CHAPTER 10

Object-Oriented C++

After studying this chapter, you will be able to:

- Create a simple programmer-defined class
- Use inheritance to create a derived C++ class

This chapter covers topics that include the material covered in Chapters 10 and 11 in *Programming Logic and Design, Eighth Edition*, by Joyce Farrell.

A Programmer-Defined Class

Before you continue with this chapter, you should take a moment to review the object-oriented terminology (class, attribute, and method) presented in Chapter 1 of this book and in Chapters 10 and 11 of *Programming Logic and Design, Eighth Edition*.

You have been using prewritten classes, objects, and methods throughout this book. For example, you have used the `open` method that is a member of the `ifstream` and `ofstream` class to open input and output files. In this section, you learn how to create your own class that includes attributes and methods of your choice. In programming terminology, a class created by the programmer is referred to as a **programmer-defined class** or a **custom class**.

To review, procedural programming focuses on declaring data and defining functions separate from the data and then calling those functions to operate on the data. This is the style of programming you have been using in Chapters 1 through 9 of this book. Object-oriented programming is different from procedural programming. Object-oriented programming focuses on an application's data and the functions you need to manipulate that data. The data and functions (which are called "methods" in object-oriented programming) are **encapsulated**, or contained within, a class. Objects are created as an instance of a class. The program tells an object to perform tasks by passing messages to it. Such a message consists of an instruction to execute one of the class's methods. The class method then manipulates the data (which is part of the object itself).

Creating a Programmer-Defined Class

In Chapter 10 of *Programming Logic and Design,* you studied pseudocode for the `Employee` class. This pseudocode is shown in Figure 10-1. The C++ code that implements the `Employee` class is shown in Figure 10-2.

```
1     class Employee
2        string lastName
3        num hourlyWage
4        num weeklyPay
5
6        void setLastName(string name)
7           lastName = name
8        return
9
10       void setHourlyWage(num wage)
11          hourlyWage = wage
12          calculateWeeklyPay()
13       return
```

Figure 10-1 Pseudocode for `Employee` class *(continues)*

(continued)

```
14
15        string getLastName()
16        return lastName
17
18        num getHourlyWage()
19        return hourlyWage
20
21        num getWeeklyPay()
22        return weeklyPay
23
24        void calculateWeeklyPay()
25           num WORK_WEEK_HOURS = 40
26           weeklyPay = hourlyWage * WORK_WEEK_HOURS
27        return
28     endClass
```

Figure 10-1 Pseudocode for `Employee` class

```
1     // Employee.cpp
2     #include <string>
3     using namespace std;
4     class Employee
5     {
6        public:
7           void setLastName(string);
8           void setHourlyWage(double);
9           double getHourlyWage();
10          double getWeeklyPay();
11          string getLastName();
12       private:
13          string lastName;
14          double hourlyWage;
15          double weeklyPay;
16          void calculateWeeklyPay();
17    }; // You end a class definition with a semicolon
18
19    void Employee::setLastName(string name)
20    {
21       lastName = name;
22       return;
23    }
24    void Employee::setHourlyWage(double wage)
25    {
26       hourlyWage = wage;
27       calculateWeeklyPay();
28       return;
29    }
```

Figure 10-2 `Employee` class implemented in C++ *(continues)*

(continued)

```
30      string Employee::getLastName()
31      {
32         return lastName;
33      }
34      double Employee::getHourlyWage()
35      {
36         return hourlyWage;
37      }
38      double Employee::getWeeklyPay()
39      {
40         return weeklyPay;
41      }
42      void Employee::calculateWeeklyPay()
43      {
44         const int WORK_WEEK_HOURS = 40;
45         weeklyPay = hourlyWage * WORK_WEEK_HOURS;
46         return;
47      }
```


Figure 10-2 `Employee` class implemented in C++

Looking at the pseudocode in Figure 10-1, you see that you begin creating a class by specifying that it is a class. In the C++ code in Figure 10-2, line 1 is a comment, line 2 is a preprocessor directive, `#include <string>`, and line 3 is a `using` statement, `using namespace std;`. (You learned about preprocessor directives and `using` statements in Chapter 1 of this book.) This is followed by the class declaration for the `Employee` class on line 4. The class declaration begins with the keyword `class`, which specifies that what follows is a C++ class. The opening curly brace on line 5 and the closing curly brace on line 17 mark the beginning and the end of the class.

Notice the closing curly brace on line 17 is followed by a semicolon. This closing curly brace and semicolon is *absolutely necessary* because it is part of the C++ syntax for creating a class. Omitting the semicolon is a common error.

