phone; }

   const char *getName() const
   { return name; }

   const char *getPhoneNumber() const
   { return phone; }
};
#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

```
// This program demonstrates an array of class objects.
#include <iostream>
#include <iomanip>
#include "InventoryItem.h"
using namespace std;

int main()
{
   const int NUM_ITEMS = 5;
   InventoryItem inventory[NUM_ITEMS] = {
                  InventoryItem("Hammer", 6.95, 12),
                  InventoryItem("Wrench", 8.75, 20),
                  InventoryItem("Pliers", 3.75, 10),
                  InventoryItem("Ratchet", 7.95, 14),
                  InventoryItem("Screwdriver", 2.50, 22) };

```

```
    cout << setw(14) <<"Inventory Item"
         << setw(8) << "Cost" << setw(8)
         << setw(16) << "Units on Hand\n";
    cout << "-------------------------------------\n";

    for (int i = 0; i < NUM_ITEMS; i++)
    {
       cout << setw(14) << inventory[i].getDescription();
       cout << setw(8) << inventory[i].getCost();
       cout << setw(7) << inventory[i].getUnits() << endl;
    }

    return 0;
}
```

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`

```
// Specification file for the Account class.
#ifndef ACCOUNT_H
#define ACCOUNT_H

class Account
{
private:
   double balance;          // Account balance
   double interestRate;     // Interest rate for the period
   double interest;         // Interest earned for the period
   int transactions;        // Number of transactions
public:
   Account(double iRate = 0.045, double bal = 0)
      { balance = bal;
        interestRate = iRate;
        interest = 0;
        transactions = 0; }

   void setInterestRate(double iRate)
      { interestRate = iRate; }

   void makeDeposit(double amount)
      { balance += amount; transactions++; }

   void withdraw(double amount); // Defined in Account.cpp

   void calcInterest()
      { interest = balance * interestRate; balance += interest; }

   double getInterestRate() const
      { return interestRate; }

   double getBalance() const
      { return balance; }

   double getInterest() const
      { return interest; }

   int getTransactions() const
      { return transactions; }
};
#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`

```
// Implementation file for the Account class.
#include "Account.h"

bool Account::withdraw(double amount)
{
   if (balance < amount)
      return false; // Not enough in the account
   else
   {
      balance -= amount;
      transactions++;
      return true;
   }
}
```

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)*

```
    cin.ignore();
    acnt.makeDeposit(dollars);
}

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

void withdraw(Account &acnt)
{
    double dollars;

    cout << "Enter the amount of the withdrawal: ";
    cin >> dollars;
    cin.ignore();
    if (!acnt.withdraw(dollars))
        cout << "ERROR: Withdrawal amount too large.\n\n";
}
```

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`

```
// Specification file for the Die class
#ifndef DIE_H
#define DIE_H

class Die
{
private:
     int sides;    // Number of sides
     int value;    // The die's value

public:
     Die(int = 6);     // Constructor
     void roll();      // Rolls the die
     int getSides();   // Returns the number of sides
     int getValue();   // Returns the die's value
};
#endif
```

Contents of `Die.cpp`

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

```
#include "Die.h"
using namespace std;

//*******************************************************
// The constructor accepts an argument for the number  *
// of sides for the die, and performs a roll.          *
//*******************************************************
Die::Die(int numSides)
{
    // Get the system time.
    unsigned seed = time(0);

    // Seed the random number generator.
    srand(seed);

    // Set the number of sides.
    sides = numSides;

    // Perform an initial roll.
    roll();
}

//*******************************************************
// The roll member function simulates the rolling of   *
// the die.                                            *
//*******************************************************
void Die::roll()
{
  // Constant for the minimum die value
  const int MIN_VALUE = 1; // Minimum die value

  // Get a random value for the die.
  value = (rand() % (sides - MIN_VALUE + 1)) + MIN_VALUE;
}

//*******************************************************
// The getSides member function returns the number of  *
// for this die.                                       *
//*******************************************************
int Die::getSides()
{
  return sides;
}