Adding Attributes to a Class

The next step is to define the attributes (data) that are included in the `Employee` class. As shown on lines 2, 3, and 4 of the pseudocode in Figure 10-1, there are three attributes in this class: `string lastName`, `num hourlyWage`, and `num weeklyPay`.

Lines 13, 14, and 15 in Figure 10-2 include these attributes in the C++ version of the `Employee` class. Notice in the C++ code that `hourlyWage` and `weeklyPay` are defined using the `double` data type, and `lastName` is defined as a `string`. Also, notice that all three attributes are included in the **private** section of the class. The private section is defined by including the keyword `private` followed by a colon on line 12. As explained in *Programming Logic and Design*, this means the data cannot be accessed by any method (function) that is not part of the class. Programs that use the `Employee` class must use the methods that are part of the class to access private data.

Adding Methods to a Class

The next step is to add methods to the `Employee` class. The pseudocode versions of these methods, shown on lines 6 through 27 in Figure 10-1, are nonstatic methods. As you learned in Chapter 10 of *Programming Logic and Design*, **nonstatic methods** are methods that are meant to be used with an object created from a class. In other words, to use these methods, you must create an object of the `Employee` class first and then use that object to **invoke** (or call) the method.

The code shown in Figure 10-2 shows how to include methods in the `Employee` class using C++. The discussion starts with the set methods. You learned in *Programming Logic and Design* that **set methods** are methods that set the values of attributes (data fields) within the class. There are three data fields in the `Employee` class, but you will only add two set methods, `setLastName()` and `setHourlyWage()`. You do not add a `setWeeklyPay()` method because the `weeklyPay` data field is set by the `setHourlyWage()` method. The `setHourlyWage()` method uses another method, `calculateWeeklyPay()`, to accomplish this.

The two set methods, `setLastName()`, shown on lines 19 through 23 in Figure 10-2, and `setHourlyWage()`, shown on lines 24 through 29, are declared in the `public` section of the class on lines 7 and 8. The public section of the class is specified by using the keyword `public` followed by a colon on line 6. Including methods in the **public** section means that programs may use these methods to gain access to the private data. The `calculateWeeklyPay()` method, shown on lines 42 through 47 in Figure 10-2, is **private**, which means it can only be called from within another method that already belongs to the class. Notice the `calculateWeeklyPay()` method is declared in the `private` section in the `Employee` class on line 16. The `calculateWeeklyPay()` method is called from the `setHourlyWage()` method (line 27) and ensures that the class retains full control over when and how the `calculateWeeklyPay()` method is used.

The `setLastName()` method (lines 19 through 23) begins with the keyword `void`, followed by the name of the class, `Employee`, followed by the scope resolution operator (`::`), followed by the name of the method, `setLastName()`. The **scope resolution operator** is used to associate the method with the class. In this case, the `setLastName()` method is associated with the `Employee` class. The `setLastName()` method accepts one argument, `string name`, that is assigned to the private attribute `lastName`. This sets the value of `lastName`. The `setLastName()` method is a `void` method—that is, it returns nothing. Notice the declaration of the `setLastName()` method on line 7 agrees with the method definition; that is, both specify a return value of `void` and a single `string` argument.

The `setHourlyWage()` method (lines 24 through 29) accepts one argument, `double wage`, that is assigned to the private attribute `hourlyWage`. This sets the value of `hourlyWage`. Next, it calls the `private` method `calculateWeeklyPay()` on line 27. The `calculateWeeklyPay()` method does not accept arguments. Within the method, on line 44, a constant, `const int WORK_WEEK_HOURS`, is declared and initialized with the value 40. The `calculateWeeklyPay()` method then calculates the weekly pay (line 45) by multiplying the private attribute `hourlyWage` by `WORK_WEEK_HOURS`. The result is assigned to the private attribute `weeklyPay`. The `setHourlyWage()` method and the `calculateWeeklyPay()` method are `void` methods, which means they return nothing.

The final step in creating the `Employee` class is adding the get methods. **Get methods** are methods that return a value to the program using the class. The pseudocode in Figure 10-1 includes three get methods: `getLastName()` on lines 15 and 16, `getHourlyWage()` on lines 18 and 19, and `getWeeklyPay()` on lines 21 and 22. Lines 30 through 41 in Figure 10-2 illustrate the C++ version of the get methods in the `Employee` class.

It is not a requirement to use the prefix *get* when naming get methods, but by naming them using the word *get*, their intended purpose is clearer.

The three get methods are `public` methods and accept no arguments. The `getLastName()` method, shown on lines 30 through 33, returns a `string`, which is the value of the private attribute `lastName`. The `getHourlyWage()` method, shown on lines 34 through 37, returns a `double`, which is the value of the private attribute `hourlyWage`. The `getWeeklyPay()` method, shown on lines 38 through 41, also returns a `double`, which is the value of the private attribute `weeklyPay`. The three get methods are declared in the `Employee` class on lines 9, 10, and 11.

The `Employee` class is now complete and may be used in a C++ program. The `Employee` class does not contain a `main()` method (function) because it is not an application but rather a class that an application may now use to instantiate objects. The completed `Employee` class is included in the student files provided for this book, in a file named `Employee.cpp`.

Figure 10-3 illustrates a program named `MyEmployeeClassProgram` that uses the `Employee` class.

```
1    // This program uses the programmer-defined Employee class.
2    #include "Employee.cpp"
3    #include <iostream>
4    using namespace std;
5
6    int main()
7    {
8        const double LOW = 9.00;
9        const double HIGH = 14.65;
10       Employee myGardener;
11
12       myGardener.setLastName("Greene");
13       myGardener.setHourlyWage(LOW);
14       cout << "My gardener makes " << myGardener.getWeeklyPay()
             << " per week." << endl;
15
16       myGardener.setHourlyWage(HIGH);
17       cout << "My gardener makes " << myGardener.getWeeklyPay()
             << " per week." << endl;
18       return 0;
19   }
```

Figure 10-3 `MyEmployeeClassProgram` that uses the `Employee` class

As shown in Figure 10-3, the `MyEmployeeClassProgram` begins with a comment on line 1, followed by the preprocessor directive `#include "Employee.cpp"` on line 2. This preprocessor directive instructs the C++ compiler to include the file named `Employee.cpp`. The `Employee.cpp` file contains the `Employee` class declaration and the implementation of the methods defined in the `Employee` class. Notice the file named `Employee.cpp` is enclosed in double quotes. The double quotes specify that this is a user-created file rather than a C++ header file. C++ header file names are surrounded by angle brackets as shown on line 3, which instructs the C++ compiler to include the C++ `iostream` header file. Line 4 includes a `using` statement that specifies the `std` namespace.

Since this is a C++ program, it must contain a `main()` function. The `main()` function begins on line 6. On lines 7 and 19, you see the opening and closing curly braces that define the beginning and end of the `main()` function. Within the `main()` function, two constants, `LOW` and `HIGH`, are declared and initialized on lines 8 and 9, respectively. Next, on line 10, an `Employee` object (an instance of the `Employee` class) is created with the following statement:

```
Employee myGardener;
```

In C++, a statement that creates a new object consists of the class name followed by the object's name. In the preceding example, the class is `Employee` and the name of the object is `myGardener`.

As you learned in *Programming Logic and Design,* a **constructor** is a method that creates an object. You also learned that you can use a **default constructor**, which is a constructor that expects no arguments and is created automatically by the compiler for every class you write. The `Employee` constructor used in the `MyEmployeeClassProgram` is an example of a prewritten default constructor.

You can also write your own constructors if you wish. You will learn more about additional constructors in C++ courses you take after this course.

Once the `myGardener` object is created, you can use `myGardener` to invoke the set methods to set the value of `lastName` to `"Greene"` and the `hourlyWage` to `LOW`. The syntax used is shown in the following code sample:

```
myGardener.setLastName("Greene");
myGardener.setHourlyWage(LOW);
```

This is the syntax used to invoke a method with an instance (an object) of a class.

Notice the syntax *`objectName.methodName`*; the name of the object is separated from the name of the method by a dot. A dot is actually a period.

On line 14 in Figure 10-3, the program then prints `"My gardener makes "` (a string constant) followed by the return value of `myGardener.getWeeklyPay()`, followed by the string constant `" per week."`, followed by a newline. Here, the `myGardener` object is used again—this time to invoke the `getWeeklyPay()` method.

On line 16, `myGardener` invokes the set method `setHourlyWage()` to specify a new value for `hourlyWage`. This time, `hourlyWage` is set to `HIGH`. The program then prints (line 17) `"My gardener makes "` (a string constant) followed by the return value of `myGardener.getWeeklyPay()`, followed by the string constant `" per week."`, followed by a newline. The `return 0;` statement on line 18 ends the program. The output from this program is shown in Figure 10-4.