//********************************************************
// The getValue member function returns the die's value.*
//********************************************************
int Die::getValue()
{
  return value;
}
```

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

```
// This program simulates the rolling of dice.
#include <iostream>
#include "Die.h"
using namespace std;

int main()
{
    const int DIE1_SIDES = 6;     // Number of sides for die #1
    const int DIE2_SIDES = 12;    // Number of sides for die #2
    const int MAX_ROLLS = 5;      // Number of times to roll

    // Create two instances of the Die class.
    Die die1(DIE1_SIDES);
    Die die2(DIE2_SIDES);

    // Display the initial state of the dice.
    cout << "This simulates the rolling of a "
         << die1.getSides() << " sided die and a "
         << die2.getSides() << " sided die.\n";

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`

```
// Specification file for the IntegerList class.
#ifndef INTEGERLIST_H
#define INTEGERLIST_H

class IntegerList
{
private:
   int *list;                    // Pointer to the array.
   int numElements;              // Number of elements.
   bool isValid(int);            // Validates subscripts.
public:
   IntegerList(int);              // Constructor
   ~IntegerList();                // Destructor
   void setElement(int, int);     // Sets an element to a value.
   void getElement(int, int&);    // Returns an element.
};
#endif
```

Contents of `IntegerList.cpp`

```
// Implementation file for the IntegerList class.
#include <iostream>
#include <cstdlib>
#include "IntegerList.h"
using namespace std;

//***********************************************************
// The constructor sets each element to zero.               *
//***********************************************************

IntegerList::IntegerList(int size)
{
   list = new int[size];
   numElements = size;
   for (int ndx = 0; ndx < size; ndx++)
      list[ndx] = 0;
}
```

```

//*********************************************************
// The destructor releases allocated memory.              *
//*********************************************************

IntegerList::~IntegerList()
{
   delete [] list;
}

//***********************************************************
// isValid member function.                                 *
// This private member function returns true if the argument *
// is a valid subscript, or false otherwise.                *
//***********************************************************

bool IntegerList::isValid(int element) const
{
   bool status;

   if (element < 0 || element >= numElements)
      status = false;
   else
      status = true;
   return status;
}

//**********************************************************
// setElement member function.                             *
// Stores a value in a specific element of the list. If an *
// invalid subscript is passed, the program aborts.        *
//**********************************************************

void IntegerList::setElement(int element, int value)
{
   if (isValid(element))
      list[element] = value;
   else
   {
      cout << "Error: Invalid subscript\n";
      exit(EXIT_FAILURE);
   }
}

//*********************************************************
// getElement member function.                            *
// Returns the value stored at the specified element.     *
// If an invalid subscript is passed, the program aborts. *
//*********************************************************

int IntegerList::getElement(int element) const
{
    if (isValid(element))
       return list[element];
    else
    {
       cout << "Error: Invalid subscript\n";
       exit(EXIT_FAILURE);
    }
 }
```

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

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

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

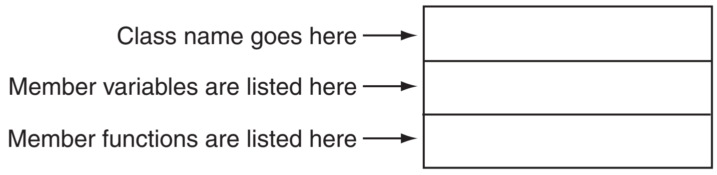
```
* * * * * * * * * * * * * * * * * * * *
9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
Error: Invalid subscript
```

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


Class name goes here
Member variables are listed here
Member functions are listed here

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:

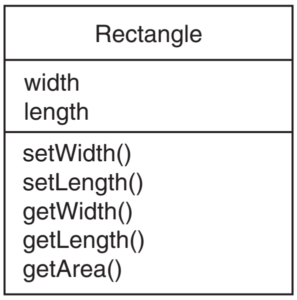
- `width`
- `length`

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

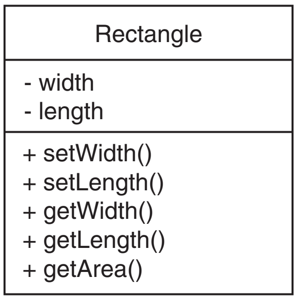


The UML diagram in Figure 13-19 tells us the name of the class, the names of the member variables, and the names of the member functions. The UML diagram in Figure 13-19 does not convey many of the class details, however, such as access specification, member variable data types, parameter data types,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