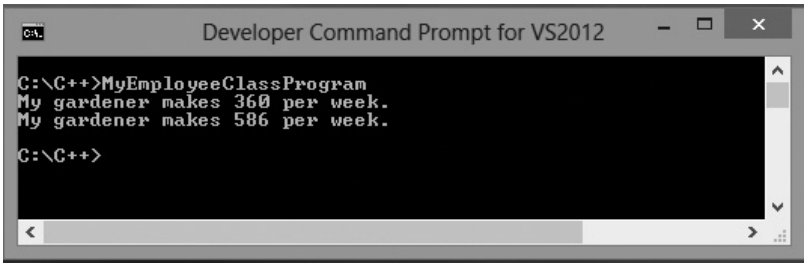

Figure 10-4 Output from the `MyEmployeeClassProgram`

You will find the completed program in a file named `MyEmployeeClassProgram.cpp` included with the student files for this book.

Exercise 10-1: Creating a Class in C++

In this exercise, you use what you have learned about creating and using a programmer-defined class. Study the following code, and then answer Questions 1–4.

```
class Circle
{
    public:
         void setRadius(double);
         double getRadius();
         double calculateCircumference();
         double calculateArea();
    private:
         double radius; // Radius of this circle
         const double PI = 3.14159;
};
void Circle::setRadius(double rad)
{
    radius = rad;
}
double Circle::getRadius()
{
    return radius;
}
```

```
double Circle::calculateCircumference()
{
    return (2 * PI * radius)
}
double Circle::calculateArea()
{
    return(PI * radius * radius)
}
```

In the following exercise, assume that a `Circle` object named `oneCircle` has been created in a program that uses the `Circle` class, and `radius` is given a value as shown in the following code:

```
Circle oneCircle;
oneCircle.setRadius(4.5);
```

1. What is the output when the following line of C++ code executes?

   ```
   cout << "The circumference is : " << oneCircle.calculateCircumference();
   ```

   _____

2. Is the following a legal C++ statement? Why or why not?

   ```
   cout << "The area is : " << calculateArea();
   ```

   _____

3. Consider the following C++ code. What is the value stored in the `oneCircle` object's attribute named `radius`?

   ```
   oneCircle.setRadius(10.0);
   ```

   _____

4. Write the C++ code that will assign the circumference of `oneCircle` to a `double` variable named `circumference1`.

   _____

Lab 10-1: Creating a Class in C++

In this lab, you create a programmer-defined class and then use it in a C++ program. The program should create two `Rectangle` objects and find their area and perimeter. Use the `Circle` class that you worked with in Exercise 10-1 as a guide.

1. Open the class file named `Rectangle.cpp` using Notepad or the text editor of your choice.
2. In the `Rectangle` class, create two private attributes named `length` and `width`. Both `length` and `width` should be data type `double`.
3. Write `public` set methods to set the values for `length` and `width`.
4. Write `public` get methods to retrieve the values for `length` and `width`.

5. Write a `public calculateArea()` method and a `public calculatePerimeter()` method to calculate and return the area of the rectangle and the perimeter of the rectangle.
6. Save this class file, `Rectangle.cpp`, in a directory of your choice, and then open the file named `MyRectangleClassProgram.cpp`.
7. In the `MyRectangleClassProgram`, create two `Rectangle` objects named `rectangle1` and `rectangle2` using the default constructor as you saw in `MyEmployeeClassProgram.cpp`.
8. Set the length of `rectangle1` to `10.0` and the width to `5.0`. Set the length of `rectangle2` to `7.0` and the width to `3.0`.
9. Print the value of `rectangle1`'s perimeter and area, and then print the value of `rectangle2`'s perimeter and area.
10. Save `MyRectangleClassProgram.cpp` in the same directory as `Rectangle.cpp`.
11. Compile the source code file `MyRectangleClassProgram.cpp`.
12. Execute the program.
13. Record the output below.

Reusing C++ Classes

You can reuse classes and reuse the code that is written for the members of a class by taking advantage of inheritance. **Inheritance** allows you to create a new class that is based on an existing class. To use inheritance, you create a new class (called the **derived class**) that contains the members of the original class (called the **base class**). You can then modify the derived class by adding new members that allow it to behave in new ways. You can also redefine methods that are inherited from the base class if they do not meet your exact needs in the derived class.

For example, if you have an existing `Vehicle` class, you could derive a new class named `Automobile` from the `Vehicle` class. You could also derive a `TaxiCab` class, a `Motorcycle` class, or a `GoKart` class from the `Vehicle` class. In C++, any class can serve as a base class.

Defining a Derived Class

You begin by taking a look at the base class, `Vehicle`. Figure 10-5 contains the C++ code that implements the `Vehicle` class.


```
1     // Vehicle.cpp
2     #include <iostream>
3     using namespace std;
4     class Vehicle
5     {
6        public:
7           void setSpeed(double);
8           double getSpeed();
9           void accelerate(double);
10          void setFuelCapacity(double);
11          double getFuelCapacity();
12          void setMaxSpeed(double);
13          double getMaxSpeed();
14       private:
15          double fuelCapacity;
16          double maxSpeed;
17          double currentSpeed;
18    };
19
20    void Vehicle::setSpeed(double speed)
21    {
22       currentSpeed = speed;
23       return;
24    }
25
26    double Vehicle::getSpeed()
27    {
28       return currentSpeed;
29    }
30
```

Figure 10-5 `Vehicle` class implemented in C++ *(continues)*

(continued)

```
31     void Vehicle::accelarate(double mph)
32     {
33        if(currentSpeed + mph < maxSpeed)
34           currentSpeed = currentSpeed + mph;
35        else
36           cout << "This vehicle cannot go that fast." <<endl;
37     {
38
39     void Vehicle::setFuelCapacity(double fuel)
40     {
41        fuelCapacity = fuel;
42     }
43
44     double Vehicle::getFuelCapacity()
45     {
46        return fuelCapacity;
47     }
48
49     void Vehicle::setMaxSpeed(double max)
50     {
51        maxSpeed = max;
52     }
53
54     double Vehicle::getMaxSpeed()
55     {
56        return maxSpeed;
57     }
```

Figure 10-5 `Vehicle` class implemented in C++

The `Vehicle` class contains three private attributes: `fuelCapacity`, `maxSpeed`, and `currentSpeed` (lines 15 through 17). The `Vehicle` class also contains seven methods that are declared in the `public` section (lines 7 through 13). The methods are `setSpeed()`, `getSpeed()`, `accelerate()`, `setFuelCapacity()`, `getFuelCapacity()`, `setMaxSpeed()`, and `getMaxSpeed()`. The methods are implemented on lines 20 through 57. Read over the C++ code written for these methods to be sure you understand them. Three of the methods are set methods and are written in a similar manner as the set methods that belong to the `Employee` class discussed in the previous section. Three of the methods are get methods and are written in a similar manner as the get methods that belong to the `Employee` class discussed in the previous section. Note that the `accelerate()` method contains something you have not yet seen. The `if` statement that is part of this method (line 33) tests to prohibit this `Vehicle` from exceeding its maximum speed.

The C++ program `MyVehicleClassProgram.cpp` shown in Figure 10-6 uses the `Vehicle` class, and the output generated by this program is shown in Figure 10-7.

```
1     // This program uses the programmer-defined Vehicle class.
2     #include "Vehicle.cpp"
3     #include <iostream>
4     using namespace std;
5     int main()
6     {
7        Vehicle vehicleOne;
8
9        vehicleOne.setMaxSpeed(100.0);
10       vehicleOne.setSpeed(35.0);
11       vehicleOne.accelerate(10.0);
12       cout << "The current speed is " << vehicleOne.getSpeed() << endl;
13
14       vehicleOne.accelerate(60.0);
15       cout << "The current speed is " << vehicleOne.getSpeed() << endl;
16
17       return 0;
18    }
```

Figure 10-6 `MyVehicleClassProgram` that uses the `Vehicle` class

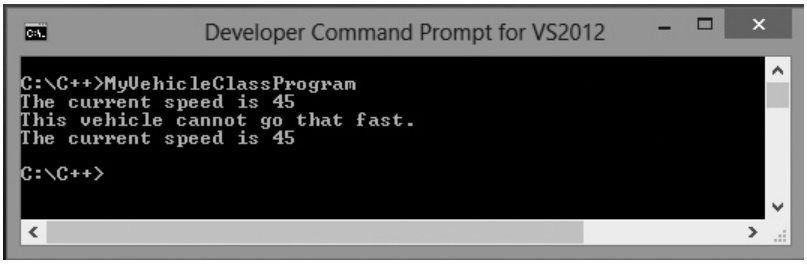

Figure 10-7 Output generated by the `MyVehicleClassProgram`

Looking at Figure 10-6, you see that line 1 is a comment that describes what the program does. On line 2, you see `#include "Vehicle.cpp"`. This preprocessor directive instructs C++ to include the file that contains the definition of the `Vehicle` class. Next, on lines 3 and 4, you see the `#include` directive that allows you to use `cout` in this program and the `using` statement you learned about in Chapter 1 of this book. The `main()` function begins on line 5.