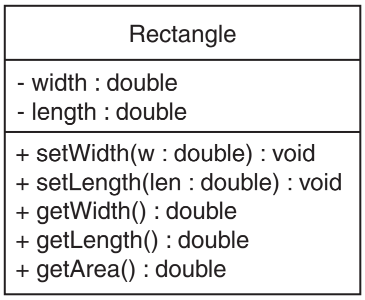
```
+ getLength() : double
```

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.

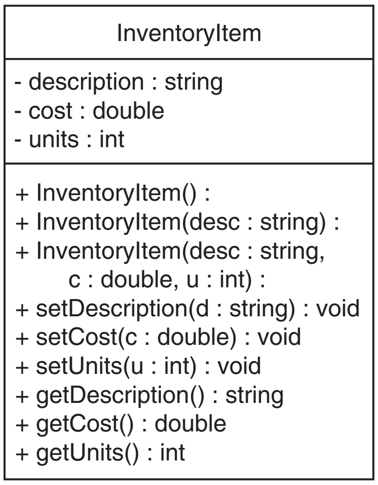
Figure 13-21



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


InventoryItem
- description : string
- cost : double
- units : int
+ InventoryItem() :
+ InventoryItem(desc : string) :
+ InventoryItem(desc : string,
c : double, u : int) :
+ setDescription(d : string) : void
+ setCost(c : double) : void
+ setUnits(u : int) : void
+ getDescription() : string
+ getCost() : double
+ getUnits() : int

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

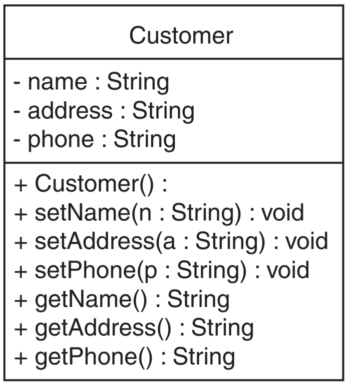
From this list we can see that the `Customer` class will have a constructor, as well as accessor and mutator functions for each of its attributes. Figure 13-23 shows a UML diagram for the `Customer` class.

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

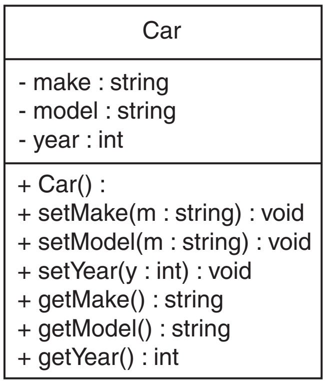


Now let's identify the class's member functions. In the context of our problem domain, what must the `Car` class do? Once again, the only obvious actions are the standard set of member functions that we will find in most classes (constructors, accessors, and mutators). Specifically, the actions are:

- 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



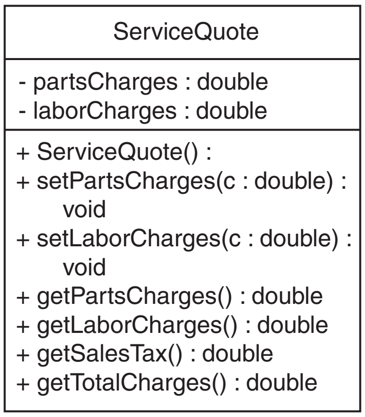
The `ServiceQuote` Class

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


ServiceQuote
- partsCharges : double
- laborCharges : double
+ ServiceQuote() :
+ setPartsCharges(c : double) : void
+ setLaborCharges(c : double) : void
+ getPartsCharges() : double
+ getLaborCharges() : double
+ getSalesTax() : double
+ getTotalCharges() : double

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.