On line 7, a `Vehicle` object is created named `vehicleOne`. Next, on line 9, the maximum speed for `vehicleOne` is set to 100.0 miles per hour (mph) using the `setMaxSpeed()` method. Line 10 uses the `setSpeed()` method to set the current speed for `vehicleOne` at 35.0 mph, and on line 11 `vehicleOne` accelerates by 10.0 mph using the `accelerate()` method. The

`cout` statement on line 12 produces the following output: `"The current speed is 45"`. Notice that the `getSpeed()` method is used in the `cout` statement to retrieve `vehicleOne`'s current speed. On line 14 `vehicleOne` invokes the `accelerate()` method again; this time to accelerate by 60.0 mph. Accelerating by 60.0 mph would cause `vehicleOne` to travel faster than its maximum speed and causes the `accelerate()` method to produce the following output: `"This vehicle cannot go that fast."` The `cout` statement on line 15 produces the following output: `"The current speed is 45"`. This output shows that the `accelerate` method will not allow this `Vehicle` object to travel faster than its maximum speed.

Now that the `Vehicle` class is complete, it can be used as a base class. You would like to use the `Vehicle` class to create a new `Automobile` class. You realize that an automobile is a vehicle just like the vehicle defined by the `Vehicle` class. You would like to be able to set a maximum speed, a current speed, and the fuel capacity for the automobile. You would also like to get its maximum speed, its current speed, and its fuel capacity. If you use inheritance, you will able to do all of these actions without having to write any new code. Additionally, you would like to accelerate the automobile, but if you want the automobile to accelerate beyond its maximum speed, you would like to change the message generated to say `"This automobile cannot go that fast"`. You also want to be able to indicate if the automobile has a convertible top. You realize you will have to modify the inherited `accelerate()` method and add new data to the `Automobile` class to indicate whether or not a particular `Automobile` object is a convertible, along with get and set methods to set the convertible top status and to get the convertible top status.

Figure 10-8 contains the C++ code that uses inheritance to create a derived class named `Automobile` using the `Vehicle` class as its base class.

```
1     // Automobile.cpp
2     #include "Vehicle.cpp"
3     #include <iostream>
4     using namespace std;
5     class Automobile : public Vehicle
6     {
7        public:
8           void accelerate(double);
9           void setConvertibleStatus(bool);
10          bool getConvertibleStatus();
11       private:
12          bool convertibleStatus;
13    };
14
15    void Automobile::accelerate(double mph)
16    {
17       if(getSpeed() + mph > getMaxSpeed())
18          cout << "This automobile cannot go that fast" << endl;
19       else
20          setSpeed(getSpeed() + mph);
21    }
22
23    void Automobile::setConvertibleStatus(bool status)
24    {
25       convertibleStatus = status;
26       return;
27    }
28
29    bool Automobile::getConvertibleStatus()
30    {
31       return convertibleStatus;
32    }
```

Figure 10-8 `Automobile` class implemented in C++

Looking at Figure 10-8, you see that line 1 is a comment, and on line 2 you see `#include "Vehicle.cpp"`. This preprocessor directive instructs C++ to include the file that contains the definition of the `Vehicle` class. C++ needs to know about the `Vehicle` class in order to create the `Automobile` class. Next, on lines 3 and 4, you see the `#include` directive that allows you to use `cout` in this class's methods and the `using` statement you learned about in Chapter 1 of this book. The `Automobile` class begins on line 5.

As shown on line 5, when you write the declaration for a derived class, you begin with the keyword `class` followed by the name of the derived class (`Automobile`). Next, you include a colon (`:`) followed by the keyword `public` and then the name of the class from which you are deriving the new class (`Vehicle`).

The keyword `public` specifies a derivation type. A **public** derivation means that all of the public members of the base class will be public in the derived class. Therefore, you do not have to repeat

these members in the derived class; you simply use the inherited members. This means that the new `Automobile` class inherits, and will be able to use, the `public` methods from the `Vehicle` class. These methods are `setSpeed()`, `getSpeed()`, `accelerate()`, `setFuelCapacity()`, `getFuelCapacity()`, `setMaxSpeed()`, and `getMaxSpeed()`. Now, in the `Automobile` class, you are able to use these `public` methods to gain access to the `private` members (`fuelCapacity`, `maxSpeed`, and `currentSpeed`) in the `Vehicle` class. In the derived `Automobile` class, you can also add new members or modify the inherited members. There are additional derivation types in C++, but in this book, you will always use the `public` derivation type. You will learn more about derivation types when you take additional courses in C++.

On line 12 in Figure 10-8, you see that one `private` data member, `convertibleStatus`, is added to the `Automobile` class. This `bool` data member is used to store a `true` or `false` value and specifies whether or not the `Automobile` has a convertible top. To provide the ability to set a value and get a value for the new `convertibleStatus` data member, two `public` methods, `setConvertibleStatus()` and `getConvertibleStatus()`, are added to the `Automobile` class on lines 9 and 10.

On line 8, you see the `accelerate()` method. This method has the same signature as the `accelerate()` method in the `Vehicle` class.

When you declare a method in a derived class (`Automobile`) with the same signature as a method in the base class (`Vehicle`), the derived class method **overrides** the inherited method. This means the method must be rewritten in the derived class and will be used with `Automobile` objects. The `accelerate()` method needs to be rewritten for the `Automobile` class because you want it to generate the message `"This automobile cannot go that fast"` rather than the message `"This vehicle cannot go that fast."`

In Chapter 9 of this book, you learned that a signature is made up of the method (function) name, the number of arguments it receives, and the data type of the arguments.

Now, the methods declared in the derived `Automobile` class must be written. The `accelerate()` method is written on lines 15 through 21. On line 17, you see the following `if` statement:

```
if(getSpeed() + mph > getMaxSpeed())
```

Notice the test portion of the `if` statement uses (1), the inherited `getSpeed()` method, to retrieve the value stored in the `private` data member `currentSpeed`, and (2), the inherited `getMaxSpeed()` method, to retrieve the value stored in the `private` data member `maxSpeed`. You do not have direct access to the `private` members of the base class (`Vehicle`); therefore, the inherited `public` methods must be used to gain access to the `private` data members `currentSpeed` and `maxSpeed`.

On line 18, you see the `cout` statement that generates the modified output: `"This automobile cannot go that fast"`. This statement executes if you try to accelerate the `Automobile` beyond its maximum speed. If the acceleration does not cause the `Automobile` to travel at a speed that is beyond its maximum speed, line 20 executes. Line 20 invokes the

inherited `setSpeed()` method and passes the acceleration amount (`mph`) added to the current speed. The current speed is retrieved by using the inherited `getSpeed()` method.

On lines 23 through 27, you see the `setConvertibleStatus()` method, which sets the value of the `private bool` data member `convertibleStatus`. And, on lines 29 through 32, you see the `getConvertibleStatus()` method that retrieves the value of the `private bool` data member `convertibleStatus`.

Using a Derived Class in a C++ Program

Now that the derived `Automobile` class is defined, you can use it in a C++ program. The C++ program, `MyAutomobileClassProgram.cpp`, shown in Figure 10-9 includes a comment on line 1. Line 2 includes the file, `Automobile.cpp`, that contains the `Automobile` class definition; line 3 includes `<iostream>`; and the `using` statement is on line 4. An `Automobile` object (`automobileOne`) is then created on line 7, and a `bool` variable (`convertible`) is declared on line 8.

```
1     // This program uses the programmer-defined Automobile class.
2     #include "Automobile.cpp"
3     #include <iostream>
4     using namespace std;
5     int main()
6     {
7        Automobile automobileOne;
8        bool convertible;
9
10       automobileOne.setMaxSpeed(100.0);
11       automobileOne.setSpeed(35.0);
12       automobileOne.accelerate(10.0);
13       cout << "The current speed is " << automobileOne.getSpeed() << endl;
14
15       automobileOne.accelerate(60.0);
16       cout << "The current speed is " << automobileOne.getSpeed() << endl;
17
18       automobileOne.setConvertibleStatus(true);
19       convertible = automobileOne.getConvertibleStatus();
20
21       if(convertible == true)
22          cout << "This automobile is a convertible" << endl;
23       else
24          cout << "This automobile is not a convertible" << endl;
25
26       return 0;
27    }
```

Figure 10-9 `MyAutomobileClassProgram` that uses the `Automobile` class

Next, on line 10, the maximum speed for `automobileOne` is set to 100.0 mph using the inherited `setMaxSpeed()` method. Line 11 uses the inherited `setSpeed()` method to set the current speed for `automobileOne` at 35.0 mph, and on line 12 `automobileOne` accelerates by 10.0 mph using the `accelerate()` method that was overridden in the `Automobile` class. The `cout` statement on line 13 produces the following output: `"The current speed is 45"`. Notice that the inherited `getSpeed()` method is used in the `cout` statement to retrieve `automobileOne`'s current speed. On line 15, `automobileOne` invokes the overridden `accelerate()` method again, this time to accelerate by 60.0 mph. Accelerating by 60.0 mph would cause `automobileOne` to travel faster than its maximum speed, and so it causes the `accelerate()` method that was overridden in the `Automobile` class to produce the following output: `"This automobile cannot go that fast"`. This output shows that the overridden `accelerate()` method is used with an `Automobile` object. The `cout` statement on line 16 produces the following output: `"The current speed is 45"`. This output shows that the `accelerate()` method will not allow this `Automobile` object to travel faster than its maximum speed.

Next, on line 18, `automobileOne` invokes the `setConvertibleStatus()` method and passes a `true` value that indicates this `Automobile` object is a convertible. The `getConvertibleStatus()` method is invoked on line 19, where its return value is assigned to the `bool` variable `convertible`. The value of `convertible` is tested in the `if` statement on line 21. If the value of `convertible` is `true`, the `cout` statement on line 22 executes, generating the following output: `"This automobile is a convertible"`. If the value of `convertible` is `false`, the `cout` statement on line 24 executes, generating the following output: `"This automobile is not a convertible"`.

The output from the `MyAutomobileClassProgram` is shown in Figure 10-10.

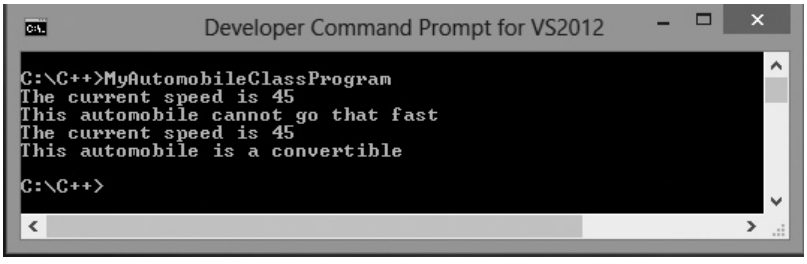

Figure 10-10 `MyAutomobileClassProgram` output

Exercise 10-2: Using Inheritance to Create a Derived Class in C++

In this exercise, you use what you have learned about using inheritance to create a derived class to answer Questions 1–4.

1. Which line of code is used to create a derived class named `SubWidget` from a base class named `Widget`?

   a. `class Widget : public SubWidget`

   b. `class SubWidget : base public Widget`

   c. `class Widget : derived public SubWidget`

   d. `class SubWidget : public Widget`

   ---

2. An advantage of using inheritance is:

   a. It maximizes the number of functions.

   b. It allows reuse of code.

   c. It requires no coding.

   ---

3. True or False: The methods in a derived class have direct access to the base class private data members.

   ---

4. True or False: A derived class may add new methods or override existing methods when inheriting from a base class.

Lab 10-2: Using Inheritance to Create a Derived Class in C++

In this lab, you create a derived class from a base class, and then use the derived class in a C++ program. The program should create two `Motorcycle` objects, and then set the `Motorcycle`'s speed, accelerate the `Motorcycle` object, and check its sidecar status. Use the `Vehicle` and `Automobile` classes that you worked with earlier in this chapter as a guide.

1. Open the file named `Motorcycle.cpp` using Notepad or the text editor of your choice.
2. Create the `Motorcycle` class by deriving it from the `Vehicle` class. Use a `public` derivation.
3. In the `Motorcycle` class, create a `private` attribute named `sidecar`. The `sidecar` attribute should be data type `bool`.
4. Write a `public` set method to set the value for `sidecar`.
5. Write a `public` get method to retrieve the value of `sidecar`.
6. Write a `public accelerate()` method. This method overrides the `accelerate()` method inherited from the `Vehicle` class. Change the message in the `accelerate()`

method so the following is displayed when the `Motorcycle` tries to accelerate beyond its maximum speed: `"This motorcycle cannot go that fast"`.

7. Save this class file, `Motorcycle.cpp`, in a directory of your choice, and then open the file named `MyMotorcycleClassProgram.cpp`.
8. In the `MyMotorcycleClassProgram`, create two `Motorcycle` objects named `motorcycleOne` and `motorcycleTwo`.
9. Set the `sidecar` value of `motorcycleOne` to `true` and the `sidecar` value of `motorcycleTwo` to `false`.
10. Set `motorcycleOne`'s maximum speed to 90 and `motorcycleTwo`'s maximum speed to 85.
11. Set `motorcycleOne`'s current speed to 65 and `motorcycleTwo`'s current speed to 60.
12. Accelerate `motorcycleOne` by 30 mph, and accelerate `motorcycleTwo` by 20 mph.
13. Print the current speed of `motorcycleOne` and `motorcycleTwo`.
14. Determine if `motorcycleOne` and `motorcycleTwo` have sidecars. If yes, display the following: `"This motorcycle has a sidecar"`. If not, display the following: `"This motorcycle does not have a sidecar"`.
15. Save `MyMotorcycleClassProgram.cpp` in the same directory as `Motorcycle.cpp`.
16. Compile the source code file `MyMotorcycleClassProgram.cpp`.
17. Execute the program.
18. Record the output below.

_____

_____

_____

_____